Contents

Windows Sockets 2

Af_irda.h

Overview

SOCKADDR_IRDA structure

In6addr.h

Overview

Inaddr.h

Overview

Mstcpip.h

Overview

ASSOCIATE_NAMERES_CONTEXT_INPUT structure

CONTROL_CHANNEL_TRIGGER_STATUS enumeration

INET_PORT_RANGE structure

INET_PORT_RESERVATION_INSTANCE structure

INET_PORT_RESERVATION_TOKEN structure

REAL_TIME_NOTIFICATION_SETTING_INPUT structure

REAL_TIME_NOTIFICATION_SETTING_OUTPUT structure

SOCKET_PEER_TARGET_NAME structure

SOCKET_SECURITY_PROTOCOL enumeration

SOCKET_SECURITY_QUERY_INFO structure

SOCKET_SECURITY_QUERY_TEMPLATE structure

SOCKET_SECURITY_SETTINGS structure

SOCKET_SECURITY_SETTINGS_IPSEC structure

SOCKET_USAGE_TYPE enumeration

TCP_INFO_v0 structure

TCP_INFO_v1 structure

TCP_INITIAL_RTO_PARAMETERS structure

TCPSTATE enumeration

TIMESTAMPING_CONFIG structure

TRANSPORT_SETTING_ID structure
Mswsock.h
Overview
AcceptEx function
GetAcceptExSockaddrs function
LPFN_CONNECTEX callback function
LPFN_DISCONNECTEX callback function
LPFN_RIOCLOSECOMPLETIONQUEUE callback function
LPFN_RIOCREATECOMPLETIONQUEUE callback function
LPFN_RIOCREATEREQUESTQUEUE callback function
LPFN_RIODEQUEUECOMPLETION callback function
LPFN_RIODEREGISTERBUFFER callback function
LPFN_RIONOTIFY callback function
LPFN_RIORECEIVE callback function
LPFN_RIORECEIVEEX callback function
LPFN_RIOREGISTERBUFFER callback function
LPFN_RIORESIZECOMPLETIONQUEUE callback function
LPFN_RIORESIZEREQUESTQUEUE callback function
LPFN_RIOSEND callback function
LPFN_RIOSENDEX callback function
LPFN_TRANSMITPACKETS callback function
LPFN_WSARECVMSG callback function
RIO_EXTENSION_FUNCTION_TABLE structure
RIO_NOTIFICATION_COMPLETION structure
RIO_NOTIFICATION_COMPLETION_TYPE enumeration
TRANSMIT_FILE_BUFFERS structure
TRANSMIT_PACKETS_ELEMENT structure
TransmitFile function
WSARecvEx function
Mswsockdef.h
Overview
RIO_BUF structure

RIORESULT structure
Nsemail.h
Overview
NAPI_DOMAIN_DESCRIPTION_BLOB structure
NAPI_PROVIDER_INSTALLATION_BLOB structure
NAPI_PROVIDER_LEVEL enumeration
NAPI_PROVIDER_TYPE enumeration
Nspapi.h
Overview
BLOB structure
CSADDR_INFO structure
EnumProtocolsA function
EnumProtocolsW function
GetAddressByNameA function
GetAddressByNameW function
GetNameByTypeA function
GetNameByTypeW function
GetServiceA function
GetServiceW function
GetTypeByNameA function
GetTypeByNameW function
NS_SERVICE_INFOA structure
NS_SERVICE_INFOW structure
PROTOCOL_INFOA structure
PROTOCOL_INFOW structure
SERVICE_ADDRESS structure
SERVICE_ADDRESSES structure
SERVICE_INFOA structure
SERVICE_INFOW structure
SERVICE_TYPE_INFO_ABSA structure
SERVICE_TYPE_INFO_ABSW structure
SERVICE_TYPE_VALUE_ABSA structure

SERVICE_TYPE_VALUE_ABSW structure
SetServiceA function
SetServiceW function
Socketapi.h
Overview
SetSocketMediaStreamingMode function
Sporder.h
Overview
WSCWriteNameSpaceOrder function
WSCWriteNameSpaceOrder32 function
WSCWriteProviderOrder function
WSCWriteProviderOrder32 function
Transportsettingcommon.h
Overview
TRANSPORT_SETTING_ID structure
Winsock.h
Overview
__WSAFDIsSet function
AcceptEx function
bind function
closesocket function
fd_set structure
FD_SET macro
GetAcceptExSockaddrs function
gethostbyaddr function
gethostbyname function
gethostname function
getpeername function
getprotobyname function
getprotobynumber function
getservbyname function
getservbyport function

getsockname function

getsockopt function

HOSTENT structure

htonl function

htons function

inet_addr function

inet_ntoa function

ioctlsocket function

LINGER structure

ntohl function

ntohs function

PROTOENT structure

recv function

recvfrom function

sendto function

SERVENT structure

setsockopt function

shutdown function

SOCKADDR structure

SOCKADDR_IN structure

TIMEVAL structure

TRANSMIT_FILE_BUFFERS structure

TransmitFile function

WSAAsyncGetHostByAddr function

WSAAsyncGetHostByName function

WSAAsyncGetProtoByName function

WSAAsyncGetProtoByNumber function

WSAAsyncGetServByName function

WSAAsyncGetServByPort function

WSAAsyncSelect function

WSACancelAsyncRequest function

WSACleanup function

WSADATA structure
WSAGetLastError function
WSARecvEx function
WSASetLastError function
WSAStartup function
Winsock2.h
Overview
__WSAFDIsSet function
accept function
AFPROTOCOLS structure
bind function
BLOB structure
closesocket function
connect function
fd_set structure
FD_SET macro
gethostbyaddr function
gethostbyname function
gethostname function
GetHostNameW function
getpeername function
getprotobyname function
getprotobynumber function
getservbyname function
getservbyport function
getsockname function
getsockopt function
HOSTENT structure
htond function
htonf function
htonl function
htonll function

htons function

in_addr structure

inet_addr function

inet_ntoa function

ioctlsocket function

LINGER structure

listen function

LPWSAOVERLAPPED_COMPLETION_ROUTINE callback function

ntohd function

ntohf function

ntohl function

ntohll function

ntohs function

ProcessSocketNotifications function

PROTOENT structure

recv function

recvfrom function

select function

send function

sendto function

SERVENT structure

setsockopt function

shutdown function

SOCK_NOTIFY_REGISTRATION structure

socket function

SocketNotificationRetrieveEvents function

TIMEVAL structure

WSAAccept function

WSAAddressToStringA function

WSAAddressToStringW function

WSAAsyncGetHostByAddr function

WSAAsyncGetHostByName function

WSAAsyncGetProtoByName function

WSAAsyncGetProtoByNumber function

WSAAsyncGetServByName function

WSAAsyncGetServByPort function

WSAAsyncSelect function

WSACancelAsyncRequest function

WSACancelBlockingCall function

WSACleanup function

WSACloseEvent function

WSACOMPLETION structure

WSAConnect function

WSAConnectByList function

WSAConnectByNameA function

WSAConnectByNameW function

WSACreateEvent function

WSADATA structure

WSADuplicateSocketA function

WSADuplicateSocketW function

WSAECOMPARATOR enumeration

WSAEnumNameSpaceProvidersA function

WSAEnumNameSpaceProvidersExA function

WSAEnumNameSpaceProvidersExW function

WSAEnumNameSpaceProvidersW function

WSAEnumNetworkEvents function

WSAEnumProtocolsA function

WSAEnumProtocolsW function

WSAEventSelect function

WSAGetLastError function

WSAGetOverlappedResult function

WSAGetQOSByName function

WSAGetServiceClassInfoA function

WSAGetServiceClassInfoW function

WSAGetServiceClassNameByClassIdA function
WSAGetServiceClassNameByClassIdW function
WSAHtonl function
WSAHtons function
WSAInstallServiceClassA function
WSAInstallServiceClassW function
WSAIoctl function
WSAIsBlocking function
WSAJoinLeaf function
WSALookupServiceBeginA function
WSALookupServiceBeginW function
WSALookupServiceEnd function
WSALookupServiceNextA function
WSALookupServiceNextW function
WSANAMESPACE_INFOA structure
WSANAMESPACE_INFOEXA structure
WSANAMESPACE_INFOEXW structure
WSANAMESPACE_INFOW structure
WSANETWORKEVENTS structure
WSANSCLASSINFOA structure
WSANSCLASSINFOW structure
WSANSPIoctl function
WSANtohl function
WSANtohs function
WSAOVERLAPPED structure
WSAPoll function
WSAPOLLFD structure
WSAPROTOCOL_INFOA structure
WSAPROTOCOL_INFOW structure
WSAPROTOCOLCHAIN structure
WSAProviderConfigChange function
WSAQUERYSET2A structure

WSAQUERYSET2W structure
WSAQUERYSETA structure
WSAQUERYSETW structure
WSARecv function
WSARecvDisconnect function
WSARecvFrom function
WSARemoveServiceClass function
WSAResetEvent function
WSASend function
WSASendDisconnect function
WSASendMsg function
WSASendTo function
WSASERVICECLASSINFOA structure
WSASERVICECLASSINFOW structure
WSASetBlockingHook function
WSASetEvent function
WSASetLastError function
WSASetServiceA function
WSASetServiceW function
WSASocketA function
WSASocketW function
WSAStartup function
WSAStringToAddressA function
WSAStringToAddressW function
WSAUnhookBlockingHook function
WSAVERSION structure
WSAWaitForMultipleEvents function
Ws2atm.h
Overview
ATM_ADDRESS structure
ATM_BHLI structure
ATM_BLLI structure

sockaddr_atm structure
Ws2def.h
Overview
ADDRINFO_DNS_SERVER structure
ADDRINFOA structure
ADDRINFOEX2A structure
ADDRINFOEX2W structure
ADDRINFOEX3 structure
ADDRINFOEX4 structure
ADDRINFOEX5 structure
ADDRINFOEX6 structure
ADDRINFOEXA structure
ADDRINFOEXW structure
ADDRINFOW structure
CSADDR_INFO structure
SOCKET_ADDRESS structure
SOCKET_PROCESSOR_AFFINITY structure
WSABUF structure
WSAMSG structure
Ws2ipdef.h
Overview
GROUP_FILTER structure
GROUP_REQ structure
GROUP_SOURCE_REQ structure
ICMP_ERROR_INFO structure
IN_PKTINFO structure
IN6_PKTINFO structure
INTERFACE_INFO structure
INTERFACE_INFO_EX structure
IP_MREQ structure
IP_MREQ_SOURCE structure
IP_MSFILTER structure

IPV6_MREQ structure
MULTICAST_MODE_TYPE enumeration
sockaddr_gen union
sockaddr_in6_old structure
Ws2spi.h
Overview
LPNSPCLEANUP callback function
LPNSPGETSERVICECLASSINFO callback function
LPNSPINSTALLSERVICECLASS callback function
LPNSPIOCTL callback function
LPNSPLOOKUPSERVICEBEGIN callback function
LPNSPLOOKUPSERVICEEND callback function
LPNSPLOOKUPSERVICENEXT callback function
LPNSPREMOVESERVICECLASS callback function
LPNSPSETSERVICE callback function
LPNSPV2CLEANUP callback function
LPNSPV2CLIENTSESSIONRUNDOWN callback function
LPNSPV2LOOKUPSERVICEBEGIN callback function
LPNSPV2LOOKUPSERVICEEND callback function
LPNSPV2LOOKUPSERVICENEXTEX callback function
LPNSPV2SETSERVICEEX callback function
LPNSPV2STARTUP callback function
LPWSPACCEPT callback function
LPWSPADDRESSTOSTRING callback function
LPWSPASYNCSELECT callback function
LPWSPBIND callback function
LPWSPCANCELBLOCKINGCALL callback function
LPWSPCLEANUP callback function
LPWSPCLOSESOCKET callback function
LPWSPCONNECT callback function
LPWSPDUPLICATESOCKET callback function
LPWSPENUMNETWORKEVENTS callback function

LPWSPEVENTSELECT callback function

LPWSPGETOVERLAPPEDRESULT callback function

LPWSPGETPEERNAME callback function

LPWSPGETQOSBYNAME callback function

LPWSPGETSOCKNAME callback function

LPWSPGETSOCKOPT callback function

LPWSPIOCTL callback function

LPWSPJOINLEAF callback function

LPWSPLISTEN callback function

LPWSPRECV callback function

LPWSPRECVDISCONNECT callback function

LPWSPRECVFROM callback function

LPWSPSELECT callback function

LPWSPSEND callback function

LPWSPSENDDISCONNECT callback function

LPWSPSENDTO callback function

LPWSPSETSOCKOPT callback function

LPWSPSHUTDOWN callback function

LPWSPSOCKET callback function

LPWSPSTRINGTOADDRESS callback function

NSP_ROUTINE structure

NSPStartup function

NSPV2_ROUTINE structure

WPUCloseEvent function

WPUCloseSocketHandle function

WPUCloseThread function

WPUCompleteOverlappedRequest function

WPUCreateEvent function

WPUCreateSocketHandle function

WPUFDIsSet function

WPUGetProviderPath function

WPUModifyIFSHandle function

WPUOpenCurrentThread function
WPUPostMessage function
WPUQueryBlockingCallback function
WPUQuerySocketHandleContext function
WPUQueueApc function
WPUResetEvent function
WPUSetEvent function
WSAAdvertiseProvider function
WSAProviderCompleteAsyncCall function
WSATHREADID structure
WSAUnadvertiseProvider function
WSC_PROVIDER_AUDIT_INFO structure
WSC_PROVIDER_INFO_TYPE enumeration
WSCDeinstallProvider function
WSCDeinstallProvider32 function
WSCEnableNSProvider function
WSCEnableNSProvider32 function
WSCEnumNameSpaceProviders32 function
WSCEnumNameSpaceProvidersEx32 function
WSCEnumProtocols function
WSCEnumProtocols32 function
WSCGetApplicationCategory function
WSCGetProviderInfo function
WSCGetProviderInfo32 function
WSCGetProviderPath function
WSCGetProviderPath32 function
WSCInstallNameSpace function
WSCInstallNameSpace32 function
WSCInstallNameSpaceEx function
WSCInstallNameSpaceEx32 function
WSCInstallProvider function
WSCInstallProvider64_32 function

WSCInstallProviderAndChains function
WSCInstallProviderAndChains64_32 function
WSCInstallQOSTemplate function
WSCRemoveQOSTemplate function
WSCSetApplicationCategory function
WSCSetProviderInfo function
WSCSetProviderInfo32 function
WSCUnInstallNameSpace function
WSCUnInstallNameSpace32 function
WSCUpdateProvider function
WSCUpdateProvider32 function
WSPDATA structure
WSPPROC_TABLE structure
WSPStartup function
WSPUPCALLTABLE structure
Ws2tcpip.h
Overview
freeaddrinfo function
FreeAddrInfoEx function
FreeAddrInfoExW function
FreeAddrInfoW function
gai_strerrorA function
gai_strerrorW function
getaddrinfo function
GetAddrInfoExA function
GetAddrInfoExCancel function
GetAddrInfoExOverlappedResult function
GetAddrInfoExW function
GetAddrInfoW function
getipv4sourcefilter function
getnameinfo function
GetNameInfoW function

getsourcefilter function
inet_ntop function
inet_pton function
InetNtopW function
InetPtonW function
SetAddrInfoExA function
SetAddrInfoExW function
setipv4sourcefilter function
setsourcefilter function
WSADeleteSocketPeerTargetName function
WSAGetFailConnectOnIcmpError function
WSAGetIcmpErrorInfo function
WSAGetIPUserMtu function
WSAGetUdpRecvMaxCoalescedSize function
WSAGetUdpSendMessageSize function
WSAImpersonateSocketPeer function
WSAQuerySocketSecurity function
WSARevertImpersonation function
WSASetFailConnectOnIcmpError function
WSASetIPUserMtu function
WSASetSocketPeerTargetName function
WSASetSocketSecurity function
WSASetUdpRecvMaxCoalescedSize function
WSASetUdpSendMessageSize function
Wsipv6ok.h
Overview
gethostbyaddr macro
gethostbyname macro
inet_addr macro
inet_ntoa macro
WSAAsyncGetHostByAddr macro
WSAAsyncGetHostByName macro

Wsnwlink.h
- Overview
- IPX_ADDRESS_DATA structure
- IPX_NETNUM_DATA structure
- IPX_SPXCONNSTATUS_DATA structure

Wsrm.h
- Overview
- eWINDOW_ADVANCE_METHOD enumeration
- RM_FEC_INFO structure
- RM_RECEIVER_STATS structure
- RM_SEND_WINDOW structure
- RM_SENDER_STATS structure

Wtypesbase.h
- Overview
- BLOB structure

Windows Sockets 2

10/22/2022 • 41 minutes to read • Edit Online

Overview of the Windows Sockets 2 technology.

To develop Windows Sockets 2, you need these headers:

- af_irda.h
- in6addr.h
- mstcpip.h
- mswsock.h
- mswsockdef.h
- nsemail.h
- nspapi.h
- socketapi.h
- sporder.h
- transportsettingcommon.h
- winsock.h
- ws2atm.h
- ws2spi.h
- ws2tcpip.h
- wsipv6ok.h
- wsnwlink.h
- wsrm.h

For programming guidance for this technology, see:

- Windows Sockets 2

Enumerations

CONTROL_CHANNEL_TRIGGER_STATUS Specifies the status from a query for the REAL_TIME_NOTIFICATION_CAPABILITY transport setting for a TCP socket that is used with ControlChannelTrigger to receive background network notifications in a Windows Store app.
eWINDOW_ADVANCE_METHOD The eWINDOW_ADVANCE_METHOD enumeration specifies the window advance mode used for Reliable Multicast.
MULTICAST_MODE_TYPE Specifies the filter mode for multicast group addresses.
NAPI_PROVIDER_LEVEL Specifies the provider authority level of a NS_EMAIL namespace provider for a given domain.

NAPI_PROVIDER_TYPE

Specifies the type of hosting expected for a namespace provider.

RIO_NOTIFICATION_COMPLETION_TYPE

Specifies the type of completion queue notifications to use with the RIONotify function when sending or receiving data using the Winsock registered I/O extensions.

SOCKET_SECURITY_PROTOCOL

Indicates the type of security protocol to be used on a socket to secure network traffic.

SOCKET_USAGE_TYPE

Used to specified the usage type for the socket.

TCPSTATE

Indicates the possible states of a Transmission Control Protocol (TCP) connection.

WSAECOMPARATOR

The Windows Sockets WSAECOMPARATOR enumeration type is used for version-comparison semantics in Windows Sockets 2.

WSC_PROVIDER_INFO_TYPE

Enumeration type is used to specify the information class of a layered service protocol (LSP) in Windows Sockets 2.

Functions

__WSAFDIsSet

The __WSAFDIsSet function (winsock2.h) specifies whether a socket is included in a set of socket descriptors.

__WSAFDIsSet

The __WSAFDIsSet function (winsock.h) specifies whether a socket is included in a set of socket descriptors.

accept

The accept function permits an incoming connection attempt on a socket.

AcceptEx

The AcceptEx function (mswsock.h) accepts a new connection, returns the local and remote address, and receives the first block of data sent by the client application.

AcceptEx

The AcceptEx function (winsock.h) accepts a new connection, returns the local and remote address, and receives the first block of data sent by the client application.

bind

The bind function associates a local address with a socket. (bind function (winsock2.h))

bind

The bind function (winsock.h) associates a local address with a socket.

closesocket

The closesocket function closes an existing socket. (closesocket function (winsock2.h))

closesocket

The closesocket function (winsock.h) closes an existing socket.

connect

The connect function establishes a connection to a specified socket.

EnumProtocolsA

The EnumProtocols function retrieves information about a specified set of network protocols that are active on a local host. (ANSI)

EnumProtocolsW

The EnumProtocols function retrieves information about a specified set of network protocols that are active on a local host. (Unicode)

FD_SET

The FD_SET macro (winsock2.h) is used by Windows Sockets (Winsock) functions and service providers to place sockets into a set.

FD_SET

The FD_SET macro (winsock.h) structure is used by Windows Sockets (Winsock) functions and service providers to place sockets into a set.

freeaddrinfo

Frees address information that the getaddrinfo function dynamically allocates in addrinfo structures.

FreeAddrInfoEx

The FreeAddrInfoEx function (ws2tcpip.h) frees address information that the GetAddrInfoEx function dynamically allocates in addrinfoex structures.

FreeAddrInfoExW

The FreeAddrInfoExW (Unicode) function (ws2tcpip.h) frees address information that the GetAddrInfoEx function dynamically allocates in addrinfoex structures.

FreeAddrInfoW

Frees address information that the GetAddrInfoW function dynamically allocates in addrinfoW structures.

gai_strerrorA

The gai_strerror function assists in printing error messages based on the EAI_* errors returned by the getaddrinfo function. (ANSI)

gai_strerrorW

The gai_strerror function assists in printing error messages based on the EAI_* errors returned by the getaddrinfo function. (Unicode)

GetAcceptExSockaddrs

The GetAcceptExSockaddrs function (mswsock.h) parses data obtained from a call to the AcceptEx function and passes local and remote addresses to a sockaddr structure.

GetAcceptExSockaddrs

The GetAcceptExSockaddrs function (winsock.h) parses the data obtained from a call to the AcceptEx function and passes the local and remote addresses to a sockaddr structure.

GetAddressByNameA

GetAddressByName is no longer available for use as of Windows Sockets 2. (ANSI)

GetAddressByNameW

GetAddressByName is no longer available for use as of Windows Sockets 2. (Unicode)

getaddrinfo

Provides protocol-independent translation from an ANSI host name to an address.

GetAddrInfoExA

Provides protocol-independent name resolution with additional parameters to qualify which namespace providers should handle the request. (ANSI)

GetAddrInfoExCancel

Cancels an asynchronous operation by the GetAddrInfoEx function.

GetAddrInfoExOverlappedResult

Gets the return code for an OVERLAPPED structure used by an asynchronous operation for the GetAddrInfoEx function.

GetAddrInfoExW

Provides protocol-independent name resolution with additional parameters to qualify which namespace providers should handle the request. (Unicode)

GetAddrInfoW

Provides protocol-independent translation from a Unicode host name to an address.

gethostbyaddr

The gethostbyaddr function retrieves the host information corresponding to a network address. (gethostbyaddr is no longer recommended, use getnameinfo instead)

gethostbyaddr

As of Windows Sockets 2, the gethostbyaddr function (winsock.h) is no longer recommended for use.

gethostbyaddr

The gethostbyaddr macro function (wsipv6ok.h) retrieves the host information corresponding to a network address.

gethostbyname

The gethostbyname function (winsock2.h) function retrieves host information corresponding to a host name from a host database.

gethostbyname

The gethostbyname function (winsock.h) retrieves host information corresponding to a host name from a host database.

gethostbyname

The gethostbyname macro function (wsipv6ok.h) retrieves host information corresponding to a host name from a host database.

gethostname

The gethostname function (winsock2.h) retrieves the standard host name for the local computer.

gethostname

The gethostname function (winsock.h) retrieves the standard host name for the local computer.

GetHostNameW

The GetHostNameW function retrieves the standard host name for the local computer as a Unicode string.

getipv4sourcefilter

Retrieves the multicast filter state for an IPv4 socket.

GetNameByTypeA

The GetNameByType function retrieves the name of a network service for the specified service type. (ANSI)

GetNameByTypeW

The GetNameByType function retrieves the name of a network service for the specified service type. (Unicode)

getnameinfo

Provides protocol-independent name resolution from an address to an ANSI host name and from a port number to the ANSI service name.

GetNameInfoW

Provides protocol-independent name resolution from an address to a Unicode host name and from a port number to the Unicode service name.

getpeername

The getpeername function (winsock2.h) retrieves the address of the peer to which a socket is connected.

getpeername

The getpeername function (winsock.h) retrieves the address of the peer to which a socket is connected.

getprotobyname

The getprotobyname function (winsock2.h) retrieves the protocol information corresponding to a protocol name.

getprotobyname

The getprotobyname function (winsock.h) retrieves the protocol information corresponding to a protocol name.

getprotobynumber

The getprotobynumber function (winsock2.h) retrieves protocol information corresponding to a protocol number.

getprotobynumber

The getprotobynumber function (winsock.h) retrieves protocol information corresponding to a protocol number.

getservbyname

The getservbyname function (winsock2.h) retrieves service information corresponding to a service name and protocol.

getservbyname

The getservbyname function (winsock.h) retrieves service information corresponding to a service name and protocol.

getservbyport

The getservbyport function (winsock2.h) retrieves service information corresponding to a port and protocol.

getservbyport

The getservbyport function (winsock.h) retrieves service information corresponding to a port and protocol.

GetServiceA

The GetService function retrieves information about a network service in the context of a set of default namespaces or a specified namespace. (ANSI)

GetServiceW

The GetService function retrieves information about a network service in the context of a set of default namespaces or a specified namespace. (Unicode)

getsockname

The getsockname function (winsock2.h) retrieves the local name for the specified socket descriptor.

getsockname

The getsockname function (winsock.h) retrieves the local name for a socket.

getsockopt

The getsockopt function retrieves the current value for a socket option associated with a socket of any type, in any state, and stores the result in optval.

getsockopt

The getsockopt function (winsock.h) retrieves a socket option.

getsourcefilter

Retrieves the multicast filter state for an IPv4 or IPv6 socket.

GetTypeByNameA

The GetTypeByName function retrieves a service type GUID for a network service specified by name. (ANSI)

GetTypeByNameW

The GetTypeByName function retrieves a service type GUID for a network service specified by name. (Unicode)

htond

Converts a double from host to TCP/IP network byte order (which is big-endian).

htonf

Converts a float from host to TCP/IP network byte order (which is big-endian).

htonl

The htonl function (winsock2.h) converts a u_long from host to TCP/IP network byte order, which is big-endian.

htonl

The htonl function (winsock.h) converts a u_long from host to TCP/IP network byte order (which is big-endian).

htonll

Converts an unsigned __int64 from host to TCP/IP network byte order (which is big-endian).

htons

The htons function (winsock2.h) converts a u_short from host to TCP/IP network byte order, which is big-endian.

htons

The htons function (winsock.h) converts a u_short from host to TCP/IP network byte order (which is big-endian).

inet_addr

The inet_addr function (winsock2.h) converts a string containing an IPv4 dotted-decimal address into a proper address for the IN_ADDR structure.

inet_addr

The inet_addr function (winsock.h) converts a string containing an IPv4 dotted-decimal address into a proper address for the IN_ADDR structure.

inet_addr

The inet_addr macro function (wsipv6ok.h) converts a string containing an IPv4 dotted-decimal address into a proper address for the IN_ADDR structure.

inet_ntoa

The inet_ntoa function (winsock2.h) converts an (Ipv4) Internet network address into an ASCII string in Internet standard dotted-decimal format.

inet_ntoa

The inet_ntoa function (winsock.h) converts an (Ipv4) Internet network address into an ASCII string in Internet standard dotted-decimal format.

inet_ntoa

The inet_ntoa macro function (wsipv6ok.h) converts an (Ipv4) Internet network address into an ASCII string in Internet standard dotted-decimal format.

inet_ntop

The InetNtop function converts an IPv4 or IPv6 Internet network address into a string in Internet standard format. The ANSI version of this function is inet_ntop. (InetNtopA or inet_ntop)

inet_pton

The InetPton function 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 inet_pton. (InetPtonA or inet_pton)

InetNtopW

The InetNtop function converts an IPv4 or IPv6 Internet network address into a string in Internet standard format. The ANSI version of this function is inet_ntop. (InetNtopW)

InetPtonW

The InetPton function 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 inet_pton. (InetPtonW)

ioctlsocket

The ioctlsocket function (winsock2.h) controls the I/O mode of a socket and can be used on any socket in any state.

ioctlsocket

The ioctlsocket function (winsock.h) controls the I/O mode of a socket.

listen

The listen function places a socket in a state in which it is listening for an incoming connection.

LPFN_CONNECTEX

The ConnectEx function establishes a connection to a specified socket, and optionally sends data once the connection is established.

LPFN_DISCONNECTEX

Closes a connection on a socket, and allows the socket handle to be reused.Note This function is a Microsoft-specific extension to the Windows Sockets specification.

LPFN_RIOCLOSECOMPLETIONQUEUE

Closes an existing completion queue used for I/O completion notification by send and receive requests with the Winsock registered I/O extensions.

LPFN_RIOCREATECOMPLETIONQUEUE

Creates an I/O completion queue of a specific size for use with the Winsock registered I/O extensions.

LPFN_RIOCREATEREQUESTQUEUE

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.

LPFN_RIODEQUEUECOMPLETION

Removes entries from an I/O completion queue for use with the Winsock registered I/O extensions.

LPFN_RIODEREGISTERBUFFER

Deregisters a registered buffer used with the Winsock registered I/O extensions.

LPFN_RIONOTIFY

Registers the method to use for notification behavior with an I/O completion queue for use with the Winsock registered I/O extensions.

LPFN_RIORECEIVE

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.

LPFN_RIORECEIVEEX

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.

LPFN_RIOREGISTERBUFFER

Registers a RIO_BUFFERID, a registered buffer descriptor, with a specified buffer for use with the Winsock registered I/O extensions.

LPFN_RIORESIZECOMPLETIONQUEUE

Resizes an I/O completion queue to be either larger or smaller for use with the Winsock registered I/O extensions.

LPFN_RIORESIZEREQUESTQUEUE

Resizes a request queue to be either larger or smaller for use with the Winsock registered I/O extensions.

LPFN_RIOSEND

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.

LPFN_RIOSENDEX

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.

LPFN_TRANSMITPACKETS

Transmits in-memory data or file data over a connected socket.

LPFN_WSARECVMSG

LPFN_WSARECVMSG is a function pointer type. You implement a matching **WSARecvMsg** callback function in your app. The system uses your callback function to transmit to you in-memory data, or file data, over a connected socket.

LPNSPCLEANUP

Terminates the use of a particular Windows Sockets namespace service provider.

LPNSPGETSERVICECLASSINFO

Retrieves all the pertinent class information (schema) pertaining to the namespace provider.

LPNSPINSTALLSERVICECLASS

The NSPInstallServiceClass function registers service class schema within the namespace providers.

LPNSPIOCTL

Sends an IOCTL to a namespace service provider.

LPNSPLOOKUPSERVICEBEGIN

Initiates a client query that is constrained by the information contained within a WSAQUERYSET structure.

LPNSPLOOKUPSERVICEEND

Called to free the handle after previous calls to NSPLookupServiceBegin and NSPLookupServiceNext.

LPNSPLOOKUPSERVICENEXT

Called after obtaining a handle from a previous call to NSPLookupServiceBegin in order to retrieve the requested service information.

LPNSPREMOVESERVICECLASS

Permanently removes a specified service class from the namespace.

LPNSPSETSERVICE

Registers or deregisters a service instance within a namespace.

LPNSPV2CLEANUP

Notifies a namespace service provider version-2 (NSPv2) provider that a client session has terminated.

LPNSPV2CLIENTSESSIONRUNDOWN

Notifies a namespace service provider version-2 (NSPv2) provider that the client session is terminating.

LPNSPV2LOOKUPSERVICEBEGIN

Initiates a client query of a namespace version-2 service provider that is constrained by the information contained within a WSAQUERYSET2 structure.

LPNSPV2LOOKUPSERVICEEND

Called to free the handle after previous calls to NSPv2LookupServiceBegin and NSPv2LookupServiceNextEx.

LPNSPV2LOOKUPSERVICENEXTEX

Called after obtaining a handle from a previous call to NSPv2LookupServiceBegin in order to retrieve the requested information from a namespace version-2 service provider.

LPNSPV2SETSERVICEEX

Registers or deregisters a name or service instance within a namespace of a namespace service provider version-2 (NSPv2) provider.

LPNSPV2STARTUP

Notifies a namespace service provider version-2 (NSPv2) provider that a new client process is to begin using the provider.

LPWSAOVERLAPPED_COMPLETION_ROUTINE

TBD (LPWSAOVERLAPPED_COMPLETION_ROUTINE)

LPWSPACCEPT

The LPWSPAccept function conditionally accepts a connection based on the return value of a condition function.

LPWSPADDRESSTOSTRING

The LPWSPAddressToString function converts all components of a sockaddr structure into a human-readable numeric string representation of the address. This is used mainly for display purposes.

LPWSPASYNCSELECT

The LPWSPAsyncSelect function requests Windows message-based event notification of network events for a socket.

LPWSPBIND

The LPWSPBind function associates a local address (that is, name) with a socket.

LPWSPCANCELBLOCKINGCALL

The LPWSPCancelBlockingCall function cancels a blocking call that is currently in progress.

LPWSPCLEANUP

The LPWSPCleanup function terminates use of the Windows Sockets service provider.

LPWSPCLOSESOCKET

The LPWSPCloseSocket function closes a socket.

LPWSPCONNECT

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

LPWSPDUPLICATESOCKET

The LPWSPDuplicateSocket function returns a WSAPROTOCOL_INFO structure that can be used to create a new socket descriptor for a shared socket.

LPWSPENUMNETWORKEVENTS

The LPWSPEnumNetworkEvents function reports occurrences of network events for the indicated socket.

LPWSPEVENTSELECT

The **LPWSPEventSelect** function specifies an event object to be associated with the supplied set of network events.

LPWSPGETOVERLAPPEDRESULT

The LPWSPGetOverlappedResult function returns the results of an overlapped operation on the specified socket.

LPWSPGETPEERNAME

The LPWSPGetPeerName function gets the address of the peer to which a socket is connected.

LPWSPGETQOSBYNAME

The LPWSPGetQOSByName function initializes a QOS structure based on a named template, or retrieves an enumeration of the available template names.

LPWSPGETSOCKNAME

The LPWSPGetSockName function gets the local name for a socket.

LPWSPGETSOCKOPT

The LPWSPGetSockOpt function retrieves a socket option.

LPWSPIOCTL

The LPWSPIoctl function controls the mode of a socket.

LPWSPJOINLEAF

The LPWSPJoinLeaf function joins a leaf node into a multipoint session, exchanges connect data, and specifies needed quality of service based on the supplied flow specifications.

LPWSPLISTEN

The LPWSPListen function establishes a socket to listen for incoming connections.

LPWSPRECV

The LPWSPRecv function receives data on a socket.

LPWSPRECVDISCONNECT

The LPWSPRecvDisconnect function terminates reception on a socket and retrieves the disconnect data, if the socket is connection oriented.

LPWSPRECVFROM

The LPWSPRecvFrom function receives a datagram and stores the source address.

LPWSPSELECT

The LPWSPSelect function determines the status of one or more sockets.

LPWSPSEND

The LPWSPSend function sends data on a connected socket.

LPWSPSENDDISCONNECT

The LPWSPSendDisconnect function initiates termination of the connection for the socket and sends disconnect data.

LPWSPSENDTO

The WSPSendTo function sends data to a specific destination using overlapped I/O.

LPWSPSETSOCKOPT

The LPWSPSetSockOpt function sets a socket option.

LPWSPSHUTDOWN

The LPWSPShutdown function disables sends and/or receives on a socket.

LPWSPSOCKET

The LPWSPSocket function creates a socket.

LPWSPSTRINGTOADDRESS

The WSPStringToAddress function converts a human-readable numeric string to a socket address structure (sockaddr) suitable to passing to Windows Sockets routines that take such a structure.

NSPStartup

Retrieves the dynamic information about a provider, such as the list of the DLL entry points.

ntohd

Converts an unsigned __int64 from TCP/IP network order to host byte order (which is little-endian on Intel processors) and returns a double.

ntohf

Converts an unsigned __int32 from TCP/IP network order to host byte order (which is little-endian on Intel processors) and returns a float.

ntohl

The ntohl function (winsock2.h) converts a u_long from TCP/IP network order to host byte order (which is little-endian on Intel processors).

ntohl

The ntohl function (winsock.h) converts a u_long from TCP/IP network order to host byte order, which is little-endian on Intel processors.

ntohll

Converts an unsigned __int64 from TCP/IP network order to host byte order (which is little-endian on Intel processors).

ntohs

The ntohs function (winsock2.h) converts a u_short from TCP/IP network byte order to host byte order (which is little-endian on Intel processors).

ntohs

The ntohs function (winsock.h) converts a u_short from TCP/IP network byte order to host byte order, which is little-endian on Intel processors.

ProcessSocketNotifications

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.

recv

The recv function (winsock2.h) receives data from a connected socket or a bound connectionless socket.

recv

The recv function (winsock.h) receives data from a connected socket or a bound connectionless socket.

recvfrom

The recvfrom function receives a datagram, and stores the source address.

recvfrom

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

select

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

send

Sends data on a connected socket. (send)

sendto

The sendto function (winsock2.h) is used to write outgoing data on a socket.

sendto

The sendto function (winsock.h) sends data to a specific destination.

SetAddrInfoExA

Registers or deregisters a name, a service name, and associated addresses with a specific namespace provider. (ANSI)

SetAddrInfoExW

Registers or deregisters a name, a service name, and associated addresses with a specific namespace provider. (Unicode)

setipv4sourcefilter

Sets the multicast filter state for an IPv4 socket.

SetServiceA

The SetService function registers or removes from the registry a network service within one or more namespaces. (ANSI)

SetServiceW

The SetService function registers or removes from the registry a network service within one or more namespaces. (Unicode)

SetSocketMediaStreamingMode

Indicates whether the network is to be used for transferring streaming media that requires quality of service.

setsockopt

The setsockopt function (winsock2.h) sets the current value for a socket option associated with a socket of any type, in any state.

setsockopt

The setsockopt function (winsock.h) sets a socket option.

setsourcefilter

Sets the multicast filter state for an IPv4 or IPv6 socket.

shutdown

The shutdown function (winsock2.h) is used on all types of sockets to disable reception, transmission, or both.

shutdown

The shutdown function (winsock.h) disables sends or receives on a socket.

socket

The socket function creates a socket that is bound to a specific transport service provider.

SocketNotificationRetrieveEvents

This inline helper function is provided as a convenience to retrieve the events mask from an OVERLAPPED_ENTRY.

TransmitFile

The TransmitFile function (mswsock.h) transmits file data over a connected socket handle.

TransmitFile

The TransmitFile function (winsock.h) transmits file data over a connected socket handle.

WPUCloseEvent

The WPUCloseEvent function closes an open event object handle.

WPUCloseSocketHandle

The WPUCloseSocketHandle function closes an existing socket handle.

WPUCloseThread

The WPUCloseThread function closes a thread opened with a call to WPUOpenCurrentThread.

WPUCompleteOverlappedRequest

The WPUCompleteOverlappedRequest function performs overlapped I/O completion notification for overlapped I/O operations.

WPUCreateEvent

The WPUCreateEvent function creates a new event object.

WPUCreateSocketHandle

The WPUCreateSocketHandle function creates a new socket handle.

WPUFDIsSet

The WPUFDIsSet function checks the membership of the specified socket handle.

WPUGetProviderPath

The WPUGetProviderPath function retrieves the DLL path for the specified provider.

WPUModifyIFSHandle

The WPUModifyIFSHandle function receives a (possibly) modified IFS handle from Ws2_32.dll.

WPUOpenCurrentThread

The WPUOpenCurrentThread function opens a handle to the current thread that can be used with overlapped functions in a layered service provider.

WPUPostMessage

The WPUPostMessage function performs the standard Windows PostMessage function in a way that maintains backward compatibility with older versions of WSOCK32.dll.

WPUQueryBlockingCallback

The WPUQueryBlockingCallback function returns a pointer to a callback function the service provider should invoke periodically while servicing blocking operations.

WPUQuerySocketHandleContext

The WPUQuerySocketHandleContext function queries the context value associated with the specified socket handle.

WPUQueueApc

The WPUQueueApc function queues a user mode asynchronous procedure call (APC) to the specified thread in order to facilitate invocation of overlapped I/O completion routines.

WPUResetEvent

The WPUResetEvent function resets the state of the specified event object to nonsignaled. This call is safe for use within interrupt context.

WPUSetEvent

The WPUSetEvent function sets the state of the specified event object to signaled. This call is safe for use within interrupt context.

WSAAccept

The WSAAccept function 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.

WSAAddressToStringA

Converts all components of a sockaddr structure into a human-readable string representation of the address. (ANSI)

WSAAddressToStringW

Converts all components of a sockaddr structure into a human-readable string representation of the address. (Unicode)

WSAAdvertiseProvider

Makes a specific namespace version-2 provider available for all eligible clients.

WSAAsyncGetHostByAddr

The WSAAsyncGetHostByAddr function (winsock2.h) asynchronously retrieves host information that corresponds to an address.

WSAAsyncGetHostByAddr

The WSAAsyncGetHostByAddr function (winsock.h) asynchronously retrieves host information that corresponds to an address.

WSAAsyncGetHostByAddr

The WSAAsyncGetHostByAddr macro function (wsipv6ok.h) asynchronously retrieves host information that corresponds to an address.

WSAAsyncGetHostByName

The WSAAsyncGetHostByName function (winsock2.h) asynchronously retrieves host information that corresponds to a host name.

WSAAsyncGetHostByName

The WSAAsyncGetHostByName function (winsock.h) asynchronously retrieves host information that corresponds to a host name.

WSAAsyncGetHostByName

The WSAAsyncGetHostByName macro function (wsipv6ok.h) asynchronously retrieves host information that corresponds to a host name.

WSAAsyncGetProtoByName

The WSAAsyncGetProtoByName function (winsock2.h) asynchronously retrieves protocol information that corresponds to a protocol name.

WSAAsyncGetProtoByName

The WSAAsyncGetProtoByName function (winsock.h) asynchronously retrieves protocol information that corresponds to a protocol name.

WSAAsyncGetProtoByNumber

The WSAAsyncGetProtoByNumber function (winsock2.h) asynchronously retrieves protocol information that corresponds to a protocol number.

WSAAsyncGetProtoByNumber

The WSAAsyncGetProtoByNumber function (winsock.h) asynchronously retrieves protocol information that corresponds to a protocol number.

WSAAsyncGetServByName

The WSAAsyncGetServByName function (winsock2.h) asynchronously retrieves service information that corresponds to a service name and port.

WSAAsyncGetServByName

The WSAAsyncGetServByName function (winsock.h) asynchronously retrieves service information that corresponds to a service name and port.

WSAAsyncGetServByPort

The WSAAsyncGetServByPort function (winsock2.h) asynchronously retrieves service information that corresponds to a port and protocol.

WSAAsyncGetServByPort

The WSAAsyncGetServByPort function (winsock.h) asynchronously retrieves service information that corresponds to a port and protocol.

WSAAsyncSelect

The WSAAsyncSelect function (winsock2.h) requests Windows message based notification of network events for a socket.

WSAAsyncSelect

The WSAAsyncSelect function (winsock.h) requests Windows message-based notification of network events for a socket.

WSACancelAsyncRequest

The WSACancelAsyncRequest function (winsock2.h) cancels an incomplete asynchronous operation.

WSACancelAsyncRequest

The WSACancelAsyncRequest function (winsock.h) cancels an incomplete asynchronous operation.

WSACancelBlockingCall

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

WSACleanup

The WSACleanup function (winsock2.h) terminates use of the WS2_32.dll.

WSACleanup

The WSACleanup function (winsock.h) terminates use of the WS2_32.dll.

WSACloseEvent

The WSACloseEvent function closes an open event object handle.

WSAConnect

The WSAConnect function establishes a connection to another socket application, exchanges connect data, and specifies required quality of service based on the specified FLOWSPEC structure.

WSAConnectByList

Establishes a connection to one out of a collection of possible endpoints represented by a set of destination addresses (host names and ports).

WSAConnectByNameA

Establishes a connection to a specified host and port. (ANSI)

WSAConnectByNameW

Establishes a connection to a specified host and port. (Unicode)

WSACreateEvent

The WSACreateEvent function creates a new event object.

WSADeleteSocketPeerTargetName

Removes the association between a peer target name and an IP address for a socket. After a successful return, there will be no future association between the IP address and the target name.

WSADuplicateSocketA

The WSADuplicateSocket function returns a WSAPROTOCOL_INFO structure that can be used to create a new socket descriptor for a shared socket. The WSADuplicateSocket function cannot be used on a QOS-enabled socket. (ANSI)

WSADuplicateSocketW

The WSADuplicateSocket function returns a WSAPROTOCOL_INFO structure that can be used to create a new socket descriptor for a shared socket. The WSADuplicateSocket function cannot be used on a QOS-enabled socket. (Unicode)

WSAEnumNameSpaceProvidersA

The WSAEnumNameSpaceProviders function retrieves information on available namespace providers. (ANSI)

WSAEnumNameSpaceProvidersExA

Retrieves information on available namespace providers. (ANSI)

WSAEnumNameSpaceProvidersExW

Retrieves information on available namespace providers. (Unicode)

WSAEnumNameSpaceProvidersW

The WSAEnumNameSpaceProviders function retrieves information on available namespace providers. (Unicode)

WSAEnumNetworkEvents

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

WSAEnumProtocolsA

The WSAEnumProtocols function retrieves information about available transport protocols. (ANSI)

WSAEnumProtocolsW

The WSAEnumProtocols function retrieves information about available transport protocols. (Unicode)

WSAEventSelect

The WSAEventSelect function specifies an event object to be associated with the specified set of FD_XXX network events.

WSAGetFailConnectOnIcmpError

Queries the state of the **TCP_FAIL_CONNECT_ON_ICMP_ERROR** socket option.

WSAGetIcmpErrorInfo

Retrieves information about an ICMP error received on a TCP socket during connection setup.

WSAGetIPUserMtu

Retrieves the user-defined IP layer MTU for a socket.

WSAGetLastError

The WSAGetLastError function (winsock2.h) returns the error status for the last Windows Sockets operation that failed.

WSAGetLastError

The WSAGetLastError function (winsock.h) returns the error status for the last Windows Sockets operation that failed.

WSAGetOverlappedResult

The WSAGetOverlappedResult function retrieves the results of an overlapped operation on the specified socket.

WSAGetQOSByName

The WSAGetQOSByName function initializes a QOS structure based on a named template, or it supplies a buffer to retrieve an enumeration of the available template names.

WSAGetServiceClassInfoA

The WSAGetServiceClassInfo function retrieves the class information (schema) pertaining to a specified service class from a specified namespace provider. (ANSI)

WSAGetServiceClassInfoW

The WSAGetServiceClassInfo function retrieves the class information (schema) pertaining to a specified service class from a specified namespace provider. (Unicode)

WSAGetServiceClassNameByClassIdA

The WSAGetServiceClassNameByClassId function retrieves the name of the service associated with the specified type. This name is the generic service name, like FTP or SNA, and not the name of a specific instance of that service. (ANSI)

WSAGetServiceClassNameByClassIdW

The WSAGetServiceClassNameByClassId function retrieves the name of the service associated with the specified type. This name is the generic service name, like FTP or SNA, and not the name of a specific instance of that service. (Unicode)

WSAGetUdpRecvMaxCoalescedSize

Retrieves the maximum size of a received, coalesced message for a UDP socket.

WSAGetUdpSendMessageSize

Retrieves the segmentation message size for a UDP socket.

WSAHtonl

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

WSAHtons

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

WSAImpersonateSocketPeer

Used to impersonate the security principal corresponding to a socket peer in order to perform application-level authorization.

WSAInstallServiceClassA

The WSAInstallServiceClass function registers a service class schema within a namespace. (ANSI)

WSAInstallServiceClassW

The WSAInstallServiceClass function registers a service class schema within a namespace. (Unicode)

WSAIoctl

The WSAIoctl function controls the mode of a socket.

WSAIsBlocking

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

WSAJoinLeaf

The WSAJoinLeaf function joins a leaf node into a multipoint session, exchanges connect data, and specifies needed quality of service based on the specified FLOWSPEC structures.

WSALookupServiceBeginA

The WSALookupServiceBegin function initiates a client query that is constrained by the information contained within a WSAQUERYSET structure. (ANSI)

WSALookupServiceBeginW

The WSALookupServiceBegin function initiates a client query that is constrained by the information contained within a WSAQUERYSET structure. (Unicode)

WSALookupServiceEnd

The WSALookupServiceEnd function is called to free the handle after previous calls to WSALookupServiceBegin and WSALookupServiceNext.

WSALookupServiceNextA

The WSALookupServiceNext function is called after obtaining a handle from a previous call to WSALookupServiceBegin in order to retrieve the requested service information. (ANSI)

WSALookupServiceNextW

The WSALookupServiceNext function is called after obtaining a handle from a previous call to WSALookupServiceBegin in order to retrieve the requested service information. (Unicode)

WSANSPIoctl

Enables developers to make I/O control calls to a registered namespace.

WSANtohl

The WSANtohl function converts a u_long from network byte order to host byte order.

WSANtohs

The WSANtohs function converts a u_short from network byte order to host byte order.

WSAPoll

The WSAPoll function determines status of one or more sockets.

WSAProviderCompleteAsyncCall

Notifies a client when an asynchronous call to a namespace version-2 provider is completed.

WSAProviderConfigChange

The WSAProviderConfigChange function notifies the application when the provider configuration is changed.

WSAQuerySocketSecurity

Queries information about the security applied to a connection on a socket.

WSARecv

Receives data from a connected socket or a bound connectionless socket. (WSARecv)

WSARecvDisconnect

The WSARecvDisconnect function terminates reception on a socket, and retrieves the disconnect data if the socket is connection oriented.

WSARecvEx

The WSARecvEx function (mswsock.h) receives data from a connected socket or a bound connectionless socket.

WSARecvEx

The WSARecvEx function (winsock.h) receives data from a connected socket or a bound connectionless socket.

WSARecvFrom

Receives a datagram and stores the source address.

WSARemoveServiceClass

The WSARemoveServiceClass function permanently removes the service class schema from the registry.

WSAResetEvent

The WSAResetEvent function resets the state of the specified event object to nonsignaled.

WSARevertImpersonation

Terminates the impersonation of a socket peer. This must be called after calling WSAImpersonateSocketPeer and finishing any access checks.

WSASend

Sends data on a connected socket. (WSASend)

WSASendDisconnect

The WSASendDisconnect function initiates termination of the connection for the socket and sends disconnect data.

WSASendMsg

Sends data and optional control information from connected and unconnected sockets. Note This function is a Microsoft-specific extension to the Windows Sockets specification. .

WSASendTo

Sends data to a specific destination, using overlapped I/O where applicable.

WSASetBlockingHook

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

WSASetEvent

The WSASetEvent function sets the state of the specified event object to signaled.

WSASetFailConnectOnIcmpError

Sets the state of the **TCP_FAIL_CONNECT_ON_ICMP_ERROR** socket option.

WSASetIPUserMtu

Sets the user-defined IP layer MTU on a socket.

WSASetLastError

The WSASetLastError function (winsock2.h) sets the error code that can be retrieved through the WSAGetLastError function.

WSASetLastError

The WSASetLastError function (winsock.h) sets the error code that can be retrieved through the WSAGetLastError function.

WSASetServiceA

The WSASetService function registers or removes from the registry a service instance within one or more namespaces. (ANSI)

WSASetServiceW

The WSASetService function registers or removes from the registry a service instance within one or more namespaces. (Unicode)

WSASetSocketPeerTargetName

Is 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.

WSASetSocketSecurity

Enables and applies security for a socket.

WSASetUdpRecvMaxCoalescedSize

Sets the maximum size of a coalesced message set on a UDP socket.

WSASetUdpSendMessageSize

Sets the segmentation message size on a UDP socket.

WSASocketA

The WSASocket function creates a socket that is bound to a specific transport-service provider. (ANSI)

WSASocketW

The WSASocket function creates a socket that is bound to a specific transport-service provider. (Unicode)

WSAStartup

The WSAStartup function (winsock2.h) initiates use of the Winsock DLL by a process.

WSAStartup

The WSAStartup function (winsock.h) initiates use of the Winsock DLL by a process.

WSAStringToAddressA

The WSAStringToAddress function converts a network address in its standard text presentation form into its numeric binary form in a sockaddr structure, suitable for passing to Windows Sockets routines that take such a structure. (ANSI)

WSAStringToAddressW

The WSAStringToAddress function converts a network address in its standard text presentation form into its numeric binary form in a sockaddr structure, suitable for passing to Windows Sockets routines that take such a structure. (Unicode)

WSAUnadvertiseProvider

Makes a specific namespace version-2 provider no longer available for clients.

WSAUnhookBlockingHook

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

WSAWaitForMultipleEvents

Returns when one or all of the specified event objects are in the signaled state, when the time-out interval expires, or when an I/O completion routine has executed.

WSCDeinstallProvider

Removes the specified transport provider from the system configuration database.

WSCDeinstallProvider32

Removes the specified 32-bit transport provider from the system configuration database.

WSCEnableNSProvider

Changes the state of a given namespace provider.

WSCEnableNSProvider32

Enables or disables a specified 32-bit namespace provider.

WSCEnumNameSpaceProviders32

Returns information on available 32-bit namespace providers.Note This call is a strictly 32-bit version of WSAEnumNameSpaceProviders for use on 64-bit platforms. It is provided to allow 64-bit processes to access the 32-bit catalogs. .

WSCEnumNameSpaceProvidersEx32

Retrieves information on available 32-bit namespace providers.

WSCEnumProtocols

The WSCEnumProtocols function retrieves information about available transport protocols.

WSCEnumProtocols32

Retrieves information about available transport protocols.Note This call is a strictly 32-bit version of WSCEnumProtocols for use on 64-bit platforms. It is provided to allow 64-bit processes to access the 32-bit catalogs. .

WSCGetApplicationCategory

Retrieves the layered service provider (LSP) categories associated with an application.

WSCGetProviderInfo

Retrieves the data associated with an information class for a layered service provider (LSP).

WSCGetProviderInfo32

Retrieves the data associated with an information class for a 32-bit layered service provider (LSP).Note This call is a strictly 32-bit version of WSCGetProviderInfo for use on 64-bit platforms.

WSCGetProviderPath

The WSCGetProviderPath function retrieves the DLL path for the specified provider.

WSCGetProviderPath32

Retrieves the DLL path for the specified 32-bit provider.Note This call is a strictly 32-bit version of WSCGetProviderPath for use on 64-bit platforms. It is provided to allow 64-bit processes to access the 32-bit catalogs. .

WSCInstallNameSpace

Installs a namespace provider. (WSCInstallNameSpace)

WSCInstallNameSpace32

Installs a specified 32-bit namespace provider. (WSCInstallNamespace32)

WSCInstallNameSpaceEx

Installs a namespace provider. (WSCInstallNameSpaceEx)

WSCInstallNameSpaceEx32

Installs a specified 32-bit namespace provider. (WSCInstallNameSpaceEx32)

WSCInstallProvider

Installs the specified transport provider into the system configuration database.

WSCInstallProvider64_32

Installs the specified transport service provider into the 32-bit and 64-bit system configuration databases on a 64-bit computer.

WSCInstallProviderAndChains

Installs the specified 32-bit transport provider as well as its specific protocol chains into the Winsock 2 system configuration database on a 32-bit computer.

WSCInstallProviderAndChains64_32

Installs the specified transport provider and its specific protocol chains into both the 32-bit and 64-bit Winsock 2 system configuration databases on a 64-bit computer.

WSCInstallQOSTemplate

Installs the specified QoS template in the system configuration database.

WSCRemoveQOSTemplate

Removes the specified QoS template from the system configuration database.

WSCSetApplicationCategory

Sets the permitted layered service provider (LSP) categories associated with an application.

WSCSetProviderInfo

Sets the data value for the specified information class for a layered service provider (LSP).

WSCSetProviderInfo32

Sets the data value for specified information class for a layered service provider (LSP).

WSCUnInstallNameSpace

Uninstalls the indicated name-space provider.

WSCUnInstallNameSpace32

Uninstalls a specific 32-bit namespace provider.

WSCUpdateProvider

Modifies the specified transport provider in the system configuration database.

WSCUpdateProvider32

Modifies the specified 32-bit transport provider in the system configuration database.Note This call is a strictly 32-bit version of WSCUpdateProvider for use on 64-bit platforms. It is provided to allow 64-bit processes to access the 32-bit catalogs. .

WSCWriteNameSpaceOrder

Changes the order of available Windows Sockets (Winsock) 2 namespace providers. The order of the namespace providers determines the priority of the namespace when enumerated or queried for name resolution.

WSCWriteNameSpaceOrder32

Changes the order of available Windows Sockets (Winsock) 2 namespace providers in a 32-bit catalog.

WSCWriteProviderOrder

Used to reorder the available transport providers.

WSCWriteProviderOrder32

Used to reorder the available 32-bit transport providers.

WSPStartup

The WSPStartup function initiates use of a Windows Sockets service provider interface (SPI) by a client.

Structures

ADDRINFO_DNS_SERVER

Represents a custom Domain Name System (DNS) server, used in the Winsock APIs.

ADDRINFOA

Used by the getaddrinfo function to hold host address information.

ADDRINFOEX2A

Used by the GetAddrInfoEx function to hold host address information when both a canonical name and a fully qualified domain name have been requested. (ANSI)

ADDRINFOEX2W

Used by the GetAddrInfoEx function to hold host address information when both a canonical name and a fully qualified domain name have been requested. (Unicode)

ADDRINFOEX3

Used by the GetAddrInfoEx function to hold host address information when a specific network interface has been requested. (addrinfoex3W)

ADDRINFOEX4

Used by the GetAddrInfoEx function to hold host address information when a specific network interface has been requested. (addrinfoex4W)

ADDRINFOEX5

Used by the GetAddrInfoExW function to hold host address information when a specific network interface has been requested. (ADDRINFOEX5)

ADDRINFOEX6

Used by the GetAddrInfoExW function to hold host address information when a specific network interface has been requested. (ADDRINFOEX6)

ADDRINFOEXA

Used by the GetAddrInfoEx function to hold host address information. (ANSI)

ADDRINFOEXW

Used by the GetAddrInfoEx function to hold host address information. (Unicode)

ADDRINFOW

Used by the GetAddrInfoW function to hold host address information.

AFPROTOCOLS

The AFPROTOCOLS structure supplies a list of protocols to which application programmers can constrain queries. The AFPROTOCOLS structure is used for query purposes only.

ASSOCIATE_NAMERES_CONTEXT_INPUT

Contains the transport setting ID and handle to a fully qualified domain name.

ATM_ADDRESS

The ATM_ADDRESS structure holds ATM address data for ATM-based sockets.

ATM_BHLI

The ATM_BHLI structure is used to identify B-HLI information for an associated ATM socket.

ATM_BLLI

The ATM_BLLI structure is used to identify B-LLI information for an associated ATM socket.

BLOB

The BLOB (wtypesbase.h) structure, derived from Binary Large Object, contains information about a block of data.

BLOB

The BLOB structure (winsock2.h), derived from Binary Large Object, contains information about a block of data.

BLOB

The BLOB structure (nspapi.h), which is derived from Binary Large Object, contains information about a block of data.

CSADDR_INFO

The CSADDR_INFO (ws2def.h) structure contains Windows Sockets address information for a socket, network service, or namespace provider.

CSADDR_INFO

The CSADDR_INFO structure (nspapi.h) contains Windows Sockets address information for a socket, network service, or namespace provider.

fd_set

The Fd_set structure (winsock2.h) is used by Windows Sockets (Winsock) functions and service providers to place sockets into a set.

fd_set

The fd_set structure (winsock.h) is used by Windows Sockets (Winsock) functions and service providers to place sockets into a set.

GROUP_FILTER

Provides multicast filtering parameters for multicast IPv6 or IPv4 addresses.

GROUP_REQ

Provides multicast group information for IPv6 or IPv4 addresses.

GROUP_SOURCE_REQ

Provides multicast group information for IPv6 or IPv4 addresses that includes the source IP address.

HOSTENT

The HOSTENT structure (winsock2.h) is used by functions to store information about a given host, such as host name, IPv4 address, and so forth.

HOSTENT

The HOSTENT (winsock.h) structure is used by functions to store information about a given host, such as host name, IPv4 address, and so forth.

ICMP_ERROR_INFO

Used to store received ICMP error information.

in_addr

The in_addr structure represents an IPv4 Internet address.

IN_PKTINFO

The in_pktinfo structure is used to store received packet address information, and is used by Windows to return information about received packets and also allows specifying the local IPv4 address to use for sending packets.

IN6_PKTINFO

The in6_pktinfo structure is used to store received IPv6 packet address information, and is used by Windows to return information about received packets and also allows specifying the local IPv6 address to use for sending packets.

INET_PORT_RANGE

Provides input data used by the SIO_ACQUIRE_PORT_RESERVATION IOCTL to acquire a runtime reservation for a block of TCP or UDP ports.

INET_PORT_RESERVATION_INSTANCE

Contains a port reservation and a token for a block of TCP or UDP ports.

INET_PORT_RESERVATION_TOKEN

Contains a port reservation token for a block of TCP or UDP ports.

INTERFACE_INFO

The INTERFACE_INFO structure is used in conjunction with the SIO_GET_INTERFACE_LIST ioctl command to obtain information about an interface IP address.

INTERFACE_INFO_EX

The INTERFACE_INFO_EX structure is used in conjunction with the SIO_GET_INTERFACE_LIST IOCTL command to obtain information about an interface IP address.

IP_MREQ

The ip_mreq structure provides multicast group information for IPv4 addresses.

IP_MREQ_SOURCE

The ip_mreq_source structure provides multicast group information for IPv4 addresses.

IP_MSFILTER

The ip_msfilter structure provides multicast filtering parameters for IPv4 addresses.

IPV6_MREQ

The ipv6_mreq structure provides multicast group information for IPv6 addresses.

IPX_ADDRESS_DATA

The IPX_ADDRESS_DATA structure provides information about a specific adapter to which IPX is bound. Used in conjunction with getsockopt function calls that specify IPX_ADDRESS in the optname parameter.

IPX_NETNUM_DATA

The IPX_NETNUM_DATA structure provides information about a specified IPX network number. Used in conjunction with getsockopt function calls that specify IPX_GETNETINFO in the optname parameter.

IPX_SPXCONNSTATUS_DATA

The IPX_SPXCONNSTATUS_DATA structure provides information about a connected SPX socket.

LINGER

The LINGER structure (winsock2.h) maintains information about a specific socket that specifies how that socket should behave when data is queued.

LINGER

The LINGER (winsock.h) structure maintains information about a specific socket that specifies how that socket should behave when data is queued to be sent.

NAPI_DOMAIN_DESCRIPTION_BLOB

Describes a domain handled by a namespace provider for the NS_EMAIL namespace.

NAPI_PROVIDER_INSTALLATION_BLOB

Contains the information required to install a namespace provider for the NS_EMAIL namespace.

NS_SERVICE_INFOA

Contains information about a network service or a network service type in the context of a specified namespace, or a set of default namespaces. (ANSI)

NS_SERVICE_INFOW

Contains information about a network service or a network service type in the context of a specified namespace, or a set of default namespaces. (Unicode)

NSP_ROUTINE

Contains information regarding the functions implemented by a namespace service provider version 1 (NSPv1) provider.

NSPV2_ROUTINE

Contains information on the functions implemented by a namespace service provider version-2 (NSPv2) provider.

PROTOCOL_INFOA

Contains information about a protocol. (ANSI)

PROTOCOL_INFOW

Contains information about a protocol. (Unicode)

PROTOENT

The PROTOENT structure (winsock2.h) contains the name and protocol numbers that correspond to a given protocol name.

PROTOENT

The PROTOENT (winsock.h) structure contains the name and protocol numbers that correspond to a given protocol name.

REAL_TIME_NOTIFICATION_SETTING_INPUT

Provides input settings to apply for the REAL_TIME_NOTIFICATION_CAPABILITY transport setting for a TCP socket that is used with ControlChannelTrigger to receive background network notifications in a Windows Store app.

REAL_TIME_NOTIFICATION_SETTING_OUTPUT

Provides the output settings from a query for the REAL_TIME_NOTIFICATION_CAPABILITY transport setting for a TCP socket that is used with ControlChannelTrigger to receive background network notifications in a Windows Store app.

RIO_BUF

Specifies a portion of a registered buffer used for sending or receiving network data with the Winsock registered I/O extensions.

RIO_EXTENSION_FUNCTION_TABLE

Contains information on the functions that implement the Winsock registered I/O extensions.

RIO_NOTIFICATION_COMPLETION

Specifies the method for I/O completion to be used with a RIONotify function for sending or receiving network data with the Winsock registered I/O extensions.

RIORESULT

Contains data used to indicate request completion results used with the Winsock registered I/O extensions.

RM_FEC_INFO

The RM_FEC_INFO structure specifies settings for using forward error correction (FEC) with Reliable Multicast. This structure is used with the RM_USE_FEC socket option.

RM_RECEIVER_STATS

Provides statistical information for a Reliable Multicast receiver session. This structure is used with the RM_RECEIVER_STATISTICS socket option.

RM_SEND_WINDOW

The RM_SEND_WINDOW structure specifies the Reliable Multicast send window. This structure is used with the RM_RATE_WINDOW_SIZE socket option.

RM_SENDER_STATS

Provides statistical information for a Reliable Multicast sender session. This structure is used with the RM_SENDER_STATISTICS socket option.

SERVENT

The SERVENT structure (winsock2.h) is used to store or return the name and service number for a given service name.

SERVENT

The SERVENT (winsock.h) structure is used to store or return the name and service number for a given service name.

SERVICE_ADDRESS

Contains address information for a service. The structure can accommodate many types of interprocess communications (IPC) mechanisms and their address forms, including remote procedure calls (RPC), named pipes, and sockets.

SERVICE_ADDRESSES

The SERVICE_ADDRESSES structure contains an array of SERVICE_ADDRESS data structures.

SERVICE_INFOA

Contains information about a network service or a network service type. (ANSI)

SERVICE_INFOW

Contains information about a network service or a network service type. (Unicode)

SERVICE_TYPE_INFO_ABSA

The SERVICE_TYPE_INFO_ABS structure contains information about a network service type. Use SERVICE_TYPE_INFO_ABS to add a network service type to a namespace. (ANSI)

SERVICE_TYPE_INFO_ABSW

The SERVICE_TYPE_INFO_ABS structure contains information about a network service type. Use SERVICE_TYPE_INFO_ABS to add a network service type to a namespace. (Unicode)

SERVICE_TYPE_VALUE_ABSA

Contains information about a network-service type value. This information may be specific to a namespace. (ANSI)

SERVICE_TYPE_VALUE_ABSW

Contains information about a network-service type value. This information may be specific to a namespace. (Unicode)

SOCK_NOTIFY_REGISTRATION

Represents info supplied to the ProcessSocketNotifications function.

SOCKADDR

The sockaddr structure varies depending on the protocol selected. (sockaddr)

sockaddr_atm

The Windows Sockets sockaddr_atm structure stores socket address information for ATM sockets.

sockaddr_gen

Provides generic socket address information, and is used with the INTERFACE_INFO structure.

SOCKADDR_IN

The SOCKADDR_IN (winsock.h) structure varies depending on the protocol selected.

sockaddr_in6_old

The sockaddr_in6_old (ws2ipdef.h) structure varies depending on the protocol selected.

SOCKADDR_IRDA

The SOCKADDR_IRDA structure is used in conjunction with IrDA socket operations, defined by address family AF_IRDA.

SOCKET_ADDRESS

SOCKET_ADDRESS structure stores protocol-specific address information.

SOCKET_PEER_TARGET_NAME

Contains the IP address and name for a peer target and the type of security protocol to be used on a socket.

SOCKET_PROCESSOR_AFFINITY

Contains the association between a socket and an RSS processor core and NUMA node.

SOCKET_SECURITY_QUERY_INFO

Contains security information returned by the WSAQuerySocketSecurity function.

SOCKET_SECURITY_QUERY_TEMPLATE

Contains the security template used by the WSAQuerySocketSecurity function.

SOCKET_SECURITY_SETTINGS

Specifies generic security requirements for a socket.

SOCKET_SECURITY_SETTINGS_IPSEC

Specifies various security requirements and settings that are specific to IPsec.

TCP_INFO_v0

Contains the Transmission Control Protocol (TCP) statistics that were collected for a socket.

TCP_INFO_v1

Contains the Transmission Control Protocol (TCP) statistics that were collected for a socket. (version 1.0)

TCP_INITIAL_RTO_PARAMETERS

Specifies data used by the SIO_TCP_INITIAL_RTO IOCTL to configure initial re-transmission timeout (RTO) parameters to be used on the socket.

TIMESTAMPING_CONFIG

Describes the input structure used by the SIO_TIMESTAMPING IOCTL to configure timestamp reception for a datagram socket.

TIMEVAL

The TIMEVAL structure (winsock2.h) is used to specify a time interval. It is associated with the Berkeley Software Distribution (BSD) Time.h header file.

TIMEVAL

The TIMEVAL (winsock.h) structure is used to specify a time interval. It is associated with the Berkeley Software Distribution (BSD) Time.h header file.

TRANSMIT_FILE_BUFFERS

The TRANSMIT_FILE_BUFFERS structure (mswsock.h) specifies data to be transmitted before and after file data during a TransmitFile function file transfer operation.

TRANSMIT_FILE_BUFFERS

The TRANSMIT_FILE_BUFFERS (winsock.h) structure specifies data to be transmitted before and after file data during a TransmitFile function file transfer operation.

TRANSMIT_PACKETS_ELEMENT

Specifies a single data element to be transmitted by the TransmitPackets function.

TRANSPORT_SETTING_ID

The TRANSPORT_SETTING_ID structure (mstcpip.h) specifies the transport setting ID used by specific IOCTLs to apply or query the transport setting for a socket.

TRANSPORT_SETTING_ID

The TRANSPORT_SETTING_ID (transportsettingcommon.h) structure specifies the transport setting ID used by the SIO_APPLY_TRANSPORT_SETTING and SIO_QUERY_TRANSPORT_SETTING IOCTLs.

WSABUF

The WSABUF structure enables the creation or manipulation of a data buffer used by some Winsock functions.

WSACOMPLETION

Specifies completion notification settings for I/O control calls made to a registered namespace.

WSADATA

The WSADATA structure (winsock2.h) contains information about the Windows Sockets implementation.

WSADATA

The WSADATA (winsock.h) structure contains information about the Windows Sockets implementation.

WSAMSG

Used with the WSARecvMsg and WSASendMsg functions to store address and optional control information about connected and unconnected sockets as well as an array of buffers used to store message data.

WSANAMESPACE_INFOA

Contains all registration information for a namespace provider. (WSANAMESPACE_INFOA)

WSANAMESPACE_INFOEXA

Contains all registration information for a namespace provider. (WSANAMESPACE_INFOEXA)

WSANAMESPACE_INFOEXW

Contains all registration information for a namespace provider. (WSANAMESPACE_INFOEXW)

WSANAMESPACE_INFOW

Contains all registration information for a namespace provider. (WSANAMESPACE_INFOW)

WSANETWORKEVENTS

The WSANETWORKEVENTS structure is used to store a socket's internal information about network events.

WSANSCLASSINFOA

The WSANSCLASSINFO structure provides individual parameter information for a specific Windows Sockets namespace. (ANSI)

WSANSCLASSINFOW

The WSANSCLASSINFO structure provides individual parameter information for a specific Windows Sockets namespace. (Unicode)

WSAOVERLAPPED

Provides a communication medium between the initiation of an overlapped I/O operation and its subsequent completion.

WSAPOLLFD

Stores socket information used by the WSAPoll function.

WSAPROTOCOL_INFOA

Used to store or retrieve complete information for a given protocol. (ANSI)

WSAPROTOCOL_INFOW

Used to store or retrieve complete information for a given protocol. (Unicode)

WSAPROTOCOLCHAIN

The WSAPROTOCOLCHAIN structure contains a counted list of Catalog Entry identifiers that comprise a protocol chain.

WSAQUERYSET2A

Provides relevant information about a given service, including service class ID, service name , applicable namespace identifier and protocol information, as well as a set of transport addresses at which the service listens. (ANSI)

WSAQUERYSET2W

Provides relevant information about a given service, including service class ID, service name , applicable namespace identifier and protocol information, as well as a set of transport addresses at which the service listens. (Unicode)

WSAQUERYSETA

Provides relevant information about a given service, including service class ID, service name, applicable namespace identifier and protocol information, as well as a set of transport addresses at which the service listens. (ANSI)

WSAQUERYSETW

Provides relevant information about a given service, including service class ID, service name, applicable namespace identifier and protocol information, as well as a set of transport addresses at which the service listens. (Unicode)

WSASERVICECLASSINFOA

The WSASERVICECLASSINFO structure contains information about a specified service class. For each service class in Windows Sockets 2, there is a single WSASERVICECLASSINFO structure. (ANSI)

WSASERVICECLASSINFOW

The WSASERVICECLASSINFO structure contains information about a specified service class. For each service class in Windows Sockets 2, there is a single WSASERVICECLASSINFO structure. (Unicode)

WSATHREADID

The WSATHREADID structure enables a provider to identify a thread on which asynchronous procedure calls (APCs) can be queued using the WPUQueueApc function.

WSAVERSION

The WSAVERSION structure provides version comparison in Windows Sockets.

WSC_PROVIDER_AUDIT_INFO

Contains audit information for a layered service provider (LSP) entry in Windows Sockets 2.

WSPDATA

The WSPDATA structure contains service provider information.

WSPPROC_TABLE

Contains a table of pointers to service provider functions.

WSPUPCALLTABLE

Contains a table of pointers to service provider upcall functions.

af_irda.h header

10/22/2022 • 2 minutes to read • Edit Online

This header is used by Windows Sockets 2. For more information, see:

- Windows Sockets 2

af_irda.h contains the following programming interfaces:

Structures

SOCKADDR_IRDA The SOCKADDR_IRDA structure is used in conjunction with IrDA socket operations, defined by address family AF_IRDA.

SOCKADDR_IRDA structure (af_irda.h)

10/22/2022 • 2 minutes to read • Edit Online

The **SOCKADDR_IRDA** structure is used in conjunction with IrDA socket operations, defined by address family AF_IRDA.

Syntax

```
typedef struct _SOCKADDR_IRDA {
  u_short irdaAddressFamily;
  u_char  irdaDeviceID[4];
  char    irdaServiceName[25];
} SOCKADDR_IRDA, *PSOCKADDR_IRDA, *LPSOCKADDR_IRDA;
```

Members

`irdaAddressFamily`

Address family. This member is always AF_IRDA.

`irdaDeviceID[4]`

Device identifier (ID) of the IrDA device to which the client wants to issue the connect function call. Ignored by server applications.

`irdaServiceName[25]`

Well-known service name associated with a server application. Specified by servers during their bind function call.

Remarks

Client applications make use of each field in the **SOCKADDR_IRDA** structure. The **irdaDeviceID** member is obtained by a previous discovery operation performed by making a getsockopt(IRLMP_ENUMDEVICES) function call. For more information on performing a discovery operation, see the Notes for IrDA Sockets section in the Remarks section of **getsockopt**.

The **irdaServiceName** member is filled with the well-known value that the server application specified in its bind function call.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	af_irda.h

See also

bind

connect

getsockopt

setsockopt

in6addr.h header

10/22/2022 • 2 minutes to read • Edit Online

This header is used by multiple technologies. For more information, see:

- Network Drivers Prior to Windows Vista
- Network Drivers, Windows Vista and Later
- Windows Sockets 2

in6addr.h contains the following programming interfaces:

Structures

IN6_ADDR	The IN6_ADDR structure specifies an IPv6 transport address.

inaddr.h header

10/22/2022 • 2 minutes to read • Edit Online

This header is used by multiple technologies. For more information, see:

- IP Helper
- Network Drivers, Windows Vista and Later
- Windows Sockets 2

inaddr.h contains the following programming interfaces:

Structures

IN_ADDR	The in_addr structure represents an IPv4 address.

mstcpip.h header

10/22/2022 • 2 minutes to read • Edit Online

This header is used by Windows Sockets 2. For more information, see:

- Windows Sockets 2

mstcpip.h contains the following programming interfaces:

Structures

ASSOCIATE_NAMERES_CONTEXT_INPUT Contains the transport setting ID and handle to a fully qualified domain name.
INET_PORT_RANGE Provides input data used by the SIO_ACQUIRE_PORT_RESERVATION IOCTL to acquire a runtime reservation for a block of TCP or UDP ports.
INET_PORT_RESERVATION_INSTANCE Contains a port reservation and a token for a block of TCP or UDP ports.
INET_PORT_RESERVATION_TOKEN Contains a port reservation token for a block of TCP or UDP ports.
REAL_TIME_NOTIFICATION_SETTING_INPUT Provides input settings to apply for the REAL_TIME_NOTIFICATION_CAPABILITY transport setting for a TCP socket that is used with ControlChannelTrigger to receive background network notifications in a Windows Store app.
REAL_TIME_NOTIFICATION_SETTING_OUTPUT Provides the output settings from a query for the REAL_TIME_NOTIFICATION_CAPABILITY transport setting for a TCP socket that is used with ControlChannelTrigger to receive background network notifications in a Windows Store app.
SOCKET_PEER_TARGET_NAME Contains the IP address and name for a peer target and the type of security protocol to be used on a socket.
SOCKET_SECURITY_QUERY_INFO Contains security information returned by the WSAQuerySocketSecurity function.
SOCKET_SECURITY_QUERY_TEMPLATE Contains the security template used by the WSAQuerySocketSecurity function.

SOCKET_SECURITY_SETTINGS

Specifies generic security requirements for a socket.

SOCKET_SECURITY_SETTINGS_IPSEC

Specifies various security requirements and settings that are specific to IPsec.

TCP_INFO_v0

Contains the Transmission Control Protocol (TCP) statistics that were collected for a socket.

TCP_INFO_v1

Contains the Transmission Control Protocol (TCP) statistics that were collected for a socket. (version 1.0)

TCP_INITIAL_RTO_PARAMETERS

Specifies data used by the SIO_TCP_INITIAL_RTO IOCTL to configure initial re-transmission timeout (RTO) parameters to be used on the socket.

TIMESTAMPING_CONFIG

Describes the input structure used by the SIO_TIMESTAMPING IOCTL to configure timestamp reception for a datagram socket.

TRANSPORT_SETTING_ID

The TRANSPORT_SETTING_ID structure (mstcpip.h) specifies the transport setting ID used by specific IOCTLs to apply or query the transport setting for a socket.

Enumerations

CONTROL_CHANNEL_TRIGGER_STATUS

Specifies the status from a query for the REAL_TIME_NOTIFICATION_CAPABILITY transport setting for a TCP socket that is used with ControlChannelTrigger to receive background network notifications in a Windows Store app.

SOCKET_SECURITY_PROTOCOL

Indicates the type of security protocol to be used on a socket to secure network traffic.

SOCKET_USAGE_TYPE

Used to specified the usage type for the socket.

TCPSTATE

Indicates the possible states of a Transmission Control Protocol (TCP) connection.

ASSOCIATE_NAMERES_CONTEXT_INPUT structure (mstcpip.h)

10/22/2022 • 2 minutes to read • Edit Online

The **ASSOCIATE_NAMERES_CONTEXT_INPUT** structure contains the transport setting ID and handle to a fully qualified domain name.

Syntax

```
typedef struct _ASSOCIATE_NAMERES_CONTEXT_INPUT {
  TRANSPORT_SETTING_ID TransportSettingId;
  UINT64               Handle;
} ASSOCIATE_NAMERES_CONTEXT_INPUT, *PASSOCIATE_NAMERES_CONTEXT_INPUT;
```

Members

`TransportSettingId`

The transport setting ID.

`Handle`

Handle to a fully qualified domain name.

Remarks

Generally speaking, you can use **ASSOCIATE_NAMERES_CONTEXT_INPUT** to enforce policy based on Fully Qualified Domain Name (FQDN), rather than just IP address. you can do so by retrieving a handle to a FQDN with a call to GetAddrInfoEx, using the addinfoex4 structure. From there, you can use the handle in **ASSOCIATE_NAMERES_CONTEXT_INPUT** in a call to WSAIoctl, using the **SIO_APPLY_TRANSPORT_SETTING** ioctl.

Examples

The following code describes making a call to GetAddrInfoEx with a addinfoex4 structure to retrieve the handle to a FQDN. the sample then call WSAIoctl with the **ASSOCIATE_NAMERES_CONTEXT_INPUT** structure.

```
//
// Connect to a server using its IPv4 addresses
//

VOID
ConnectServer(
    PCWSTR server)
{
    int iResult;
    PADDRINFOEX4 pResult = NULL;
    ADDRINFOEX3 hints = { 0 };
    PADDRINFOEX4 pCur = NULL;
    WSADATA wsaData;
    SOCKET connectSocket = INVALID_SOCKET;
    ULONG bytesReturned = 0;
    ASSOCIATE_NAMERES_CONTEXT_INPUT input = { 0 };
    SOCKADDR_IN clientService;
    wchar_t instringbuffer[46];
```

```
    wchar_t ipstringbuffer[46];
    String string;
    DWORD dwRetval;
    //
    //  Initialize Winsock
    //
    iResult = WSAStartup(
        MAKEWORD(2, 2),
        &wsaData);
    if (iResult != 0) {
        printf("WSAStartup failed: %d\n", iResult);
        goto Exit;
    }

    //
    // Create a SOCKET for connection
    //
    connectSocket = socket(
        AF_UNSPEC,
        SOCK_STREAM,
        IPPROTO_TCP);
    if (connectSocket == INVALID_SOCKET)
    {
        printf("socket failed: %d\n", WSAGetLastError());
        goto Exit;
    }

    //
    // Do name resolution
    //

    hints.ai_family = AF_INET;
    hints.ai_socktype = SOCK_STREAM;
    hints.ai_flags = AI_EXTENDED | AI_FQDN | AI_CANONNAME | AI_RESOLUTION_HANDLE;
    hints.ai_version = ADDRINFOEX_VERSION_4;

    dwRetval = GetAddrInfoExW(
        server,
        NULL,
        NS_DNS,
        NULL,
        (const ADDRINFOEXW*)&hints,
        (PADDRINFOEXW*)&pResult,
        NULL,
        NULL,
        NULL, NULL);
    if (dwRetval != 0) {
        printf("GetAddrInfoEx failed with error: %d\n", dwRetval);
        goto Exit;
    }
    input.TransportSettingId.Guid = ASSOCIATE_NAMERES_CONTEXT;
    input.Handle = pResult->ai_resolutionhandle;

    //
    // Associate socket with the handle
    //

    if (WSAIoctl(
            connectSocket,
            SIO_APPLY_TRANSPORT_SETTING,
            (VOID *)&input,
            sizeof(input),
            NULL,
            0,
            &bytesReturned,
            NULL,
            NULL) == SOCKET_ERROR)
    if (iResult != 0){
        printf("WSAIoctl failed: %d\n", WSAGetLastError());
        goto Exit;
```

```
        goto Exit;
    }

    //
    // Connect to server
    //

    pCur = pResult;
    while (pCur != NULL)
    {
        if (pCur->ai_addr->sa_family == AF_INET)
        {
            clientService = *(const sockaddr_in*)pCur->ai_addr;
            clientService.sin_port = htons(80);
            if (connect(
                connectSocket,
                (const SOCKADDR *)&clientService,
                sizeof(clientService)) == SOCKET_ERROR)
            {
                printf("connect failed: %d\n", WSAGetLastError());
                goto Exit;
            }
        }
        pCur = pCur->ai_next;
    }

Exit:

    if (connectSocket != INVALID_SOCKET)
    {
        closesocket(connectSocket);
    }
    if (pResult)
    {
        FreeAddrInfoExW((ADDRINFOEXW*)pResult);
    }
    WSACleanup();
    return;
}
```

Requirements

Minimum supported client	Windows 10 [desktop apps only]
Minimum supported server	Windows Server 2016 [desktop apps only]
Header	mstcpip.h

See also

GetAddrInfoEx

WSAIoctl

addrinfoex4

CONTROL_CHANNEL_TRIGGER_STATUS enumeration (mstcpip.h)

10/22/2022 • 2 minutes to read • Edit Online

The CONTROL_CHANNEL_TRIGGER_STATUS enumeration specifies the status from a query for the **REAL_TIME_NOTIFICATION_CAPABILITY** transport setting for a TCP socket that is used with ControlChannelTrigger to receive background network notifications in a Windows Store app.

Syntax

```
typedef enum  {
  CONTROL_CHANNEL_TRIGGER_STATUS_INVALID = 0,
  CONTROL_CHANNEL_TRIGGER_STATUS_SOFTWARE_SLOT_ALLOCATED = 1,
  CONTROL_CHANNEL_TRIGGER_STATUS_HARDWARE_SLOT_ALLOCATED = 2,
  CONTROL_CHANNEL_TRIGGER_STATUS_POLICY_ERROR = 3,
  CONTROL_CHANNEL_TRIGGER_STATUS_SYSTEM_ERROR = 4,
  CONTROL_CHANNEL_TRIGGER_STATUS_TRANSPORT_DISCONNECTED = 5,
  CONTROL_CHANNEL_TRIGGER_STATUS_SERVICE_UNAVAILABLE = 6
} CONTROL_CHANNEL_TRIGGER_STATUS, *PCONTROL_CHANNEL_TRIGGER_STATUS;
```

Constants

`CONTROL_CHANNEL_TRIGGER_STATUS_INVALID`
Value: *0*
Status is invalid.

`CONTROL_CHANNEL_TRIGGER_STATUS_SOFTWARE_SLOT_ALLOCATED`
Value: *1*
A software slot was allocated for the ControlChannelTrigger.

`CONTROL_CHANNEL_TRIGGER_STATUS_HARDWARE_SLOT_ALLOCATED`
Value: *2*
A hardware slot was allocated for the ControlChannelTrigger.

`CONTROL_CHANNEL_TRIGGER_STATUS_POLICY_ERROR`
Value: *3*
A status policy error.

`CONTROL_CHANNEL_TRIGGER_STATUS_SYSTEM_ERROR`
Value: *4*
A status system error.

`CONTROL_CHANNEL_TRIGGER_STATUS_TRANSPORT_DISCONNECTED`
Value: *5*
The TCP transport is disconnected.

`CONTROL_CHANNEL_TRIGGER_STATUS_SERVICE_UNAVAILABLE` Value: *6* Service is unavailable.	

Remarks

The CONTROL_CHANNEL_TRIGGER_STATUS structure is supported on Windows 8, and Windows Server 2012, and later versions of the operating system.

A CONTROL_CHANNEL_TRIGGER_STATUS enumeration value is returned as output from the SIO_QUERY_TRANSPORT_SETTING IOCTL to a query the **REAL_TIME_NOTIFICATION_CAPABILITY** transport setting for a TCP socket that is used with ControlChannelTrigger to receive background network notifications in a Windows Store app.

Requirements

Minimum supported client	Windows 8 [desktop apps only]
Minimum supported server	Windows Server 2012 [desktop apps only]
Header	mstcpip.h

See also

ControlChannelTrigger

REAL_TIME_NOTIFICATION_SETTING_OUTPUT

SIO_APPLY_TRANSPORT_SETTING

SIO_QUERY_TRANSPORT_SETTING

INET_PORT_RANGE structure (mstcpip.h)

10/22/2022 • 2 minutes to read • Edit Online

The **INET_PORT_RANGE** structure provides input data used by the SIO_ACQUIRE_PORT_RESERVATION IOCTL to acquire a runtime reservation for a block of TCP or UDP ports.

Syntax

```
typedef struct _INET_PORT_RANGE {
  USHORT StartPort;
  USHORT NumberOfPorts;
} INET_PORT_RANGE, *PINET_PORT_RANGE, INET_PORT_RESERVATION, *PINET_PORT_RESERVATION;
```

Members

`StartPort`

The starting TCP or UDP port number. If this parameter is set to zero, the system will choose a starting TCP or UDP port number.

`NumberOfPorts`

The number of TCP or UDP port numbers to reserve.

Remarks

The **INET_PORT_RANGE** structure is supported on Windows Vista and later.

The **INET_PORT_RANGE** structure is the datatype passed in the input buffer to the SIO_ACQUIRE_PORT_RESERVATION IOCTL. This IOCTL is used to acquire a runtime reservation for a block of TCP or UDP ports.

The **INET_PORT_RANGE** structure is typedefed to the **INET_PORT_RESERVATION** structure.

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Header	mstcpip.h

See also

CreatePersistentTcpPortReservation

CreatePersistentUdpPortReservation

DeletePersistentTcpPortReservation

DeletePersistentUdpPortReservation

INET_PORT_RESERVATION_INSTANCE

INET_PORT_RESERVATION_TOKEN

LookupPersistentTcpPortReservation

LookupPersistentUdpPortReservation

SIO_ACQUIRE_PORT_RESERVATION

SIO_ASSOCIATE_PORT_RESERVATION

SIO_RELEASE_PORT_RESERVATION

INET_PORT_RESERVATION_INSTANCE structure (mstcpip.h)

10/22/2022 • 2 minutes to read • Edit Online

The **INET_PORT_RESERVATION_INSTANCE** structure contains a port reservation and a token for a block of TCP or UDP ports.

Syntax

```
typedef struct {
  INET_PORT_RESERVATION       Reservation;
  INET_PORT_RESERVATION_TOKEN Token;
} INET_PORT_RESERVATION_INSTANCE, *PINET_PORT_RESERVATION_INSTANCE;
```

Members

`Reservation`

A runtime port reservation for a block of TCP or UDP ports.

The INET_PORT_RESERVATION structure is typedefed to the INET_PORT_RANGE structure.

`Token`

A port reservation token for a block of TCP or UDP ports.

Remarks

The **INET_PORT_RESERVATION_INSTANCE** structure is supported on Windows Vista and later.

The **INET_PORT_RESERVATION_INSTANCE** structure is returned by the SIO_ACQUIRE_PORT_RESERVATION IOCTL when acquiring a runtime reservation for a block of TCP or UDP ports.

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Header	mstcpip.h

See also

CreatePersistentTcpPortReservation

CreatePersistentUdpPortReservation

DeletePersistentTcpPortReservation

DeletePersistentUdpPortReservation

INET_PORT_RANGE

INET_PORT_RESERVATION_TOKEN

LookupPersistentTcpPortReservation

LookupPersistentUdpPortReservation

SIO_ACQUIRE_PORT_RESERVATION

SIO_ASSOCIATE_PORT_RESERVATION

SIO_RELEASE_PORT_RESERVATION

INET_PORT_RESERVATION_TOKEN structure (mstcpip.h)

10/22/2022 • 2 minutes to read • Edit Online

The **INET_PORT_RESERVATION_TOKEN** structure contains a port reservation token for a block of TCP or UDP ports.

Syntax

```
typedef struct {
  ULONG64 Token;
} INET_PORT_RESERVATION_TOKEN, *PINET_PORT_RESERVATION_TOKEN;
```

Members

`Token`

A port reservation token for a block of TCP or UDP ports.

Remarks

The **INET_PORT_RESERVATION_TOKEN** structure is supported on Windows Vista and later.

The **INET_PORT_RESERVATION_TOKEN** structure is used by the SIO_ACQUIRE_PORT_RESERVATION , SIO_ASSOCIATE_PORT_RESERVATION, and SIO_RELEASE_PORT_RESERVATION Ioctl for TCP or UDP port reservations. The **INET_PORT_RESERVATION_TOKEN** structure is also equivalent to the ULONG64 Token parameter used by the CreatePersistentTcpPortReservation, CreatePersistentUdpPortReservation, DeletePersistentTcpPortReservation, DeletePersistentUdpPortReservation, LookupPersistentTcpPortReservation, and LookupPersistentUdpPortReservation functions in IP Helper.

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Header	mstcpip.h

See also

CreatePersistentTcpPortReservation

CreatePersistentUdpPortReservation

DeletePersistentTcpPortReservation

DeletePersistentUdpPortReservation

INET_PORT_RANGE

INET_PORT_RESERVATION_INSTANCE

LookupPersistentTcpPortReservation

LookupPersistentUdpPortReservation

SIO_ACQUIRE_PORT_RESERVATION

SIO_ASSOCIATE_PORT_RESERVATION

SIO_RELEASE_PORT_RESERVATION

REAL_TIME_NOTIFICATION_SETTING_INPUT structure (mstcpip.h)

10/22/2022 • 2 minutes to read • Edit Online

The **REAL_TIME_NOTIFICATION_SETTING_INPUT** structure provides input settings to apply for the **REAL_TIME_NOTIFICATION_CAPABILITY** transport setting for a TCP socket that is used with ControlChannelTrigger to receive background network notifications in a Windows Store app.

Syntax

```
typedef struct _REAL_TIME_NOTIFICATION_SETTING_INPUT {
  TRANSPORT_SETTING_ID TransportSettingId;
  GUID                 BrokerEventGuid;
} REAL_TIME_NOTIFICATION_SETTING_INPUT, *PREAL_TIME_NOTIFICATION_SETTING_INPUT;
```

Members

`TransportSettingId`

The transport setting ID.

`BrokerEventGuid`

The realtime notification broker event GUID for this transport ID.

Remarks

The **REAL_TIME_NOTIFICATION_SETTING_INPUT** structure is supported on Windows 8, and Windows Server 2012, and later versions of the operating system.

If the TRANSPORT_SETTING_ID in the *lpvInBuffer* parameter passed to the SIO_APPLY_TRANSPORT_SETTING IOCTL 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 ControlChannelTrigger to receive background network notifications in a Windows Store app. The *lpvInBuffer* parameter should point to a **REAL_TIME_NOTIFICATION_SETTING_INPUT** structure used as input to the **SIO_APPLY_TRANSPORT_SETTING** IOCTL to apply the transport setting.

Requirements

Minimum supported client	Windows 8 [desktop apps only]
Minimum supported server	Windows Server 2012 [desktop apps only]
Header	mstcpip.h

See also

ControlChannelTrigger

REAL_TIME_NOTIFICATION_SETTING_OUTPUT

SIO_APPLY_TRANSPORT_SETTING

SIO_QUERY_TRANSPORT_SETTING

TRANSPORT_SETTING_ID

REAL_TIME_NOTIFICATION_SETTING_OUTPUT structure (mstcpip.h)

10/22/2022 • 2 minutes to read • Edit Online

The REAL_TIME_NOTIFICATION_SETTING_OUTPUT structure provides the output settings from a query for the **REAL_TIME_NOTIFICATION_CAPABILITY** transport setting for a TCP socket that is used with ControlChannelTrigger to receive background network notifications in a Windows Store app.

Syntax

```
typedef struct _REAL_TIME_NOTIFICATION_SETTING_OUTPUT {
  CONTROL_CHANNEL_TRIGGER_STATUS ChannelStatus;
} REAL_TIME_NOTIFICATION_SETTING_OUTPUT, *PREAL_TIME_NOTIFICATION_SETTING_OUTPUT;
```

Members

`ChannelStatus`

The channel status for a socket that is used with the ControlChannelTrigger.

Remarks

The REAL_TIME_NOTIFICATION_SETTING_OUTPUT structure is supported on Windows 8, and Windows Server 2012, and later versions of the operating system.

If the TRANSPORT_SETTING_ID in the *lpvInBuffer* parameter passed to the SIO_QUERY_TRANSPORT_SETTING IOCTL 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 ControlChannelTrigger to receive background network notifications in a Windows Store app. If the WSAIoctl or LPWSPIoctl call is successful, this IOCTL returns a REAL_TIME_NOTIFICATION_SETTING_OUTPUT structure with the current status.

Requirements

Minimum supported client	Windows 8 [desktop apps only]
Minimum supported server	Windows Server 2012 [desktop apps only]
Header	mstcpip.h

See also

CONTROL_CHANNEL_TRIGGER_STATUS

ControlChannelTrigger

REAL_TIME_NOTIFICATION_SETTING_INPUT

SIO_APPLY_TRANSPORT_SETTING

SIO_QUERY_TRANSPORT_SETTING

SOCKET_PEER_TARGET_NAME structure (mstcpip.h)

10/22/2022 • 2 minutes to read • Edit Online

The **SOCKET_PEER_TARGET_NAME** structure contains the IP address and name for a peer target and the type of security protocol to be used on a socket.

Syntax

```
typedef struct _SOCKET_PEER_TARGET_NAME {
  SOCKET_SECURITY_PROTOCOL SecurityProtocol;
  SOCKADDR_STORAGE         PeerAddress;
  ULONG                    PeerTargetNameStringLen;
  wchar_t                  AllStrings[0];
} SOCKET_PEER_TARGET_NAME;
```

Members

`SecurityProtocol`

A SOCKET_SECURITY_PROTOCOL value that identifies the type of protocol used to secure the traffic on the socket.

`PeerAddress`

The IP address of the peer for the socket.

`PeerTargetNameStringLen`

The length, in bytes, of the peer target name in the **AllStrings** member.

`AllStrings[0]`

The peer target name for the socket.

Remarks

The **SOCKET_PEER_TARGET_NAME** structure is supported on Windows Vista and later.

The **SOCKET_PEER_TARGET_NAME** structure is used by the WSASetSocketPeerTargetName function to specify the peer target name 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.

Currently, the only type of security protocol that is supported is IPsec. So specifying an enumeration value of **SOCKET_SECURITY_PROTOCOL_DEFAULT** has the same effect as specifying **SOCKET_SECURITY_PROTOCOL_IPSEC** in the **SecurityProtocol** member.

The implementation of IPsec on Windows Vista and Windows Server 2008 only supports computer-to-computer and user-to-computer authentication. As a result, the peer target name specified in the **AllStrings** member of the **SOCKET_PEER_TARGET_NAME** structure should refer to the peer computer principal.

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Header	mstcpip.h

See also

SOCKADDR_STORAGE

SOCKET_SECURITY_PROTOCOL

Using Secure Socket Extensions

WSASetSocketPeerTargetName

Windows Filtering Platform

Windows Filtering Platform API Functions

Winsock Secure Socket Extensions

SOCKET_SECURITY_PROTOCOL enumeration (mstcpip.h)

10/22/2022 • 2 minutes to read • Edit Online

The **SOCKET_SECURITY_PROTOCOL** enumeration indicates the type of security protocol to be used on a socket to secure network traffic.

Syntax

```
typedef enum _SOCKET_SECURITY_PROTOCOL {
  SOCKET_SECURITY_PROTOCOL_DEFAULT,
  SOCKET_SECURITY_PROTOCOL_IPSEC,
  SOCKET_SECURITY_PROTOCOL_IPSEC2,
  SOCKET_SECURITY_PROTOCOL_INVALID
} SOCKET_SECURITY_PROTOCOL;
```

Constants

`SOCKET_SECURITY_PROTOCOL_DEFAULT` The default system security will be used.
`SOCKET_SECURITY_PROTOCOL_IPSEC` IPsec will be used.
`SOCKET_SECURITY_PROTOCOL_IPSEC2`
`SOCKET_SECURITY_PROTOCOL_INVALID` The maximum possible value for the SOCKET_SECURITY_PROTOCOL enumeration type. This is not a legal value.

Remarks

This enumeration is supported on Windows Vista and later.

Currently, the only type of security protocol that is supported is IPsec. So specifying an enumeration value of **SOCKET_SECURITY_PROTOCOL_DEFAULT** has the same effect as specifying **SOCKET_SECURITY_PROTOCOL_IPSEC**.

The **SOCKET_SECURITY_PROTOCOL** enumeration is used in the SOCKET_PEER_TARGET_NAME, SOCKET_SECURITY_QUERY_INFO, SOCKET_SECURITY_QUERY_TEMPLATE, SOCKET_SECURITY_SETTINGS, and SOCKET_SECURITY_SETTINGS_IPSEC structures to indicate the type of security protocol to be used on a socket in the **SecurityProtocol** member. These structures are used by the WSAQuerySocketSecurity, WSASetSocketPeerTargetName, and WSASetSocketSecurity functions.

In addition to identifying the security protocol, this type is also used to decide how to interpret a pointer passed to some of the secure socket functions. This is analogous to how the **sa_family** member of the sockaddr type is used to interpret a pointer as either **sockaddr_in** or **sockaddr_in6**.

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Header	mstcpip.h

See also

SOCKET_PEER_TARGET_NAME

SOCKET_SECURITY_QUERY_INFO

SOCKET_SECURITY_QUERY_TEMPLATE

SOCKET_SECURITY_SETTINGS

SOCKET_SECURITY_SETTINGS_IPSEC

Using Secure Socket Extensions

WSAQuerySocketSecurity

WSASetSocketPeerTargetName

WSASetSocketSecurity

Windows Filtering Platform

Windows Filtering Platform API Functions

Winsock Secure Socket Extensions

sockaddr

SOCKET_SECURITY_QUERY_INFO structure (mstcpip.h)

10/22/2022 • 2 minutes to read • Edit Online

The **SOCKET_SECURITY_QUERY_INFO** structure contains security information returned by the WSAQuerySocketSecurity function.

Syntax

```
typedef struct _SOCKET_SECURITY_QUERY_INFO {
  SOCKET_SECURITY_PROTOCOL SecurityProtocol;
  ULONG                    Flags;
  UINT64                   PeerApplicationAccessTokenHandle;
  UINT64                   PeerMachineAccessTokenHandle;
} SOCKET_SECURITY_QUERY_INFO;
```

Members

`SecurityProtocol`

A SOCKET_SECURITY_PROTOCOL value that identifies the protocol used to secure the traffic.

`Flags`

The set of possible security flags for the connection defined in the *Mstcpip.h* header file.

VALUE	MEANING
SOCKET_INFO_CONNECTION_SECURED 0x00000001	If present, traffic is being secured by a security protocol. If absent, the traffic is flowing in the clear.
SOCKET_INFO_CONNECTION_ENCRYPTED 0x00000002	If present, the connection traffic is being encrypted. The **SOCKET_INFO_CONNECTION_SECURED** flag is always set when this flag is present.

`PeerApplicationAccessTokenHandle`

A handle to the access token that represents the account under which the peer application is running. After using the token for access checks, the application should close the handle using the CloseHandle function.

`PeerMachineAccessTokenHandle`

A handle to the access token for the peer computer's account during the course of the application. After using the token for access checks, the application should close the handle using the CloseHandle function.

Remarks

The **SOCKET_SECURITY_QUERY_INFO** structure is supported on Windows Vista and later.

The **SOCKET_SECURITY_QUERY_INFO** structure is used by the WSAQuerySocketSecurity function to return information about the security applied to a connection on a socket.

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Header	mstcpip.h

See also

CloseHandle

SOCKET_SECURITY_PROTOCOL

Using Secure Socket Extensions

WSAQuerySocketSecurity

Windows Filtering Platform

Windows Filtering Platform API Functions

Winsock Secure Socket Extensions

SOCKET_SECURITY_QUERY_TEMPLATE structure (mstcpip.h)

10/22/2022 • 2 minutes to read • Edit Online

The **SOCKET_SECURITY_QUERY_TEMPLATE** structure contains the security template used by the WSAQuerySocketSecurity function.

Syntax

```
typedef struct _SOCKET_SECURITY_QUERY_TEMPLATE {
  SOCKET_SECURITY_PROTOCOL SecurityProtocol;
  SOCKADDR_STORAGE         PeerAddress;
  ULONG                    PeerTokenAccessMask;
} SOCKET_SECURITY_QUERY_TEMPLATE;
```

Members

`SecurityProtocol`

A SOCKET_SECURITY_PROTOCOL value that identifies the protocol used to secure the traffic.

`PeerAddress`

The IP address of the peer for which security information is being queried. For connection-oriented sockets (protocol of **IPPROTO_TCP**), the connected socket uniquely identifies a peer. In this case, this parameter is ignored.

`PeerTokenAccessMask`

The access mask used for opening the peer user application and computer token handles that are returned as part of the query information.

Remarks

The **SOCKET_SECURITY_QUERY_TEMPLATE** structure is supported on Windows Vista and later.

The **SOCKET_SECURITY_QUERY_TEMPLATE** structure is used by the WSAQuerySocketSecurity function to specify the type of query information to return for a socket. The **SOCKET_SECURITY_QUERY_TEMPLATE** structure passed to the **WSAQuerySocketSecurity** function may contain zeros for all members to request default security information.

If the **SOCKET_SECURITY_QUERY_TEMPLATE** structure is specified with the **PeerTokenAccessMask** member not specified (set to zero), then the WSAQuerySocketSecurity function will not return the **PeerApplicationAccessTokenHandle** and **PeerMachineAccessTokenHandle** members in the SOCKET_SECURITY_QUERY_INFO structure.

Currently, the only type of security protocol that is supported is IPsec. So specifying an enumeration value of **SOCKET_SECURITY_PROTOCOL_DEFAULT** for the **SecurityProtocol** member has the same effect as specifying **SOCKET_SECURITY_PROTOCOL_IPSEC**.

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Header	mstcpip.h

See also

SOCKET_SECURITY_PROTOCOL

SOCKET_SECURITY_QUERY_INFO

Using Secure Socket Extensions

WSAQuerySocketSecurity

Windows Filtering Platform

Windows Filtering Platform API Functions

Winsock Secure Socket Extensions

SOCKET_SECURITY_SETTINGS structure (mstcpip.h)

10/22/2022 • 2 minutes to read • Edit Online

The **SOCKET_SECURITY_SETTINGS** structure specifies generic security requirements for a socket.

Syntax

```
typedef struct _SOCKET_SECURITY_SETTINGS {
  SOCKET_SECURITY_PROTOCOL SecurityProtocol;
  ULONG                    SecurityFlags;
} SOCKET_SECURITY_SETTINGS;
```

Members

`SecurityProtocol`

A SOCKET_SECURITY_PROTOCOL value that identifies the type of security protocol to be used on the socket.

`SecurityFlags`

A set of flags that allow applications to set specific security requirements on a socket. The possible values are defined in the *Mstcpip.h* header file.

VALUE	MEANING
SOCKET_SETTINGS_GUARANTEE_ENCRYPTION 0x00000001	Indicates that guaranteed encryption of traffic is required. This flag should be set if the default policy prefers methods of protection that do not use encryption. If this flag is set and encryption is not possible for any reason, no packets will be sent and a connection will not be established.
SOCKET_SETTINGS_ALLOW_INSECURE 0x00000002	Indicates that clear text connections are allowed. If this flag is set, some or all of the sent packets will be sent in clear text, especially if security with the peer could not be negotiated. **Note** If this flag is not set, it is guaranteed that packets will never be sent in clear text, even if security negotiation fails.

Remarks

The **SOCKET_SECURITY_SETTINGS** structure is supported on Windows Vista and later.

The **SOCKET_SECURITY_SETTINGS** structure is used by the WSASetSocketSecurity function to enable and apply security on a socket.

Security settings not addressed in this structure are derived from the system default policy or the administratively configured policy. It is recommended that most applications specify a value of **SOCKET_SECURITY_PROTOCOL_DEFAULT** for the SOCKET_SECURITY_PROTOCOL enumeration in the **SecurityProtocol** member. This makes the application neutral to security protocols and allows easier deployments among different systems.

Advanced applications can specify a security protocol and associated settings by casting them to the **SOCKET_SECURITY_SETTINGS** type.

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Header	mstcpip.h

See also

SOCKET_SECURITY_PROTOCOL

Using Secure Socket Extensions

WSASetSocketSecurity

Windows Filtering Platform

Windows Filtering Platform API Functions

Winsock Secure Socket Extensions

SOCKET_SECURITY_SETTINGS_IPSEC structure (mstcpip.h)

10/22/2022 • 3 minutes to read • Edit Online

The **SOCKET_SECURITY_SETTINGS_IPSEC** structure specifies various security requirements and settings that are specific to IPsec.

Syntax

```
typedef struct _SOCKET_SECURITY_SETTINGS_IPSEC {
  SOCKET_SECURITY_PROTOCOL SecurityProtocol;
  ULONG                    SecurityFlags;
  ULONG                    IpsecFlags;
  GUID                     AuthipMMPolicyKey;
  GUID                     AuthipQMPolicyKey;
  GUID                     Reserved;
  UINT64                   Reserved2;
  ULONG                    UserNameStringLen;
  ULONG                    DomainNameStringLen;
  ULONG                    PasswordStringLen;
  wchar_t                  AllStrings[0];
} SOCKET_SECURITY_SETTINGS_IPSEC;
```

Members

`SecurityProtocol`

Type: **SOCKET_SECURITY_PROTOCOL**

A SOCKET_SECURITY_PROTOCOL value that identifies the type of security protocol to be used on the socket. This member must be set to **SOCKET_SECURITY_PROTOCOL_IPSEC**.

`SecurityFlags`

Type: **ULONG**

A set of flags that allow applications to set specific security requirements on a socket. The possible values are defined in the *Mstcpip.h* header file.

VALUE	MEANING
SOCKET_SETTINGS_GUARANTEE_ENCRYPTION 0x00000001	Indicates that guaranteed encryption of traffic is required. This flag should be set if the default policy prefers methods of protection that do not use encryption. If this flag is set and encryption is not possible for any reason, no packets will be sent and a connection will not be established.

SOCKET_SETTINGS_ALLOW_INSECURE 0x00000002	Indicates that clear text connections are allowed. If this flag is set, some or all of the sent packets will be sent in clear text, especially if security with the peer could not be negotiated. **Note** If this flag is not set, it is guaranteed that packets will never be sent in clear text, even if security negotiation fails.

`IpsecFlags`

Type: **ULONG**

Flags for IPsec security settings. The possible values are defined in the *Mstcpip.h* header file.

VALUE	MEANING
SOCKET_SETTINGS_IPSEC_SKIP_FILTER_INSTANTIATION 0x00000001	When this flag is set, IPsec filter instantiation is omitted for the socket. This flag should be set when an application knows that IPsec filters and policy already exist for its traffic. Applications running on a domain with IPsec policy in place can also set this flag.

`AuthipMMPolicyKey`

Type: **GUID**

The GUID for the Windows Filtering Platform key of the AuthIP main mode provider context. If an application wishes to use a custom main mode policy, it should first use the FwpmProviderContextAdd0 function to add the corresponding provider context and specify the returned key in this member. This field is ignored for a GUID of zero.

`AuthipQMPolicyKey`

Type: **GUID**

The Windows Filtering Platform key of the AuthIp quick mode provider context. If an application wishes to use a custom quick mode policy, it should first use the FwpmProviderContextAdd0 function to add the corresponding provider context and specify the returned key in this field. This field is ignored for a GUID of zero.

`Reserved`

Type: **GUID**

Reserved for future use.

`Reserved2`

Type: **UINT64**

Reserved for future use.

`UserNameStringLen`

Type: **ULONG**

The length, in bytes, of the user name in the **AllStrings** member.

`DomainNameStringLen`

Type: **ULONG**

The length, in bytes, of the domain name in the **AllStrings** member.

`PasswordStringLen`

Type: **ULONG**

The length, in bytes, of the password in the **AllStrings** member.

`AllStrings[0]`

Type: **wchar_t[]**

A string that contains the user name, the domain name, and the password concatenated in this order.

Remarks

The **SOCKET_SECURITY_SETTINGS_IPSEC** structure is supported on Windows Vista and later.

The **SOCKET_SECURITY_SETTINGS_IPSEC** structure is meant to be used by an advanced application that requires more flexibility and wishes to customize IPSec policy for their traffic. The pointer to the **SOCKET_SECURITY_SETTINGS_IPSEC** structure needs to cast to the SOCKET_SECURITY_SETTINGS structure type when calling the WSASetSocketSecurity function to enable and apply security on a socket.

The **SecurityProtocol** member of the **SOCKET_SECURITY_SETTINGS_IPSEC** structure must be set to **SOCKET_SECURITY_PROTOCOL_IPSEC**, not **SOCKET_SECURITY_PROTOCOL_DEFAULT**.

To simplify Internet Protocol security (IPsec) deployment, Windows Vista and later support an enhanced version of the Internet Key Exchange (IKE) protocol known as Authenticated Internet Protocol (AuthIP). AuthIP provides simplified IPsec policy configuration and maintenance in many configurations and additional flexibility for IPsec peer authentication.

There is a possibility that some of the IPsec settings specified in the **SOCKET_SECURITY_SETTINGS_IPSEC** structure may end up being different from the actual settings applied to the network traffic on a socket. For example, this could happen when an application specifies custom main mode or quick mode policy, but a different policy with a higher priority (a domain policy, for example) specifies conflicting settings for the same traffic. To be aware of such conflicts, an application can use the Windows Filtering Platform API to query the policy being applied and subscribe for notifications.

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Header	mstcpip.h

See also

About Windows Filtering Platform

AuthIP in Windows Vista

FwpmProviderContextAdd0

SOCKET_SECURITY_PROTOCOL

SOCKET_SECURITY_SETTINGS

Using Secure Socket Extensions

WSASetSocketSecurity

Windows Filtering Platform

Windows Filtering Platform API Functions

Winsock Secure Socket Extensions

SOCKET_USAGE_TYPE enumeration (mstcpip.h)

10/22/2022 • 2 minutes to read • Edit Online

The Windows Sockets **SOCKET_USAGE_TYPE** enumeration is used to specified the usage type for the socket.

Syntax

```
typedef enum _SOCKET_USAGE_TYPE {
  SYSTEM_CRITICAL_SOCKET = 1
} SOCKET_USAGE_TYPE;
```

Constants

`SYSTEM_CRITICAL_SOCKET` Value: *1* The usage type is critical to the system.

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Header	mstcpip.h

TCP_INFO_v0 structure (mstcpip.h)

10/22/2022 • 2 minutes to read • Edit Online

Contains the Transmission Control Protocol (TCP) statistics that were collected for a socket.

Syntax

```
typedef struct _TCP_INFO_v0 {
  TCPSTATE State;
  ULONG    Mss;
  ULONG64  ConnectionTimeMs;
  BOOLEAN  TimestampsEnabled;
  ULONG    RttUs;
  ULONG    MinRttUs;
  ULONG    BytesInFlight;
  ULONG    Cwnd;
  ULONG    SndWnd;
  ULONG    RcvWnd;
  ULONG    RcvBuf;
  ULONG64  BytesOut;
  ULONG64  BytesIn;
  ULONG    BytesReordered;
  ULONG    BytesRetrans;
  ULONG    FastRetrans;
  ULONG    DupAcksIn;
  ULONG    TimeoutEpisodes;
  UCHAR    SynRetrans;
} TCP_INFO_v0, *PTCP_INFO_v0;
```

Members

`State`

A value from the TCPSTATE enumeration that indicates the state of the TCP connection.

`Mss`

The current maximum segment size (MSS) for the connection, in bytes.

`ConnectionTimeMs`

The lifetime of the connection, in milliseconds.

`TimestampsEnabled`

TRUE if TCP time stamps are turned on for the connection; otherwise **FALSE**.

`RttUs`

The current estimated round-trip time for the connection, in microseconds.

`MinRttUs`

The minimum sampled round trip time, in microseconds.

`BytesInFlight`

The current number of sent bytes that are unacknowledged.

`Cwnd`

The size of the current congestion window, in bytes.

`SndWnd`

The size of the send window (SND.WND in RFC 793), in bytes.

`RcvWnd`

The size of the receive window (RCV.WND in RFC 793), in bytes.

`RcvBuf`

The size of the current receive buffer, in bytes. The size of the receive buffer changes dynamically when autotuning is turned on for the receive window.

`BytesOut`

The total number of bytes sent.

`BytesIn`

The total number of bytes received.

`BytesReordered`

The total number of bytes reordered.

`BytesRetrans`

The total number of bytes retransmitted.

`FastRetrans`

The number of calls of the Fast Retransmit algorithm.

`DupAcksIn`

The total number of duplicate acknowledgments received.

`TimeoutEpisodes`

The total number of retransmission timeout episodes. Each episode can consist of multiple timeouts.

`SynRetrans`

The total number of retransmitted synchronize control flags (SYNs).

Remarks

To get an instance of this structure, call the WSAIoctl or LPWSPIoctl function with the SIO_TCP_INFO control code. Specify 0 for the *lpvInBuffer* field to retrieve the v0 version of this structure.

Requirements

Minimum supported client	Windows 10, version 1703 [desktop apps only]
Minimum supported server	Windows Server 2016 [desktop apps only]

Header	mstcpip.h

See also

SIO_TCP_INFO

TCPSTATE

TCP_INFO_v1 structure (mstcpip.h)

10/22/2022 • 2 minutes to read • Edit Online

Contains the Transmission Control Protocol (TCP) statistics that were collected for a socket. Version 1.0 of this structure provides additional fields.

Syntax

```
typedef struct _TCP_INFO_v1 {
  TCPSTATE State;
  ULONG    Mss;
  ULONG64  ConnectionTimeMs;
  BOOLEAN  TimestampsEnabled;
  ULONG    RttUs;
  ULONG    MinRttUs;
  ULONG    BytesInFlight;
  ULONG    Cwnd;
  ULONG    SndWnd;
  ULONG    RcvWnd;
  ULONG    RcvBuf;
  ULONG64  BytesOut;
  ULONG64  BytesIn;
  ULONG    BytesReordered;
  ULONG    BytesRetrans;
  ULONG    FastRetrans;
  ULONG    DupAcksIn;
  ULONG    TimeoutEpisodes;
  UCHAR    SynRetrans;
  ULONG    SndLimTransRwin;
  ULONG    SndLimTimeRwin;
  ULONG64  SndLimBytesRwin;
  ULONG    SndLimTransCwnd;
  ULONG    SndLimTimeCwnd;
  ULONG64  SndLimBytesCwnd;
  ULONG    SndLimTransSnd;
  ULONG    SndLimTimeSnd;
  ULONG64  SndLimBytesSnd;
} TCP_INFO_v1, *PTCP_INFO_v1;
```

Members

`State`

Contains the Transmission Control Protocol (TCP) statistics that were collected for a socket.

`Mss`

The current maximum segment size (MSS) for the connection, in bytes.

`ConnectionTimeMs`

The lifetime of the connection, in milliseconds.

`TimestampsEnabled`

TRUE if TCP time stamps are turned on for the connection; otherwise **FALSE**.

`RttUs`

The current estimated round-trip time for the connection, in microseconds.

`MinRttUs`

The minimum sampled round trip time, in microseconds.

`BytesInFlight`

The current number of sent bytes that are unacknowledged.

`Cwnd`

The size of the current congestion window, in bytes.

`SndWnd`

The size of the send window (SND.WND in RFC 793), in bytes.

`RcvWnd`

The size of the receive window (RCV.WND in RFC 793), in bytes.

`RcvBuf`

The size of the current receive buffer, in bytes. The size of the receive buffer changes dynamically when autotuning is turned on for the receive window.

`BytesOut`

The total number of bytes sent.

`BytesIn`

The total number of bytes received.

`BytesReordered`

The total number of bytes reordered.

`BytesRetrans`

The total number of bytes retransmitted.

`FastRetrans`

The number of calls of the Fast Retransmit algorithm.

`DupAcksIn`

The total number of duplicate acknowledgments received.

`TimeoutEpisodes`

The total number of retransmission timeout episodes. Each episode can consist of multiple timeouts.

`SynRetrans`

The total number of retransmitted synchronize control flags (SYNs).

`SndLimTransRwin`

The number of transitions into the "Receiver Limited" state from either the "Congestion Limited" or "Sender Limited" states.

`SndLimTimeRwin`

The cumulative time, in milliseconds, spent in the "Receiver Limited" state where TCP transmission stops because the sender has filled the announced receiver window.

`SndLimBytesRwin`

The total number of bytes sent in the "Receiver Limited" state.

`SndLimTransCwnd`

The number of transitions into the "Congestion Limited" state from either the "Receiver Limited" or "Sender Limited" states.

`SndLimTimeCwnd`

The cumulative time, in milliseconds, spent in the "Congestion Limited" state. When there is a retransmission timeout, it is counted in this member and not the cumulative time for some other state.

`SndLimBytesCwnd`

The total number of bytes sent in the "Congestion Limited" state.

`SndLimTransSnd`

The number of transitions into the "Sender Limited" state from either the "Receiver Limited" or "Congestion Limited" states.

`SndLimTimeSnd`

The cumulative time, in milliseconds, spent in the "Sender Limited" state.

`SndLimBytesSnd`

The total number of bytes sent in the "Sender Limited" state.

Remarks

To get an instance of this structure, call the WSAIoctl or LPWSPIoctl function with the SIO_TCP_INFO control code. Specify 1 for the *lpvInBuffer* field to retrieve the v1 version of this structure.

Requirements

Minimum supported client	Windows 10 Build 20348
Minimum supported server	Windows 10 Build 20348
Header	mstcpip.h

See also

SIO_TCP_INFO

TCPSTATE

TCP_INITIAL_RTO_PARAMETERS structure (mstcpip.h)

10/22/2022 • 2 minutes to read • Edit Online

The TCP_INITIAL_RTO_PARAMETERS structure specifies data used by the SIO_TCP_INITIAL_RTO IOCTL to configure initial re-transmission timeout (RTO) parameters to be used on the socket.

Syntax

```
typedef struct _TCP_INITIAL_RTO_PARAMETERS {
  USHORT Rtt;
  UCHAR  MaxSynRetransmissions;
} TCP_INITIAL_RTO_PARAMETERS, *PTCP_INITIAL_RTO_PARAMETERS;
```

Members

`Rtt`

Supplies the initial RTT in milliseconds.

`MaxSynRetransmissions`

Supplies the number of retransmissions attempted before the connection setup fails.

Remarks

The TCP_INITIAL_RTO_PARAMETERS structure 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 re-transmissions that will be attempted before the connection attempt fails.

An application should supply the RTT of choice in milliseconds and the maximum number of retransmissions in this structure. The Windows TCP/IP stack will honor these parameters for the subsequent connection attempt. The retransmission behavior for TCP is documented in IETF RFC 793 and 2988.

An application may use the unspecified defines, **TCP_INITIAL_RTO_UNSPECIFIED_RTT** and **TCP_INITIAL_RTO_UNSPECIFIED_MAX_SYN_RETRANSMISSIONS** when supplying values for one of these fields. This allows the system to pick up administrator configured settings for the parameter left unspecified.

An application can choose system defaults for any of these fields and supply those values using the default defines, **TCP_INITIAL_RTO_DEFAULT_RTT** and **TCP_INITIAL_RTO_DEFAULT_MAX_SYN_RETRANSMISSIONS**.

Requirements

Minimum supported client	Windows 8 [desktop apps only]
Minimum supported server	Windows Server 2012 [desktop apps only]

Header	mstcpip.h

See also

SIO_TCP_INITIAL_RTO

TCPSTATE enumeration (mstcpip.h)

10/22/2022 • 2 minutes to read • Edit Online

The Windows Sockets **TCPSTATE** enumeration indicates the possible states of a Transmission Control Protocol (TCP) connection.

Syntax

```
typedef enum _TCPSTATE {
  TCPSTATE_CLOSED,
  TCPSTATE_LISTEN,
  TCPSTATE_SYN_SENT,
  TCPSTATE_SYN_RCVD,
  TCPSTATE_ESTABLISHED,
  TCPSTATE_FIN_WAIT_1,
  TCPSTATE_FIN_WAIT_2,
  TCPSTATE_CLOSE_WAIT,
  TCPSTATE_CLOSING,
  TCPSTATE_LAST_ACK,
  TCPSTATE_TIME_WAIT,
  TCPSTATE_MAX
} TCPSTATE;
```

Constants

`TCPSTATE_CLOSED`	The TCP connection has no connection state at all. This state represents the state when there is no Transmission Control Block (TCB), and therefore, no connection.
`TCPSTATE_LISTEN`	The TCP connection is waiting for a connection request from any remote TCP and port.
`TCPSTATE_SYN_SENT`	-The TCP connection is waiting for a matching connection request after sending a connection request.
`TCPSTATE_SYN_RCVD`	The TCP connection is waiting for an acknowledgment that confirms the connection request after both receiving and sending a connection request.
`TCPSTATE_ESTABLISHED`	The TCP connection is an open connection, so the data received can be delivered to the user. This state is normal state for the data transfer phase of the connection.

`TCPSTATE_FIN_WAIT_1`	The TCP connection is waiting for a request to end the connection from the remote TCP, or an acknowledgment of the previously sent request to end the connection.
`TCPSTATE_FIN_WAIT_2`	The TCP connection is waiting for a request to end the connection from the remote TCP.
`TCPSTATE_CLOSE_WAIT`	The TCP connection is waiting for a request to end the connection from the local user.
`TCPSTATE_CLOSING`	The TCP connection is waiting for an acknowledgment of the request to end the connection from the remote TCP.
`TCPSTATE_LAST_ACK`	The TCP connection is waiting for an acknowledgment of the request to end the connection that was previously sent to the remote TCP, which includes an acknowledgment of its request to end the connection.
`TCPSTATE_TIME_WAIT`	The TCP connection is waiting for enough time to pass to be sure the remote TCP received the acknowledgment of its request to end the connection.
`TCPSTATE_MAX`	The maximum value of the TCPSTATE enumeration.

Remarks

A TCP connection progresses from one state to another in response to events. The events are the user calls OPEN, SEND, RECEIVE, CLOSE, ABORT, and STATUS; the incoming segments, particularly those containing the SYN, ACK, RST and FIN flags; and timeouts.

For more information about TCP connection states, see RFC 793.

Requirements

Minimum supported client	Windows 10, version 1703 [desktop apps only]
Minimum supported server	Windows Server 2016 [desktop apps only]
Header	mstcpip.h

See also

SIO_TCP_INFO

TCP_INFO_v0

TIMESTAMPING_CONFIG structure (mstcpip.h)

10/22/2022 • 2 minutes to read • Edit Online

Describes the input structure used by the SIO_TIMESTAMPING configuration IOCTL to configure timestamp reception for a datagram socket.

Syntax

```
typedef struct _TIMESTAMPING_CONFIG {
  ULONG  Flags;
  USHORT TxTimestampsBuffered;
} TIMESTAMPING_CONFIG, *PTIMESTAMPING_CONFIG;
```

Members

`Flags`

Type: **ULONG**

Enable/disable timestamp reception for rx/tx direction.

Use the values **TIMESTAMPING_FLAG_RX** (0x1) and **TIMESTAMPING_FLAG_TX** (0x2) (both defined in `mstcpip.h`). Specify a value to enable timestamp reception for that direction; and omit a value to disable timestamp reception for that direction.

`TxTimestampsBuffered`

Type: **USHORT**

Determines how many tx timestamps may be buffered. When the count of tx timestamps that have been buffered reaches a value equal to *TxTimestampsBuffered*, and a new tx timestamp has been generated, the new timestamp will be discarded.

Requirements

Minimum supported client	Windows 10 Build 20348
Minimum supported server	Windows 10 Build 20348
Header	mstcpip.h

TRANSPORT_SETTING_ID structure (mstcpip.h)

10/22/2022 • 2 minutes to read • Edit Online

The **TRANSPORT_SETTING_ID** structure specifies the transport setting ID used by the SIO_APPLY_TRANSPORT_SETTING and SIO_QUERY_TRANSPORT_SETTING IOCTLs to apply or query the transport setting for a socket.

Syntax

```
typedef struct TRANSPORT_SETTING_ID {
  GUID Guid;
} TRANSPORT_SETTING_ID, *PTRANSPORT_SETTING_ID;
```

Members

`Guid`

The transport setting ID.

Remarks

The only transport setting defined for Windows 8 and Windows Server 2012 is for the **REAL_TIME_NOTIFICATION_CAPABILITY** capability on a TCP socket. For Windows 10 and Windows Server 2016, there is another transport setting defined as **ASSOCIATE_NAMERES_CONTEXT**.

The **TRANSPORT_SETTING_ID** structure is passed as input to the SIO_APPLY_TRANSPORT_SETTING and SIO_QUERY_TRANSPORT_SETTING IOCTLs. The **Guid** member determines what transport setting is applied or queried.

The only transport setting currently defines is for the **REAL_TIME_NOTIFICATION_CAPABILITY** capability on a TCP socket.

Requirements

Minimum supported client	Windows 8 [desktop apps only]
Minimum supported server	Windows Server 2012 [desktop apps only]
Header	mstcpip.h (include Mstcpip.h)

See also

ControlChannelTrigger

REAL_TIME_NOTIFICATION_SETTING_OUTPUT

SIO_APPLY_TRANSPORT_SETTING

mswsock.h header

10/22/2022 • 2 minutes to read • Edit Online

This header is used by Windows Sockets 2. For more information, see:

- Windows Sockets 2

mswsock.h contains the following programming interfaces:

Functions

AcceptEx The AcceptEx function (mswsock.h) accepts a new connection, returns the local and remote address, and receives the first block of data sent by the client application.
GetAcceptExSockaddrs The GetAcceptExSockaddrs function (mswsock.h) parses data obtained from a call to the AcceptEx function and passes local and remote addresses to a sockaddr structure.
TransmitFile The TransmitFile function (mswsock.h) transmits file data over a connected socket handle.
WSARecvEx The WSARecvEx function (mswsock.h) receives data from a connected socket or a bound connectionless socket.

Callback functions

LPFN_CONNECTEX The ConnectEx function establishes a connection to a specified socket, and optionally sends data once the connection is established.
LPFN_DISCONNECTEX Closes a connection on a socket, and allows the socket handle to be reused.Note This function is a Microsoft-specific extension to the Windows Sockets specification.
LPFN_RIOCLOSECOMPLETIONQUEUE Closes an existing completion queue used for I/O completion notification by send and receive requests with the Winsock registered I/O extensions.
LPFN_RIOCREATECOMPLETIONQUEUE Creates an I/O completion queue of a specific size for use with the Winsock registered I/O extensions.

LPFN_RIOCREATEREQUESTQUEUE

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.

LPFN_RIODEQUEUECOMPLETION

Removes entries from an I/O completion queue for use with the Winsock registered I/O extensions.

LPFN_RIODEREGISTERBUFFER

Deregisters a registered buffer used with the Winsock registered I/O extensions.

LPFN_RIONOTIFY

Registers the method to use for notification behavior with an I/O completion queue for use with the Winsock registered I/O extensions.

LPFN_RIORECEIVE

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.

LPFN_RIORECEIVEEX

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.

LPFN_RIOREGISTERBUFFER

Registers a RIO_BUFFERID, a registered buffer descriptor, with a specified buffer for use with the Winsock registered I/O extensions.

LPFN_RIORESIZECOMPLETIONQUEUE

Resizes an I/O completion queue to be either larger or smaller for use with the Winsock registered I/O extensions.

LPFN_RIORESIZEREQUESTQUEUE

Resizes a request queue to be either larger or smaller for use with the Winsock registered I/O extensions.

LPFN_RIOSEND

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.

LPFN_RIOSENDEX

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.

LPFN_TRANSMITPACKETS

Transmits in-memory data or file data over a connected socket.

LPFN_WSARECVMSG

LPFN_WSARECVMSG is a function pointer type. You implement a matching **WSARecvMsg** callback function in your app. The system uses your callback function to transmit to you in-memory data, or file data, over a connected socket.

Structures

RIO_EXTENSION_FUNCTION_TABLE

Contains information on the functions that implement the Winsock registered I/O extensions.

RIO_NOTIFICATION_COMPLETION

Specifies the method for I/O completion to be used with a RIONotify function for sending or receiving network data with the Winsock registered I/O extensions.

TRANSMIT_FILE_BUFFERS

The TRANSMIT_FILE_BUFFERS structure (mswsock.h) specifies data to be transmitted before and after file data during a TransmitFile function file transfer operation.

TRANSMIT_PACKETS_ELEMENT

Specifies a single data element to be transmitted by the TransmitPackets function.

Enumerations

RIO_NOTIFICATION_COMPLETION_TYPE

Specifies the type of completion queue notifications to use with the RIONotify function when sending or receiving data using the Winsock registered I/O extensions.

AcceptEx function (mswsock.h)

10/22/2022 • 10 minutes to read • Edit Online

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

> **Note** This function is a Microsoft-specific extension to the Windows Sockets specification.

Syntax

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

Parameters

`[in] sListenSocket`

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

`[in] sAcceptSocket`

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

`[in] lpOutputBuffer`

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

`[in] dwReceiveDataLength`

The number of bytes in *lpOutputBuffer* that will be used for actual receive data at the beginning of the buffer. This size should not include the size of the local address of the server, nor the remote address of the client; they are appended to the output buffer. If *dwReceiveDataLength* is zero, accepting the connection will not result in a receive operation. Instead, **AcceptEx** completes as soon as a connection arrives, without waiting for any data.

`[in] dwLocalAddressLength`

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

`[in] dwRemoteAddressLength`

The number of bytes reserved for the remote address information. This value must be at least 16 bytes more than the maximum address length for the transport protocol in use. Cannot be zero.

`[out] lpdwBytesReceived`

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

`[in] lpOverlapped`

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

Return value

If no error occurs, the **AcceptEx** function completed successfully and a value of **TRUE** is returned.

If the function fails, **AcceptEx** returns **FALSE**. The WSAGetLastError function can then be called to return extended error information. If **WSAGetLastError** returns **ERROR_IO_PENDING**, then the operation was successfully initiated and is still in progress. If the error is WSAECONNRESET, an incoming connection was indicated, but was subsequently terminated by the remote peer prior to accepting the call.

Remarks

The **AcceptEx** function combines several socket functions into a single API/kernel transition. The **AcceptEx** function, when successful, performs three tasks:

- A new connection is accepted.
- Both the local and remote addresses for the connection are returned.
- The first block of data sent by the remote is received.

> **Note** The function pointer for the **AcceptEx** function must be obtained at run time by making a call to the WSAIoctl function with the **SIO_GET_EXTENSION_FUNCTION_POINTER** opcode specified. The input buffer passed to the **WSAIoctl** function must contain **WSAID_ACCEPTEX**, a globally unique identifier (GUID) whose value identifies the **AcceptEx** extension function. On success, the output returned by the **WSAIoctl** function contains a pointer to the **AcceptEx** function. The **WSAID_ACCEPTEX** GUID is defined in the *Mswsock.h* header file.

A program can make a connection to a socket more quickly using **AcceptEx** instead of the accept function.

A single output buffer receives the data, the local socket address (the server), and the remote socket address (the client).

Using a single buffer improves performance. When using **AcceptEx**, the GetAcceptExSockaddrs function must be called to parse the buffer into its three distinct parts (data, local socket address, and remote socket address). On Windows XP and later, once the **AcceptEx** function completes and the **SO_UPDATE_ACCEPT_CONTEXT** option is set on the accepted socket, the local address associated with the accepted socket can also be retrieved using the getsockname function. Likewise, the remote address associated with the accepted socket can be retrieved using the getpeername function.

The buffer size for the local and remote address must be 16 bytes more than the size of the sockaddr structure for the transport protocol in use because the addresses are written in an internal format. For example, the size of a **sockaddr_in** (the address structure for TCP/IP) is 16 bytes. Therefore, a buffer size of at least 32 bytes must

be specified for the local and remote addresses.

The **AcceptEx** function uses overlapped I/O, unlike the accept function. If your application uses **AcceptEx**, it can service a large number of clients with a relatively small number of threads. As with all overlapped Windows functions, either Windows events or completion ports can be used as a completion notification mechanism.

Another key difference between the **AcceptEx** function and the accept function is that **AcceptEx** requires the caller to already have two sockets:

- One that specifies the socket on which to listen.
- One that specifies the socket on which to accept the connection.

The *sAcceptSocket* parameter must be an open socket that is neither bound nor connected.

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

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

ReadFile
WriteFile
send
WSASend
recv
WSARecv
TransmitFile
closesocket
setsockopt(only for SO_UPDATE_ACCEPT_CONTEXT)

> **Note** If the TransmitFile function is called with both the TF_DISCONNECT and TF_REUSE_SOCKET flags, the specified socket has been returned to a state in which it is neither bound nor connected. The socket handle can then be passed to the **AcceptEx** function in the *sAcceptSocket* parameter, but the socket cannot be passed to the ConnectEx function.

When the **AcceptEx** function returns, the socket *sAcceptSocket* is in the default state for a connected socket. The socket *sAcceptSocket* does not inherit the properties of the socket associated with *sListenSocket* parameter until SO_UPDATE_ACCEPT_CONTEXT is set on the socket. Use the setsockopt function to set the SO_UPDATE_ACCEPT_CONTEXT option, specifying *sAcceptSocket* as the socket handle and *sListenSocket* as the option value.

For example:

```
//Need to #include <mswsock.h> for SO_UPDATE_ACCEPT_CONTEXT

int iResult = 0;

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

If a receive buffer is provided, the overlapped operation will not complete until a connection is accepted and data is read. Use the getsockopt function with the SO_CONNECT_TIME option to check whether a connection has been accepted. If it has been accepted, you can determine how long the connection has been established. The return value is the number of seconds that the socket has been connected. If the socket is not connected, the **getsockopt** returns 0xFFFFFFFF. Applications that check whether the overlapped operation has completed, in

combination with the SO_CONNECT_TIME option, can determine that a connection has been accepted but no data has been received. Scrutinizing a connection in this manner enables an application to determine whether connections that have been established for a while have received no data. It is recommended such connections be terminated by closing the accepted socket, which forces the **AcceptEx** function call to complete with an error.

For example:

```
INT seconds;
INT bytes = sizeof(seconds);
int iResult = 0;

iResult = getsockopt( sAcceptSocket, SOL_SOCKET, SO_CONNECT_TIME,
                      (char *)&seconds, (PINT)&bytes );

if ( iResult != NO_ERROR ) {
    printf( "getsockopt(SO_CONNECT_TIME) failed: %u\n", WSAGetLastError( ) );
    exit(1);
}
```

Note All I/O initiated by a given thread is canceled when that thread exits. For overlapped sockets, pending asynchronous operations can fail if the thread is closed before the operations complete. See ExitThread for more information.

Windows Phone 8: This function is supported for Windows Phone Store apps on Windows Phone 8 and later.

Windows 8.1 and **Windows Server 2012 R2**: This function is supported for Windows Store apps on Windows 8.1, Windows Server 2012 R2, and later.

Example Code

The following example uses the **AcceptEx** function using overlapped I/O and completion ports.

```
#ifndef UNICODE
#define UNICODE
#endif

#define WIN32_LEAN_AND_MEAN

#include <winsock2.h>
#include <ws2tcpip.h>
#include <mswsock.h>
#include <stdio.h>

// Need to link with Ws2_32.lib
#pragma comment(lib, "Ws2_32.lib")

int main()
{
    //-----------------------------------------
    // Declare and initialize variables
    WSADATA wsaData;
    int iResult = 0;
    BOOL bRetVal = FALSE;

    HANDLE hCompPort;
    HANDLE hCompPort2;

    LPFN_ACCEPTEX lpfnAcceptEx = NULL;
    GUID GuidAcceptEx = WSAID_ACCEPTEX;
    WSAOVERLAPPED olOverlap;
```

```
    SOCKET ListenSocket = INVALID_SOCKET;
    SOCKET AcceptSocket = INVALID_SOCKET;
    sockaddr_in service;
    char lpOutputBuf[1024];
    int outBufLen = 1024;
    DWORD dwBytes;

    hostent *thisHost;
    char *ip;
    u_short port;

    // Initialize Winsock
    iResult = WSAStartup(MAKEWORD(2, 2), &wsaData);
    if (iResult != NO_ERROR) {
        wprintf(L"Error at WSAStartup\n");
        return 1;
    }

    // Create a handle for the completion port
    hCompPort = CreateIoCompletionPort(INVALID_HANDLE_VALUE, NULL, (u_long) 0, 0);
    if (hCompPort == NULL) {
        wprintf(L"CreateIoCompletionPort failed with error: %u\n",
            GetLastError() );
        WSACleanup();
        return 1;
    }

    // Create a listening socket
    ListenSocket = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
    if (ListenSocket == INVALID_SOCKET) {
        wprintf(L"Create of ListenSocket socket failed with error: %u\n",
            WSAGetLastError() );
        WSACleanup();
        return 1;
    }

    // Associate the listening socket with the completion port
    CreateIoCompletionPort((HANDLE) ListenSocket, hCompPort, (u_long) 0, 0);

    //-----------------------------------------
    // Bind the listening socket to the local IP address
    // and port 27015
    port = 27015;
    thisHost = gethostbyname("");
    ip = inet_ntoa(*(struct in_addr *) *thisHost->h_addr_list);

    service.sin_family = AF_INET;
    service.sin_addr.s_addr = inet_addr(ip);
    service.sin_port = htons(port);

    if (bind(ListenSocket, (SOCKADDR *) & service, sizeof (service)) == SOCKET_ERROR) {
        wprintf(L"bind failed with error: %u\n", WSAGetLastError());
        closesocket(ListenSocket);
        WSACleanup();
        return 1;
    }

    //-----------------------------------------
    // Start listening on the listening socket
    iResult = listen(ListenSocket, 100);
    if (iResult == SOCKET_ERROR) {
        wprintf(L"listen failed with error: %u\n", WSAGetLastError());
        closesocket(ListenSocket);
        WSACleanup();
        return 1;
    }

    wprintf(L"Listening on address: %s:%d\n", ip, port);
```

```
    // Load the AcceptEx function into memory using WSAIoctl.
    // The WSAIoctl function is an extension of the ioctlsocket()
    // function that can use overlapped I/O. The function's 3rd
    // through 6th parameters are input and output buffers where
    // we pass the pointer to our AcceptEx function. This is used
    // so that we can call the AcceptEx function directly, rather
    // than refer to the Mswsock.lib library.
    iResult = WSAIoctl(ListenSocket, SIO_GET_EXTENSION_FUNCTION_POINTER,
             &GuidAcceptEx, sizeof (GuidAcceptEx),
             &lpfnAcceptEx, sizeof (lpfnAcceptEx),
             &dwBytes, NULL, NULL);
    if (iResult == SOCKET_ERROR) {
        wprintf(L"WSAIoctl failed with error: %u\n", WSAGetLastError());
        closesocket(ListenSocket);
        WSACleanup();
        return 1;
    }

    // Create an accepting socket
    AcceptSocket = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
    if (AcceptSocket == INVALID_SOCKET) {
        wprintf(L"Create accept socket failed with error: %u\n", WSAGetLastError());
        closesocket(ListenSocket);
        WSACleanup();
        return 1;
    }

    // Empty our overlapped structure and accept connections.
    memset(&olOverlap, 0, sizeof (olOverlap));

    bRetVal = lpfnAcceptEx(ListenSocket, AcceptSocket, lpOutputBuf,
                 outBufLen - ((sizeof (sockaddr_in) + 16) * 2),
                 sizeof (sockaddr_in) + 16, sizeof (sockaddr_in) + 16,
                 &dwBytes, &olOverlap);
    if (bRetVal == FALSE) {
        wprintf(L"AcceptEx failed with error: %u\n", WSAGetLastError());
        closesocket(AcceptSocket);
        closesocket(ListenSocket);
        WSACleanup();
        return 1;
    }

    // Associate the accept socket with the completion port
    hCompPort2 = CreateIoCompletionPort((HANDLE) AcceptSocket, hCompPort, (u_long) 0, 0);
    // hCompPort2 should be hCompPort if this succeeds
    if (hCompPort2 == NULL) {
        wprintf(L"CreateIoCompletionPort associate failed with error: %u\n",
            GetLastError() );
        closesocket(AcceptSocket);
        closesocket(ListenSocket);
        WSACleanup();
        return 1;
    }

    // Continue on to use send, recv, TransmitFile(), etc.,.
    //...

    return 0;
}
```

Notes for QoS

The TransmitFile function allows the setting of two flags, TF_DISCONNECT or TF_REUSE_SOCKET, that return the socket to a "disconnected, reusable" state after the file has been transmitted. These flags should not be used on a socket where quality of service has been requested, since the service provider may immediately delete any quality

of service associated with the socket before the file transfer has completed. The best approach for a QoS-enabled socket is to simply call the closesocket function when the file transfer has completed, rather than relying on these flags.

Notes for ATM

There are important issues associated with connection setup when using Asynchronous Transfer Mode (ATM) with Windows Sockets 2. Please see the Remarks section in the accept function documentation for important ATM connection setup information.

Requirements

Minimum supported client	Windows 8.1, Windows Vista [desktop apps | UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps | UWP apps]
Target Platform	Windows
Header	mswsock.h (include Mswsock.h)
Library	Mswsock.lib
DLL	Mswsock.dll

See also

GetAcceptExSockaddrs

GetOverlappedResult

GetQueuedCompletionStatus

OVERLAPPED

TransmitFile

Winsock Functions

Winsock Reference

accept

closesocket

getsockopt

listen

sockaddr

GetAcceptExSockaddrs function (mswsock.h)

10/22/2022 • 2 minutes to read • Edit Online

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

> **Note** This function is a Microsoft-specific extension to the Windows Sockets specification.

Syntax

```
void GetAcceptExSockaddrs(
  [in]  PVOID    lpOutputBuffer,
  [in]  DWORD    dwReceiveDataLength,
  [in]  DWORD    dwLocalAddressLength,
  [in]  DWORD    dwRemoteAddressLength,
  [out] sockaddr **LocalSockaddr,
  [out] LPINT    LocalSockaddrLength,
  [out] sockaddr **RemoteSockaddr,
  [out] LPINT    RemoteSockaddrLength
);
```

Parameters

`[in] lpOutputBuffer`

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

`[in] dwReceiveDataLength`

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

`[in] dwLocalAddressLength`

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

`[in] dwRemoteAddressLength`

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

`[out] LocalSockaddr`

A pointer to the sockaddr structure that receives the local address of the connection (the same information that would be returned by the getsockname function). This parameter must be specified.

`[out] LocalSockaddrLength`

The size, in bytes, of the local address. This parameter must be specified.

`[out] RemoteSockaddr`

A pointer to the sockaddr structure that receives the remote address of the connection (the same information that would be returned by the getpeername function). This parameter must be specified.

`[out] RemoteSockaddrLength`

The size, in bytes, of the local address. This parameter must be specified.

Return value

None

Remarks

The **GetAcceptExSockaddrs** function is used exclusively with the AcceptEx function to parse the first data that the socket receives into local and remote addresses. The **AcceptEx** function returns local and remote address information in an internal format. Application developers need to use the **GetAcceptExSockaddrs** function if there is a need for the sockaddr structures containing the local or remote addresses.

> **Note** The function pointer for the **GetAcceptExSockaddrs** function must be obtained at run time by making a call to the WSAIoctl function with the **SIO_GET_EXTENSION_FUNCTION_POINTER** opcode specified. The input buffer passed to the **WSAIoctl** function must contain **WSAID_GETACCEPTEXSOCKADDRS**, a globally unique identifier (GUID) whose value identifies the **GetAcceptExSockaddrs** extension function. On success, the output returned by the **WSAIoctl** function contains a pointer to the **GetAcceptExSockaddrs** function. The **WSAID_GETACCEPTEXSOCKADDRS** GUID is defined in the *Mswsock.h* header file.

Windows Phone 8: This function is supported for Windows Phone Store apps on Windows Phone 8 and later.

Windows 8.1 and **Windows Server 2012 R2**: This function is supported for Windows Store apps on Windows 8.1, Windows Server 2012 R2, and later.

Requirements

Minimum supported client	Windows 8.1, Windows Vista [desktop apps | UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps | UWP apps]
Target Platform	Windows
Header	mswsock.h (include Mswsock.h)
Library	Mswsock.lib
DLL	Mswsock.dll

See also

AcceptEx

Winsock Functions

Winsock Reference

getpeername

getsockname

sockaddr

LPFN_CONNECTEX callback function (mswsock.h)

10/22/2022 • 9 minutes to read • Edit Online

The **ConnectEx** function establishes a connection to a specified socket, and optionally sends data once the connection is established. The **ConnectEx** function is only supported on connection-oriented sockets.

> **Note** This function is a Microsoft-specific extension to the Windows Sockets specification.

Syntax

```
LPFN_CONNECTEX LpfnConnectex;

BOOL LpfnConnectex(
  [in]           SOCKET s,
  [in]           const sockaddr *name,
  [in]           int namelen,
  [in, optional] PVOID lpSendBuffer,
  [in]           DWORD dwSendDataLength,
  [out]          LPDWORD lpdwBytesSent,
  [in]           LPOVERLAPPED lpOverlapped
)
{...}
```

Parameters

`[in] s`

A descriptor that identifies an unconnected, previously bound socket. See Remarks for more information.

`[in] name`

A pointer to
a sockaddr structure that specifies the address to which to connect. For IPv4, the **sockaddr** contains **AF_INET** for the address family, the destination IPv4 address, and the destination port. For IPv6, the **sockaddr** structure contains **AF_INET6** for the address family, the destination IPv6 address, the destination port, and may contain additional IPv6 flow and scope-id information.

`[in] namelen`

The length, in bytes, of the sockaddr structure pointed to by the *name* parameter.

`[in, optional] lpSendBuffer`

A pointer to the buffer to be transferred after a connection is established. This parameter is optional. If the TCP_FASTOPEN option is enabled on *s* before **ConnectEx** is called, then some of this data may be sent during connection establishment.

`[in] dwSendDataLength`

The length, in bytes, of data pointed to by the *lpSendBuffer* parameter. This parameter is ignored when the *lpSendBuffer* parameter is **NULL**.

`[out] lpdwBytesSent`

On successful return, this parameter points to a **DWORD** value that indicates the number of bytes that were sent after the connection was established. The bytes sent are from the buffer pointed to by the *lpSendBuffer* parameter. This parameter is ignored when the *lpSendBuffer* parameter is **NULL**.

`[in] lpOverlapped`

An OVERLAPPED structure used to process the request. The *lpOverlapped* parameter must be specified, and cannot be **NULL**.

Return value

On success, the **ConnectEx** function returns **TRUE**. On failure, the function returns **FALSE**. Use the WSAGetLastError function to get extended error information. If a call to the **WSAGetLastError** function returns **ERROR_IO_PENDING**, the operation initiated successfully and is in progress. Under such circumstances, the call may still fail when the overlapped operation completes.

If the error code returned is WSAECONNREFUSED, WSAENETUNREACH, or WSAETIMEDOUT, the application can call **ConnectEx**, WSAConnect, or connect again on the same socket.

ERROR CODE	DESCRIPTION
WSANOTINITIALISED	A successful WSAStartup function call must occur before using ConnectEx.
WSAENETDOWN	The network subsystem has failed.
WSAEADDRINUSE	The local address of the socket is already in use, and the socket was not marked to allow address reuse with SO_REUSEADDR. This error usually occurs during a bind operation, but the error could be delayed until a ConnectEx function call, if the **bind** function was called with a wildcard address (**INADDR_ANY** or **in6addr_any**) specified for the local IP address. A specific IP address needs to be implicitly bound by the **ConnectEx** function.
WSAEALREADY	A nonblocking connect, WSAConnect, or ConnectEx function call is in progress on the specified socket.
WSAEADDRNOTAVAIL	The remote address is not a valid address, such as ADDR_ANY (the ConnectEx function is only supported for connection-oriented sockets).
WSAEAFNOSUPPORT	Addresses in the specified family cannot be used with this socket.
WSAECONNREFUSED	The attempt to connect was rejected.
WSAEFAULT	The *name*, *lpSendBuffer*, or *lpOverlapped* parameter is not a valid part of the user address space, or *namelen* is too small.

WSAEINVAL	The parameter *s* is an unbound or a listening socket.
WSAEISCONN	The socket is already connected.
WSAENETUNREACH	The network cannot be reached from this host at this time.
WSAEHOSTUNREACH	A socket operation was attempted to an unreachable host.
WSAENOBUFS	No buffer space is available; the socket cannot be connected.
WSAENOTSOCK	The descriptor is not a socket.
WSAETIMEDOUT	The attempt to connect timed out without establishing a connection.

Remarks

The **ConnectEx** function combines several socket functions into a single API/kernel transition. The following operations are performed when a call to the **ConnectEx** function completes successfully:

- A new connection is established.
- An optional block of data is sent after the connection is established.

For applications targeted to Windows Vista and later, consider using the WSAConnectByList or WSAConnectByName function which greatly simplify client application design.

The **ConnectEx** function can only be used with connection-oriented sockets. The socket passed in the *s* parameter must be created with a socket type of **SOCK_STREAM**, **SOCK_RDM**, or **SOCK_SEQPACKET**.

The *lpSendBuffer* parameter points to a buffer of data to send after the connection is established. The *dwSendDataLength* parameter specifies the length in bytes of this data to send. An application can request to send a large buffer of data using the **ConnectEx** in the same way that the send and WSASend functions can be used. But developers are strongly advised against sending a huge buffer in a single call using **ConnectEx**, because this operation uses a large amount of system memory resources until the whole buffer has been sent.

If the **ConnectEx** function is successful, a connection was established and all of the data pointed to by the *lpSendBuffer* parameter was sent to the address specified in the **sockaddr** structure pointed to by the *name* parameter.

> **Note** The function pointer for the **ConnectEx** function must be obtained at run time by making a call to the WSAIoctl function with the **SIO_GET_EXTENSION_FUNCTION_POINTER** opcode specified. The input buffer passed to the **WSAIoctl** function must contain **WSAID_CONNECTEX**, a globally unique identifier (GUID) whose value identifies the **ConnectEx** extension function. On success, the output returned by the **WSAIoctl** function contains a pointer to the **ConnectEx** function. The **WSAID_CONNECTEX** GUID is defined in the *Mswsock.h* header file.

The **ConnectEx** function uses overlapped I/O. As a result, the **ConnectEx** function enables an application to service a large number of clients with relatively few threads. In contrast, the WSAConnect function, which does not use overlapped I/O, usually requires a separate thread to service each connection request when simultaneous requests are received.

> **Note** All I/O initiated by a given thread is canceled when that thread exits. For overlapped sockets, pending asynchronous operations can fail if the thread is closed before the operations complete. See ExitThread for more information.

Connection-oriented sockets are often unable to complete their connection immediately, and therefore the operation is initiated and the function immediately returns with the ERROR_IO_PENDING or WSA_IO_PENDING error. When the connect operation completes and success or failure is achieved, status is reported using the completion notification mechanism indicated in *lpOverlapped*. As with all overlapped function calls, you can use events or completion ports as the completion notification mechanism. The *lpNumberOfBytesTransferred* parameter of the GetQueuedCompletionStatus or GetOverlappedResult or WSAGetOverlappedResult function indicates the number of bytes sent in the request.

When the **ConnectEx** function successfully completes, socket handle *s* can be passed to only the following functions:

- ReadFile
- WriteFile
- send or WSASend
- recv or WSARecv
- TransmitFile
- closesocket

If the TransmitFile function is called on a previously connected socket with both TF_DISCONNECT and TF_REUSE_SOCKET flags, the specified socket is returned to a state in which it is not connected, but still bound. In such cases, the handle of the socket can be passed to the **ConnectEx** function in its *s* parameter, but the socket cannot be reused in an AcceptEx function call. Similarly, the accepted socket reused using the **TransmitFile** function cannot be used in a call to **ConnectEx**. Note that in the case of a reused socket, **ConnectEx** is subject to the behavior of the underlying transport. For example, a TCP socket may be subject to the TCP TIME_WAIT state, causing the **ConnectEx** call to be delayed.

When the **ConnectEx** function returns **TRUE**, the socket *s* is in the default state for a connected socket. The socket *s* does not enable previously set properties or options until SO_UPDATE_CONNECT_CONTEXT is set on the socket. Use the setsockopt function to set the SO_UPDATE_CONNECT_CONTEXT option.

For example:

```
//Need to #include <mswsock.h> for SO_UPDATE_CONNECT_CONTEXT

int iResult = 0;

iResult = setsockopt( s, SOL_SOCKET, SO_UPDATE_CONNECT_CONTEXT, NULL, 0 );
```

The getsockopt function can be used with the **SO_CONNECT_TIME** socket option to check whether a connection has been established while **ConnectEx** is in progress. If a connection has been established, the value returned in the *optval* parameter passed to the **getsockopt** function is the number of seconds the socket has been connected. If the socket is not connected, the returned *optval* parameter contains 0xFFFFFFFF. Checking a connection in this manner is necessary to determine whether connections have been established for a period of time without sending any data; in such cases, it is recommended that such connections be terminated.

For example:

```
//Need to #include <mswsock.h> for SO_CONNECT_TIME

int seconds;
int bytes = sizeof(seconds);
int iResult = 0;

iResult = getsockopt( s, SOL_SOCKET, SO_CONNECT_TIME,
                      (char *)&seconds, (PINT)&bytes );
if ( iResult != NO_ERROR ) {
    printf( "getsockopt(SO_CONNECT_TIME) failed with error: %u\n",
        WSAGetLastError() );
}
else {
    if (seconds == 0xFFFFFFFF)
        printf("Connection not established yet\n");
    else
       printf("Connection has been established %ld seconds\n",
           seconds);
}
```

> **Note** If a socket is opened, a setsockopt call is made, and then a sendto call is made, Windows Sockets performs an implicit bind function call.

If the address parameter of the sockaddr structure pointed to in the *name* parameter is all zeros, **ConnectEx** returns the error WSAEADDRNOTAVAIL. Any attempt to reconnect an active connection will fail with the error code WSAEISCONN.

When a connected socket becomes closed for any reason, it is recommended that the socket be discarded and a new socket created. The reasoning for this is that it is safest to assume that when things go awry on a connected socket for any reason, the application must discard the socket and create the needed socket again in order to return to a stable point.

If the DisconnectEx function is called with the **TF_REUSE_SOCKET** flag, the specified socket is returned to a state in which it is not connected, but still bound. In such cases, the handle of the socket can be passed to the **ConnectEx** function in its *s* parameter.

The interval of time that must elapse before TCP can release a closed connection and reuse its resources is known as the TIME_WAIT state or the 2MSL state. During this time, the connection can be reopened at much less cost to the client and server than establishing a new connection.

The TIME_WAIT behavior is specified in RFC 793, which requires that TCP maintains a closed connection for an interval at least equal to twice the maximum segment lifetime (MSL) of the network. When a connection is released, its socket pair and internal resources used for the socket can be used to support another connection.

Windows TCP reverts to a TIME_WAIT state subsequent to the closing of a connection. While in the TIME_WAIT state, a socket pair cannot be reused. The TIME_WAIT period is configurable by modifying the following **DWORD** registry setting that represents the TIME_WAIT period in seconds.

HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services\TCPIP\Parameters\TcpTimedWaitDelay

By default, the MSL is defined to be 120 seconds. The TcpTimedWaitDelay registry setting defaults to a value 240 seconds, which represents 2 times the maximum segment lifetime of 120 seconds or 4 minutes. However, you can use this entry to customize the interval.

Reducing the value of this entry allows TCP to release closed connections faster, providing more resources for new connections. However, if the value is too low, TCP might release connection resources before the connection is complete, requiring the server to use additional resources to re-establish the connection.

This registry setting can be set from 0 to 300 seconds.

Windows Phone 8: This function is supported for Windows Phone Store apps on Windows Phone 8 and later.

Windows 8.1 and **Windows Server 2012 R2**: This function is supported for Windows Store apps on Windows 8.1, Windows Server 2012 R2, and later.

Requirements

Minimum supported client	Windows 8.1, Windows Vista [desktop apps | UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps | UWP apps]
Target Platform	Windows
Header	mswsock.h

See also

AcceptEx

DisconnectEx

ExitThread

GetOverlappedResult

GetQueuedCompletionStatus

OVERLAPPED

ReadFile

TransmitFile

WSAConnect

WSAConnectByList

WSAConnectByName

WSAGetLastError

WSAIoctl

WSARecv

WSASend

WSAStartup

Winsock Functions

Winsock Reference

WriteFile

bind

closesocket

connect

getsockopt

recv

send

sendto

setsockopt

sockaddr

LPFN_DISCONNECTEX callback function (mswsock.h)

10/22/2022 • 4 minutes to read • Edit Online

The **DisconnectEx** function closes a connection on a socket, and allows the socket handle to be reused.

> **NOTE**
>
> This function is a Microsoft-specific extension to the Windows Sockets specification.

Syntax

```
LPFN_DISCONNECTEX LpfnDisconnectex;

BOOL LpfnDisconnectex(
  SOCKET s,
  LPOVERLAPPED lpOverlapped,
  DWORD dwFlags,
  DWORD dwReserved
)
{...}
```

Parameters

`s`

A handle to a connected, connection-oriented socket.

`lpOverlapped`

A pointer to an OVERLAPPED structure. If the socket handle has been opened as overlapped, specifying this parameter results in an overlapped (asynchronous) I/O operation.

`dwFlags`

A set of flags that customizes processing of the function call. When this parameter is set to zero, no flags are set. The *dwFlags* parameter can have the following value.

FLAG	MEANING

FLAG	MEANING
TF_REUSE_SOCKET	Prepares the socket handle to be reused. When the **DisconnectEx** request completes, the socket handle can be passed to the [**AcceptEx**](./nf-mswsock-acceptex.md) or [**ConnectEx**](./nc-mswsock-lpfn_connectex.md) function. [!Note] The socket level disconnect is subject to the behavior of the underlying transport. For example, a TCP socket may be subject to the TCP TIME_WAIT state, causing the **DisconnectEx** call to be delayed.

`dwReserved`

Reserved. Must be zero. If nonzero, WSAEINVAL is returned.

Return value

On success, the **DisconnectEx** function returns **TRUE**. On failure, the function returns **FALSE**. Use the **WSAGetLastError** function to get extended error information. If a call to the **WSAGetLastError** function returns **ERROR_IO_PENDING**, the operation initiated successfully and is in progress. Under such circumstances, the call may still fail when the operation completes.

ERROR CODE	DESCRIPTION
WSAEFAULT	The system detected an invalid pointer address in attempting to use a pointer argument. This error is returned if an invalid pointer value was passed in the *lpOverlapped* parameter.
WSAEINVAL	The invalid parameter was passed. This error is returned if the *dwFlags* parameter was specified with a zero value other than **TF_REUSE_SOCKET**.
WSAENOTCONN	The socket is not connected. This error is returned if the socket *s* parameter was not in a connected state. This error can also be returned if the socket was in the transmit closing state from a previous request and the *dwFlags* parameter was not set to **TF_REUSE_SOCKET** to request a reuse of the socket.

Remarks

The **DisconnectEx** function does not support datagram sockets. Therefore, the socket specified by *hSocket* must be connection-oriented, such as a SOCK_STREAM, SOCK_SEQPACKET, or SOCK_RDM socket.

> **NOTE**
> The function pointer for the **DisconnectEx** function must be obtained at run time by making a call to the **WSAIoctl** function with the **SIO_GET_EXTENSION_FUNCTION_POINTER** opcode specified. The input buffer passed to the **WSAIoctl** function must contain **WSAID_DISCONNECTEX**, a globally unique identifier (GUID) whose value identifies the **DisconnectEx** extension function. On success, the output returned by the **WSAIoctl** function contains a pointer to the **DisconnectEx** function. The **WSAID_DISCONNECTEX** GUID is defined in the *Mswsock.h* header file.

When *lpOverlapped* is not **NULL**, overlapped I/O might not finish before **DisconnectEx** returns, resulting in the **DisconnectEx** function returning **FALSE** and a call to the **WSAGetLastError** function returning **ERROR_IO_PENDING**. This design enables the caller to continue processing while the disconnect operation completes. Upon completion of the request, Windows sets either the event specified by the **hEvent** member of the **OVERLAPPED** structure, or the socket specified by *hSocket*, to the signaled state.

> **NOTE**
> All I/O initiated by a given thread is canceled when that thread exits. For overlapped sockets, pending asynchronous operations can fail if the thread is closed before the operations complete. See **ExitThread** for more information.

The TIME_WAIT state determines the time that must elapse before TCP can release a closed connection and reuse its resources. This interval between closure and release is known as the TIME_WAIT state or 2MSL state. During this time, the connection can be reopened at much less cost to the client and server than establishing a new connection. The TIME_WAIT behavior is specified in RFC 793 which requires that TCP maintains a closed connection for an interval at least equal to twice the maximum segment lifetime (MSL) of the network. When a connection is released, its socket pair and internal resources used for the socket can be used to support another connection.

Windows TCP reverts to a TIME_WAIT state subsequent to the closing of a connection. While in the TIME_WAIT state, a socket pair cannot be re-used. The TIME_WAIT period is configurable by modifying the following DWORD registry setting that represents the TIME_WAIT period in seconds.

HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services\TCPIP\Parameters\TcpTimedWaitDelay

By default, the MSL is defined to be 120 seconds. The TcpTimedWaitDelay registry setting defaults to a value 240 seconds, which represents 2 times the maximum segment lifetime of 120 seconds or 4 minutes. However, you can use this entry to customize the interval. Reducing the value of this entry allows TCP to release closed connections faster, providing more resources for new connections. However, if the value is too low, TCP might release connection resources before the connection is complete, requiring the server to use additional resources to re-establish the connection. This registry setting can be set from 0 to 300 seconds.

Windows Phone 8: This function is supported for Windows Phone Store apps on Windows Phone 8 and later.

Windows 8.1 and **Windows Server 2012 R2**: This function is supported for Windows Store apps on Windows 8.1, Windows Server 2012 R2, and later.

Requirements

Header	mswsock.h

LPFN_RIOCLOSECOMPLETIONQUEUE callback function (mswsock.h)

10/22/2022 • 2 minutes to read • Edit Online

The **RIOCloseCompletionQueue** function closes an existing completion queue used for I/O completion notification by send and receive requests with the Winsock registered I/O extensions.

Syntax

```
LPFN_RIOCLOSECOMPLETIONQUEUE LpfnRioclosecompletionqueue;

void LpfnRioclosecompletionqueue(
  RIO_CQ CQ
)
{...}
```

Parameters

`CQ`

A descriptor identifying an existing completion queue.

Return value

None

Remarks

The **RIOCloseCompletionQueue** function closes an existing completion queue used for I/O completion. The RIO_CQ passed in the *CQ* parameter is locked for writing by the kernel. The completion queue is marked as invalid, so that new completions cannot be added. Any new completions to be added are silently dropped. The application is expected to tracking any pending send or receive operations.

If an invalid completion queue is passed in the *CQ* parameter (**RIO_INVALID_CQ**, for example), this is ignored by the **RIOCloseCompletionQueue** function.

> **NOTE**
>
> The function pointer to the **RIOCloseCompletionQueue** function must be obtained at run time by making a call to the WSAIoctl function with the **SIO_GET_MULTIPLE_EXTENSION_FUNCTION_POINTER** opcode specified. The input buffer passed to the **WSAIoctl** function must contain **WSAID_MULTIPLE_RIO**, a globally unique identifier (GUID) whose value identifies the Winsock registered I/O extension functions. On success, the output returned by the **WSAIoctl** function contains a pointer to the RIO_EXTENSION_FUNCTION_TABLE structure that contains pointers to the Winsock registered I/O extension functions. The **SIO_GET_MULTIPLE_EXTENSION_FUNCTION_POINTER** IOCTL is defined in the *Ws2def.h* header file. The **WSAID_MULTIPLE_RIO** GUID is defined in the *Mswsock.h* header file.

Windows Phone 8: This function is supported for Windows Phone Store apps on Windows Phone 8 and later.

Windows 8.1 and **Windows Server 2012 R2**: This function is supported for Windows Store apps on Windows 8.1, Windows Server 2012 R2, and later.

Requirements

Header	mswsock.h

LPFN_RIOCREATECOMPLETIONQUEUE callback function (mswsock.h)

10/22/2022 • 5 minutes to read • Edit Online

The **RIOCreateCompletionQueue** function creates an I/O completion queue of a specific size for use with the Winsock registered I/O extensions.

Syntax

```
LPFN_RIOCREATECOMPLETIONQUEUE LpfnRiocreatecompletionqueue;

RIO_CQ LpfnRiocreatecompletionqueue(
  DWORD QueueSize,
  PRIO_NOTIFICATION_COMPLETION NotificationCompletion
)
{...}
```

Parameters

`QueueSize`

The size, in number of entries, of the completion queue to create.

`NotificationCompletion`

The type of notification completion to use based on the **Type** member of the RIO_NOTIFICATION_COMPLETION structure (I/O completion or event notification).

If the **Type** member is set to **RIO_EVENT_COMPLETION**, then the **Event** member of the RIO_NOTIFICATION_COMPLETION structure must be set.

If the **Type** member is set to **RIO_IOCP_COMPLETION**, then the **Iocp** member of the RIO_NOTIFICATION_COMPLETION structure must be set and the **Iocp.Overlapped** member of the **RIO_NOTIFICATION_COMPLETION** structure must not be NULL.

If the *NotificationCompletion* parameter is NULL, this specifies no notification completion is used and that polling must be used to determine completion.

Return value

If no error occurs, the **RIOCreateCompletionQueue** function returns a descriptor referencing a new completion queue. Otherwise, a value of **RIO_INVALID_CQ** is returned, and a specific error code can be retrieved by calling the WSAGetLastError function.

RETURN CODE	DESCRIPTION
WSAEFAULT	The system detected an invalid pointer address in attempting to use a pointer argument in a call.

RETURN CODE	DESCRIPTION
WSAEINVAL	An invalid parameter was passed to the function. This error is returned if the *QueueSize* parameter is less than 1 or greater than **RIO_MAX_CQ_SIZE** defined in the *Mswsockdef.h* header file.
WSAENOBUFS	Sufficient memory could not be allocated. This error is returned if there was insufficient memory to allocate the completion queue requested based on the *QueueSize* parameter.

Remarks

The **RIOCreateCompletionQueue** function creates an I/O completion queue of a specific size. The size of the completion queue restricts the set of registered I/O sockets that can be associated with the completion queue. For more information, see the **RIOCreateRequestQueue** function.

When creating a **RIO_CQ**, the **RIO_NOTIFICATION_COMPLETION** structure pointed to by the *NotificationCompletion* parameter determines how the application will receive completion queue notifications. If a **RIO_NOTIFICATION_COMPLETION** structure is provided when creating the completion queue, the application may call the **RIONotify** function to request a completion queue notification. Normally this notification occurs when the completion queue is not empty. This may happen immediately or when the next completion entry is inserted into the completion queue. However, send and receive requests may be flagged as **RIO_MSG_DONT_NOTIFY**. Completion queue notification and will never be triggered as a result of such requests. If the completion queue contains only entries with the **RIO_MSG_DONT_NOTIFY** flag set, the completion queue notification will not be triggered. Also, when a new entry enters the completion queue, the completion queue notification is only triggered if the **RIO_MSG_DONT_NOTIFY** flag was not set on the associated request. Any completed requests can still be retrieved by polling using the **RIODequeueCompletion** function. Once a completion queue notification is issued, the application must call the **RIONotify** function in order to receive another completion queue notification. When a completion queue notification occurs, the application typically calls the **RIODequeueCompletion** function to dequeue the completed send or receive requests.

Two options are available for completion queue notification.

- Event handles.
- I/O completion ports

If the **Type** member of the **RIO_NOTIFICATION_COMPLETION** structure is set to **RIO_EVENT_COMPLETION**, an event handle is used to signal completion queue notifications. An event handle is provided as the **EventNotify.EventHandle** member in the **RIO_NOTIFICATION_COMPLETION** structure passed to the **RIOCreateCompletionQueue** function. 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. The completion of the **RIONotify** function for this **RIO_CQ** will signal the event. The **Event.NotifyReset** member in the **RIO_NOTIFICATION_COMPLETION** structure passed to the **RIOCreateCompletionQueue** function indicates whether or not the event should be reset as part of a call to the **RIONotify** function. 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.

If the **Type** member of the **RIO_NOTIFICATION_COMPLETION** structure is set to **RIO_IOCP_COMPLETION**, an I/O completion port is used to signal completion queue notifications. An I/O completion port handle is

provided as the **Iocp.IocpHandle** member in the **RIO_NOTIFICATION_COMPLETION** structure passed to the **RIOCreateCompletionQueue** function. The completion of the **RIONotify** function for this **RIO_CQ** will queue an entry to the I/O completion port which can be retrieved using the **GetQueuedCompletionStatus** or **GetQueuedCompletionStatusEx** function. A queued entry will have the returned *lpCompletionKey* parameter value set to the value specified in **Iocp.CompletionKey** member of the **RIO_NOTIFICATION_COMPLETION** structure and the **Iocp.Overlapped** member in the **RIO_NOTIFICATION_COMPLETION** structure will be a non-NULL value.

In terms of its usage, completion queue notification is designed to wake up a waiting application thread so that the thread can examine the completion queue. Waking and scheduling a thread comes at a cost, so if this happens too frequently it will have a negative impact on the application performance. The **RIO_MSG_DONT_NOTIFY** flag is provided so that the application can control the frequency of these events and limit their over impact on performance.

> **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.

> **NOTE**
>
> The function pointer to the **RIOCreateCompletionQueue** function must be obtained at run time by making a call to the **WSAIoctl** function with the **SIO_GET_MULTIPLE_EXTENSION_FUNCTION_POINTER** opcode specified. The input buffer passed to the **WSAIoctl** function must contain **WSAID_MULTIPLE_RIO**, a globally unique identifier (GUID) whose value identifies the Winsock registered I/O extension functions. On success, the output returned by the **WSAIoctl** function contains a pointer to the **RIO_EXTENSION_FUNCTION_TABLE** structure that contains pointers to the Winsock registered I/O extension functions. The **SIO_GET_MULTIPLE_EXTENSION_FUNCTION_POINTER** IOCTL is defined in the *Ws2def.h* header file. The **WSAID_MULTIPLE_RIO** GUID is defined in the *Mswsock.h* header file.

Windows Phone 8: This function is supported for Windows Phone Store apps on Windows Phone 8 and later.

Windows 8.1 and **Windows Server 2012 R2**: This function is supported for Windows Store apps on Windows 8.1, Windows Server 2012 R2, and later.

Requirements

Header	mswsock.h

LPFN_RIOCREATEREQUESTQUEUE callback function (mswsock.h)

10/22/2022 • 4 minutes to read • Edit Online

The **RIOCreateRequestQueue** function 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.

Syntax

```
LPFN_RIOCREATEREQUESTQUEUE LpfnRiocreaterequestqueue;

RIO_RQ LpfnRiocreaterequestqueue(
  SOCKET Socket,
  ULONG MaxOutstandingReceive,
  ULONG MaxReceiveDataBuffers,
  ULONG MaxOutstandingSend,
  ULONG MaxSendDataBuffers,
  RIO_CQ ReceiveCQ,
  RIO_CQ SendCQ,
  PVOID SocketContext
)
{...}
```

Parameters

`Socket`

A descriptor that identifies the socket.

`MaxOutstandingReceive`

The maximum number of outstanding receives allowed on the socket.

This parameter is usually a small number for most applications.

`MaxReceiveDataBuffers`

The maximum number of receive data buffers on the socket.

> **NOTE**
>
> For Windows 8 and Windows Server 2012 , this parameter must be **1**.

`MaxOutstandingSend`

The maximum number of outstanding sends allowed on the socket.

`MaxSendDataBuffers`

The maximum number of send data buffers on the socket.

> **NOTE**
> For Windows 8 and Windows Server 2012 , this parameter must be **1**.

`ReceiveCQ`

A descriptor that identifies the I/O completion queue to use for receive request completions.

`SendCQ`

A descriptor that identifies the I/O completion queue to use for send request completions.

This parameter may have the same value as the *ReceiveCQ* parameter.

`SocketContext`

The socket context to associate with this request queue.

Return value

If no error occurs, the **RIOCreateRequestQueue** function returns a descriptor referencing a new request queue. Otherwise, a value of **RIO_INVALID_RQ** is returned, and a specific error code can be retrieved by calling the WSAGetLastError function.

RETURN CODE	DESCRIPTION
WSAEINVAL	An invalid parameter was passed to the function. This error is returned if the *ReceiveCQ* or *SendCQ* parameters contained **RIO_INVALID_CQ**. This error is returned if both the *MaxOutstandingReceive* and *MaxOutstandingSend* parameters are zero. This error is also returned if the socket passed in the *Socket* parameter is in the process of initializing or closing.
WSAENOBUFS	Sufficient memory could not be allocated. This error is returned if there was insufficient memory to allocate the request queue based on the parameters. This error is also returned if the network session limit was exceeded.
WSAENOTSOCK	The descriptor is not a socket. This error is returned if the *Socket* parameter is not a valid socket.
WSAEOPNOTSUPP	The attempted operation is not supported for the type of object referenced. This error is returned for a socket in the *Socket* parameter for an unsupported socket type (**SOCK_RAW**, for example)

Remarks

The **RIOCreateRequestQueue** function creates a registered I/O socket descriptor using a specified socket and I/O completion queues. An application must call **RIOCreateRequestQueue** to obtain a RIO_RQ for a Winsock socket before the application can use the RIOSend, RIOSendEx, RIOReceive, or RIOReceiveEx functions. In order to obtain a **RIO_RQ**, the Winsock socket must be associated with completion queues for send and receive, although the same completion queue can be used for both.

Due to the finite size of completion queues, a socket may only be associated with a completion queue for send and receive operations if it guarantees not to exceed the capacity for total queued completions. Therefore, socket

specific limits are established by the call to the **RIOCreateRequestQueue** function. These limits are used both during the **RIOCreateRequestQueue** call to verify sufficient space in the completion queues to accommodate the socket requests and during request initiation time to make sure that the request does not cause the socket to exceed its limits.

The send and receive queues can be associated with multiple sockets. The sizes of the send and receive queues must be greater than or equal to the send and receive sizes of all attached sockets. As request queues are closed by closing the sockets using the the **closesocket** function, those slots will be freed up for use by other sockets.

> **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.

When an application is finished using the **RIO_RQ**, the application should call the **closesocket** function to close the socket and free the associated resources.

> **NOTE**
>
> The function pointer to the **RIOCreateRequestQueue** function must be obtained at run time by making a call to the **WSAIoctl** function with the **SIO_GET_MULTIPLE_EXTENSION_FUNCTION_POINTER** opcode specified. The input buffer passed to the **WSAIoctl** function must contain **WSAID_MULTIPLE_RIO**, a globally unique identifier (GUID) whose value identifies the Winsock registered I/O extension functions. On success, the output returned by the **WSAIoctl** function contains a pointer to the **RIO_EXTENSION_FUNCTION_TABLE** structure that contains pointers to the Winsock registered I/O extension functions. The **SIO_GET_MULTIPLE_EXTENSION_FUNCTION_POINTER** IOCTL is defined in the *Ws2def.h* header file. The **WSAID_MULTIPLE_RIO** GUID is defined in the *Mswsock.h* header file.

Windows Phone 8: This function is supported for Windows Phone Store apps on Windows Phone 8 and later.

Windows 8.1 and **Windows Server 2012 R2**: This function is supported for Windows Store apps on Windows 8.1, Windows Server 2012 R2, and later.

Requirements

Header	mswsock.h

LPFN_RIODEQUEUECOMPLETION callback function (mswsock.h)

10/22/2022 • 3 minutes to read • Edit Online

The **RIODequeueCompletion** function removes entries from an I/O completion queue for use with the Winsock registered I/O extensions.

Syntax

```
LPFN_RIODEQUEUECOMPLETION LpfnRiodequeuecompletion;

ULONG LpfnRiodequeuecompletion(
  RIO_CQ CQ,
  PRIORESULT Array,
  ULONG ArraySize
)
{...}
```

Parameters

`CQ`

A descriptor that identifies an I/O completion queue.

`Array`

An array of RIORESULT structures to receive the description of the completions dequeued.

`ArraySize`

The maximum number of entries in the *Array* to write.

Return value

If no error occurs, the **RIODequeueCompletion** function returns the number of completion entries removed from the specified completion queue. Otherwise, a value of **RIO_CORRUPT_CQ** is returned to indicate that the state of the RIO_CQ passed in the *CQ* parameter has become corrupt due to memory corruption or misuse of the RIO functions.

Remarks

The **RIODequeueCompletion** function removes entries from an I/O completion queue for send and receive requests with the Winsock registered I/O extensions.

The **RIODequeueCompletion** function is the mechanism by which an application can find out about completed send and receive requests. An application normally calls the **RIODequeueCompletion** function after receiving notification based on the method registered with the RIONotify function when the completion queue is not empty. The notification behavior for an I/O completion queue is set when the RIO_CQ is created. The RIO_NOTIFICATION_COMPLETION structure that determines the notification behavior is passed to the RIOCreateCompletionQueue function when a **RIO_CQ** is created.

When the **RIODequeueCompletion** function completes, the *Array* parameter contains an array of pointers to

RIORESULT structures for the completed send and receive requests that were dequeued. The members of the returned **RIORESULT** structures provide information on the completion status of the completed request and the number of bytes that were transferred. Each returned **RIORESULT** structure also includes a socket context and an application context that can be used to identify the specific completed request.

If the I/O completion queue passed in the *CQ* parameter is not valid or damaged, the **RIODequeueCompletion** function returns a count of **RIO_CORRUPT_CQ**.

The **RIODequeueCompletion** function returns a value of zero is returned if there are no completed send or receive requests to be dequeued.

Only after a request's completion has been dequeued does the system release the association to its buffer and buffer registration, along with its quota charge.

> **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.

> **NOTE**
> The function pointer to the **RIODequeueCompletion** function must be obtained at run time by making a call to the WSAIoctl function with the **SIO_GET_MULTIPLE_EXTENSION_FUNCTION_POINTER** opcode specified. The input buffer passed to the **WSAIoctl** function must contain **WSAID_MULTIPLE_RIO**, a globally unique identifier (GUID) whose value identifies the Winsock registered I/O extension functions. On success, the output returned by the **WSAIoctl** function contains a pointer to the RIO_EXTENSION_FUNCTION_TABLE structure that contains pointers to the Winsock registered I/O extension functions. The **SIO_GET_MULTIPLE_EXTENSION_FUNCTION_POINTER** IOCTL is defined in the *Ws2def.h* header file. The **WSAID_MULTIPLE_RIO** GUID is defined in the *Mswsock.h* header file.

Windows Phone 8: This function is supported for Windows Phone Store apps on Windows Phone 8 and later.

Windows 8.1 and **Windows Server 2012 R2**: This function is supported for Windows Store apps on Windows 8.1, Windows Server 2012 R2, and later.

Thread Safety

If multiple threads attempt to access the same RIO_CQ using the **RIODequeueCompletion** function, 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

Header	mswsock.h

LPFN_RIODEREGISTERBUFFER callback function (mswsock.h)

10/22/2022 • 2 minutes to read • Edit Online

The **RIODeregisterBuffer** function deregisters a registered buffer used with the Winsock registered I/O extensions.

Syntax

```
LPFN_RIODEREGISTERBUFFER LpfnRioderegisterbuffer;

void LpfnRioderegisterbuffer(
  RIO_BUFFERID BufferId
)
{...}
```

Parameters

`BufferId`

A descriptor identifying a registered buffer.

Return value

None

Remarks

The **RIODeregisterBuffer** function deregisters a registered buffer. When a buffer is deregistered, the application is indicating that it is done with the buffer identifier passed in the *BufferId* parameter. Any subsequent calls to other functions that try to use this buffer identifier will fail.

If a buffer that is still in use is deregistered, the results are undefined. This is considered a serious error. In the RIORESULT structure returned by the RIODequeueCompletion function, the status will be unchanged from the normal status. An application developer can detect this error condition using the Application Verifier tool.

If an invalid buffer identifier is passed in the *BufferId* parameter, this is ignored by the **RIODeregisterBuffer** function.

> **NOTE**
>
> The function pointer to the **RIODeregisterBuffer** function must be obtained at run time by making a call to the WSAIoctl function with the **SIO_GET_MULTIPLE_EXTENSION_FUNCTION_POINTER** opcode specified. The input buffer passed to the **WSAIoctl** function must contain **WSAID_MULTIPLE_RIO**, a globally unique identifier (GUID) whose value identifies the Winsock registered I/O extension functions. On success, the output returned by the **WSAIoctl** function contains a pointer to the RIO_EXTENSION_FUNCTION_TABLE structure that contains pointers to the Winsock registered I/O extension functions. The **SIO_GET_MULTIPLE_EXTENSION_FUNCTION_POINTER** IOCTL is defined in the *Ws2def.h* header file. The **WSAID_MULTIPLE_RIO** GUID is defined in the *Mswsock.h* header file.

Windows Phone 8: This function is supported for Windows Phone Store apps on Windows Phone 8 and later.

Windows 8.1 and **Windows Server 2012 R2**: This function is supported for Windows Store apps on Windows 8.1, Windows Server 2012 R2, and later.

Requirements

Header	mswsock.h

LPFN_RIONOTIFY callback function (mswsock.h)

10/22/2022 • 4 minutes to read • Edit Online

The **RIONotify** function registers the method to use for notification behavior with an I/O completion queue for use with the Winsock registered I/O extensions.

Syntax

```
LPFN_RIONOTIFY LpfnRionotify;

INT LpfnRionotify(
  RIO_CQ CQ
)
{...}
```

Parameters

`CQ`

A descriptor that identifies an I/O completion queue.

Return value

If no error occurs, the **RIONotify** function returns **ERROR_SUCCESS**. Otherwise, the function failed and a specific error code is returned.

RETURN CODE	DESCRIPTION
WSAEINVAL	An invalid parameter was passed to the function. This error is returned if invalid completion queue is passed in the *CQ* parameter (**RIO_INVALID_CQ**, for example). This error can also be returned when an internal error occurs.
WSAEALREADY	An operation was attempted on a non-blocking socket that already had an operation in progress. This error is returned if a previous **RIONotify** request has not yet completed.

Remarks

The **RIONotify** function registers the method to be used for notification behavior for sending or receiving network data with the Winsock registered I/O extensions.

The **RIONotify** function is the mechanism by which an application finds out that requests are completed and are awaiting a call to the **RIODequeueCompletion** function. The **RIONotify** function sets the method to be used for notification behavior when an I/O completion queue is not empty and contains the completion of a result.

The notification behavior for a completion queue is set when the **RIO_CQ** is created. The **RIO_NOTIFICATION_COMPLETION** structure is passed to the **RIOCreateCompletionQueue** function when a **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.

When the **RIONotify** function is called used event completion and the specified completion queue is already not empty, the event is set either synchronously or asynchronously. In both cases, additional entries do not need to enter the completion queue before the event is set. Until the completion queue contains the completion of a request that did not have the **RIO_MSG_DONT_NOTIFY** flag set, the completion queue is considered empty for the purposes of the **RIONotify** function and the event is not set. Any completed requests can still be retrieved using the RIODequeueCompletion function. When the event is set, the application typically calls the **RIODequeueCompletion** function to dequeue the completed send and receive requests.

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 **Iocp.IocpHandle** 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 **Iocp.CompletionKey** member to distinguish **RIONotify** requests on the completion queue from other I/O completions including **RIONotify** completions for other completion queues.

An application using thread pools can use thread pool wait objects to get **RIONotify** completions via its thread pool. In that case, the call to the SetThreadpoolWait function should immediately follow the call to **RIONotify**. If the **SetThreadpoolWait** function is called before **RIONotify** and the application relies on **RIONotify** to clear the event object, this may result in spurious executions of the wait object callback.

> **NOTE**
>
> The function pointer to the **RIONotify** function must be obtained at run time by making a call to the WSAIoctl function with the **SIO_GET_MULTIPLE_EXTENSION_FUNCTION_POINTER** opcode specified. The input buffer passed to the **WSAIoctl** function must contain **WSAID_MULTIPLE_RIO**, a globally unique identifier (GUID) whose value identifies the Winsock registered I/O extension functions. On success, the output returned by the **WSAIoctl** function contains a pointer to the RIO_EXTENSION_FUNCTION_TABLE structure that contains pointers to the Winsock registered I/O extension functions. The **SIO_GET_MULTIPLE_EXTENSION_FUNCTION_POINTER** IOCTL is defined in the *Ws2def.h* header file. The **WSAID_MULTIPLE_RIO** GUID is defined in the *Mswsock.h* header file.

Windows Phone 8: This function is supported for Windows Phone Store apps on Windows Phone 8 and later.

Windows 8.1 and **Windows Server 2012 R2**: This function is supported for Windows Store apps on Windows 8.1, Windows Server 2012 R2, and later.

Thread Safety

If multiple threads attempt to access the same RIO_CQ using the RIODequeueCompletion function, 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

Header	mswsock.h

LPFN_RIORECEIVE callback function (mswsock.h)

10/22/2022 • 4 minutes to read • Edit Online

The **RIOReceive** function 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.

Syntax

```
LPFN_RIORECEIVE LpfnRioreceive;

BOOL LpfnRioreceive(
  RIO_RQ SocketQueue,
  PRIO_BUF pData,
  ULONG DataBufferCount,
  DWORD Flags,
  PVOID RequestContext
)
{...}
```

Parameters

`SocketQueue`

A descriptor that identifies a connected registered I/O TCP socket or a bound registered I/O UDP socket.

`pData`

A description of the portion of the registered buffer in which to receive data.

This parameter may be NULL for a bound registered I/O UDP socket if the application does not need to receive the data payload in the UDP datagram.

`DataBufferCount`

A data buffer count parameter that indicates if data is to be received in the buffer pointed to by the *pData* parameter.

This parameter should be set to zero if the *pData* is NULL. Otherwise, this parameter should be set to 1.

`Flags`

A set of flags that modify the behavior of the **RIOReceive** function.

The *Flags* parameter can contain a combination of the following options defined in the *Mswsockdef.h* header file:

and *RequestContext* parameters must be NULL and the *DataBufferCount* parameter must be zero.
This flag would normally be used occasionally after a number of requests were issued with the **RIO_MSG_DEFER** flag set. This eliminates the need when using the **RIO_MSG_DEFER** flag to make the last request without the **RIO_MSG_DEFER** flag, which causes the last request to complete much slower than other requests.
Unlike other calls to the **RIOReceive** function, when the **RIO_MSG_COMMIT_ONLY** flag is set calls to the **RIOReceive** function do not need to be serialized. For a single [**RIO_RQ**](/windows/win32/winsock/riorqueue), the **RIOReceive** function can be called with **RIO_MSG_COMMIT_ONLY** on one thread while calling the **RIOReceive** function on another thread.

RIO_RQ](/windows/win32/winsock/riorqueue) passed in the *SocketQueue* parameter without the **RIO_MSG_DEFER** flag set. To trigger execution for all receives in a request queue, call the **RIOReceive** or [**RIOReceiveEx**](./nc-mswsock-lpfn_rioreceiveex.md) function without the **RIO_MSG_DEFER** flag set.

> [!Note]
> The receive request is charged against the outstanding I/O capacity on the [**RIO_RQ**](/windows/win32/winsock/riorqueue) passed in the *SocketQueue* parameter regardless of whether **RIO_MSG_DEFER** is set.

parameter is completely full.

- The connection has been closed.
- The request has been canceled or an error occurred.

This flag is not supported on UDP sockets.

`RequestContext`

The request context to associate with this receive operation.

Return value

If no error occurs, the **RIOReceive** function returns **TRUE**. In this case, the receive operation is successfully initiated and the completion will have already been queued or the operation has been successfully initiated and the completion will be queued at a later time.

A value of **FALSE** indicates the function failed, the operation was not successfully initiated and no completion indication will be queued. A specific error code can be retrieved by calling the **WSAGetLastError** function.

RETURN CODE	DESCRIPTION
WSAEFAULT	The system detected an invalid pointer address in attempting to use a pointer argument in a call. This error is returned if a buffer identifier is deregistered or a buffer is freed for any of the **RIO_BUF** structures passed in parameters before the operation is queued or invoked.
WSAEINVAL	An invalid parameter was passed to the function. This error is returned if the *SocketQueue* parameter is not valid, the *Flags* parameter contains a value not valid for a receive operation, or the integrity of the completion queue has been compromised. This error can also be returned for other issues with parameters.
WSAENOBUFS	Sufficient memory could not be allocated. This error is returned if the I/O completion queue associated with the *SocketQueue* parameter is full or the I/O completion queue was created with zero receive entries.
WSA_OPERATION_ABORTED	The operation has been canceled while the receive operation was pending. This error is returned if the socket is closed locally or remotely, or the **SIO_FLUSH** command in **WSAIoctl** is executed on this socket.

Remarks

An application can use the **RIOReceive** function to receive network data into any buffer completely contained within a single registered buffer. The **Offset** and **Length** members of the RIO_BUF structure pointed to by the *pData* parameter determine where the network data is received in the buffer.

Once the **RIOReceive** function is called, the buffer passed in the *pData* parameter including the RIO_BUFFERID in the **BufferId** member of RIO_BUF structure must remain valid for the duration of the receive operation.

In order to avoid race conditions, a buffer associated with a receive request should not be read or written before the request completes. This includes using the buffer as the source for a send request or the destination for another receive request. Portions of a registered buffer not associated with any receive request are not included in this restriction.

The *Flags* parameter can be used to influence the behavior of the **RIOReceive** function invocation beyond the options specified for the associated socket. The behavior of this function is determined by a combination of any socket options set on the socket associated with the *SocketQueue* parameter and the values specified in the *Flags* parameter.

> **NOTE**
>
> The function pointer to the **RIOReceive** function must be obtained at run time by making a call to the WSAIoctl function with the **SIO_GET_MULTIPLE_EXTENSION_FUNCTION_POINTER** opcode specified. The input buffer passed to the **WSAIoctl** function must contain **WSAID_MULTIPLE_RIO**, a globally unique identifier (GUID) whose value identifies the Winsock registered I/O extension functions. On success, the output returned by the **WSAIoctl** function contains a pointer to the RIO_EXTENSION_FUNCTION_TABLE structure that contains pointers to the Winsock registered I/O extension functions. The **SIO_GET_MULTIPLE_EXTENSION_FUNCTION_POINTER** IOCTL is defined in the *Ws2def.h* header file. The **WSAID_MULTIPLE_RIO** GUID is defined in the *Mswsock.h* header file.

Windows Phone 8: This function is supported for Windows Phone Store apps on Windows Phone 8 and later.

Windows 8.1 and **Windows Server 2012 R2**: This function is supported for Windows Store apps on Windows 8.1, Windows Server 2012 R2, and later.

Requirements

Header	mswsock.h

LPFN_RIORECEIVEEX callback function (mswsock.h)

10/22/2022 • 7 minutes to read • Edit Online

The **RIOReceiveEx** function 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.

Syntax

```
LPFN_RIORECEIVEEX LpfnRioreceiveex;

int LpfnRioreceiveex(

RIO_RQ SocketQueue,

PRIO_BUF pData,

ULONG DataBufferCount,

PRIO_BUF pLocalAddress,

PRIO_BUF pRemoteAddress,

PRIO_BUF pControlContext,

PRIO_BUF pFlags,

DWORD Flags,

PVOID RequestContext
)
{...}
```

Parameters

`SocketQueue`

A descriptor that identifies a connected registered I/O UDP socket or a bound registered I/O UDP socket.

`pData`

A description of the portion of the registered buffer in which to receive data.

This parameter may be NULL for a bound registered I/O UDP socket if the application does not need to receive a data payload in the UDP datagram.

`DataBufferCount`

A data buffer count parameter that indicates if data is to be received in the buffer pointed to by the *pData* parameter.

This parameter should be set to zero if the *pData* is NULL. Otherwise, this parameter should be set to 1.

`pLocalAddress`

A buffer segment that on completion will hold the local address on which the network data was received.

This parameter may be **NULL** if the application does not want to receive the local address. If this parameter is

not **NULL**, then the buffer segment must be at least the size of a **SOCKADDR_INET** structure.

`pRemoteAddress`

A buffer segment that on completion will hold the remote address from which the network data was received.

This parameter may be **NULL** if the application does not want to receive the remote address. If this parameter is not **NULL**, then the buffer segment must be at least the size of a **SOCKADDR_INET** structure.

`pControlContext`

A buffer slice that on completion will hold additional control information about the receive operation.

This parameter may be **NULL** if the application does not want to receive the additional control information.

`pFlags`

`Flags`

A set of flags that modify the behavior of the **RIOReceiveEx** function.

The *Flags* parameter can contain a combination of the following options defined in the *Mswsockdef.h* header file:

, *pLocalAddress*, *pRemoteAddress*, *pControlContext*, *pFlags*, and *RequestContext* parameters must be NULL and the *DataBufferCount* parameter must be zero.
This flag would normally be used occasionally after a number of requests were issued with the **RIO_MSG_DEFER** flag set. This eliminates the need when using the **RIO_MSG_DEFER** flag to make the last request without the **RIO_MSG_DEFER** flag, which causes the last request to complete much slower than other requests.
Unlike other calls to the **RIOReceiveEx** function, when the **RIO_MSG_COMMIT_ONLY** flag is set calls to the **RIOReceiveEx** function do not need to be serialized. For a single [**RIO_RQ**](/windows/win32/winsock/riorqueue), the **RIOReceiveEx** function can be called with **RIO_MSG_COMMIT_ONLY** on one thread while calling the **RIOReceiveEx** function on another thread.
RIO_RQ](/windows/win32/winsock/riorqueue) passed in the *SocketQueue* parameter without the **RIO_MSG_DEFER** flag set. To trigger execution for all receives in a request queue, call the [**RIOReceive**](./nc-mswsock-lpfn_rioreceive.md) or **RIOReceiveEx** function without the **RIO_MSG_DEFER** flag set.

> [!Note]
> The receive request is charged against the outstanding I/O capacity on the [**RIO_RQ**](/windows/win32/winsock/riorqueue) passed in the *SocketQueue* parameter regardless of whether **RIO_MSG_DEFER** is set.

parameter is completely full.

- The connection has been closed.
- The request has been canceled or an error occurred.

This flag is not supported on datagram sockets or on message-oriented connectionless sockets.

`RequestContext`

The request context to associate with this receive operation.

Return value

If no error occurs, the **RIOReceiveEx** function returns **TRUE**. In this case, the receive operation is successfully initiated and the completion will have already been queued or the operation has been successfully initiated and

the completion will be queued at a later time.

A value of **FALSE** indicates the function failed, the operation was not successfully initiated and no completion indication will be queued. A specific error code can be retrieved by calling the **WSAGetLastError** function.

RETURN CODE	DESCRIPTION
WSAEFAULT	The system detected an invalid pointer address in attempting to use a pointer argument in a call. This error is returned if a buffer identifier is deregistered or a buffer is freed for any of the **RIO_BUF** structures passed in parameters before the operation is queued or invoked.
WSAEINVAL	An invalid parameter was passed to the function. This error is returned if the *SocketQueue* parameter is not valid, the *dwFlags* parameter contains a value not valid for a receive operation, or the integrity of the completion queue has been compromised. This error can also be returned for other issues with parameters.
WSAENOBUFS	Sufficient memory could not be allocated. This error is returned if the I/O completion queue associated with the *SocketQueue* parameter is full or the I/O completion queue was created with zero receive entries.
WSA_OPERATION_ABORTED	The operation has been canceled while the receive operation was pending. This error is returned if the socket is closed locally or remotely, or the the SIO_FLUSH command in **WSAIoctl** is executed.

Remarks

An application can use the **RIOReceiveEx** function to receive network data into any buffer completely contained within a single registered buffer. The **Offset** and **Length** members of the **RIO_BUF** structure pointed to by the *pData* parameter determine where the network data is received in the buffer.

Once the **RIOReceiveEx** function is called, the buffer passed in the *pData* parameter including the **RIO_BUFFERID** in the **BufferId** member of **RIO_BUF** structure must remain valid for the duration of the receive operation.

In order to avoid race conditions, a buffer associated with a receive request should not be read or written before the request completes. This includes using the buffer as the source for a send request or the destination for another receive request. Portions of a registered buffer not associated with any receive request are not included in this restriction.

The *pLocalAddress* parameter can be used to retrieve the local address where the data was received. The *pRemoteAddress* parameter can be used to retrieve the remote address from which the data was received. The local and remote addresses are returned as **SOCKADDR_INET** structures. As a result, the **Length** member of the **RIO_BUF** pointed to by *pLocalAddress* or *pRemoteAddress* parameters should be equal or greater than the size of a **SOCKADDR_INET** structure.

The following table summarizes the various uses of control data available for use with the control information in the *pControlContext* member.

PROTOCOL	CMSG_LEVEL	CMSG_TYPE	DESCRIPTION

PROTOCOL	CMSG_LEVEL	CMSG_TYPE	DESCRIPTION
IPv4	IPPROTO_IP	IP_ORIGINAL_ARRIVAL_IF	Receives the original IPv4 arrival interface where the packet was received for datagram sockets. This control data is used by firewalls when a Teredo, 6to4, or ISATAP tunnel is used for IPv4 NAT traversal. The cmsg_data[] member is a ULONG that contains the IF_INDEX defined in the *ifdef.h* header file. For more information, see the IPPROTO_IP Socket Options for the IP_ORIGINAL_ARRIVAL_IF socket option.
IPv4	IPPROTO_IP	IP_PKTINFO	Specifies/receives packet information. For more information, see the IPPROTO_IP Socket Options for the IP_PKTINFO socket option.
IPv6	IPPROTO_IPV6	IPV6_DSTOPTS	Specifies/receives destination options.
IPv6	IPPROTO_IPV6	IPV6_HOPLIMIT	Specifies/receives hop limit. For more information, see the **IPPROTO_IPV6 Socket Options** for the IPV6_HOPLIMIT socket option.
IPv6	IPPROTO_IPV6	IPV6_HOPOPTS	Specifies/receives hop-by-hop options.
IPv6	IPPROTO_IPV6	IPV6_NEXTHOP	Specifies next-hop address.
IPv6	IPPROTO_IPV6	IPV6_PKTINFO	Specifies/receives packet information. For more information, see the **IPPROTO_IPV6 Socket Options** for the IPV6_PKTINFO socket option.
IPv6	IPPROTO_IPV6	IPV6_RTHDR	Specifies/receives routing header.

Control data is made up of one or more control data objects, each beginning with a **WSACMSGHDR** structure, defined as the following:

```
} WSACMSGHDR;
```

The members of the **WSACMSGHDR** structure are as follows:

TERM	DESCRIPTION
cmsg_len	The number of bytes of data starting from the beginning of the **WSACMSGHDR** to the end of data (excluding padding bytes that may follow data).
cmsg_level	The protocol that originated the control information.
cmsg_type	The protocol-specific type of control information.

The *Flags* parameter can be used to influence the behavior of the **RIOReceiveEx** function invocation beyond the options specified for the associated socket. The behavior of this function is determined by a combination of any socket options set on the socket associated with the *SocketQueue* parameter and the values specified in the *Flags* parameter.

> **NOTE**
> The function pointer to the **RIOReceiveEx** function must be obtained at run time by making a call to the **WSAIoctl** function with the **SIO_GET_MULTIPLE_EXTENSION_FUNCTION_POINTER** opcode specified. The input buffer passed to the **WSAIoctl** function must contain **WSAID_MULTIPLE_RIO**, a globally unique identifier (GUID) whose value identifies the Winsock registered I/O extension functions. On success, the output returned by the **WSAIoctl** function contains a pointer to the **RIO_EXTENSION_FUNCTION_TABLE** structure that contains pointers to the Winsock registered I/O extension functions. The **SIO_GET_MULTIPLE_EXTENSION_FUNCTION_POINTER** IOCTL is defined in the *Ws2def.h* header file. The **WSAID_MULTIPLE_RIO** GUID is defined in the *Mswsock.h* header file.

Windows Phone 8: This function is supported for Windows Phone Store apps on Windows Phone 8 and later.

Windows 8.1 and **Windows Server 2012 R2**: This function is supported for Windows Store apps on Windows 8.1, Windows Server 2012 R2, and later.

Requirements

Header	mswsock.h

LPFN_RIOREGISTERBUFFER callback function (mswsock.h)

10/22/2022 • 2 minutes to read • Edit Online

The **RIORegisterBuffer** function registers a RIO_BUFFERID, a registered buffer descriptor, with a specified buffer for use with the Winsock registered I/O extensions.

Syntax

```
LPFN_RIOREGISTERBUFFER LpfnRioregisterbuffer;

RIO_BUFFERID LpfnRioregisterbuffer(
  PCHAR DataBuffer,
  DWORD DataLength
)
{...}
```

Parameters

`DataBuffer`

A pointer to the beginning of the memory buffer to register.

`DataLength`

The length, in bytes, in the buffer to register.

Return value

If no error occurs, the **RIORegisterBuffer** function returns a registered buffer descriptor. Otherwise, a value of **RIO_INVALID_BUFFERID** is returned, and a specific error code can be retrieved by calling the WSAGetLastError function.

RETURN CODE	DESCRIPTION
WSAEFAULT	The system detected an invalid pointer address in attempting to use a pointer argument in a call. This error is returned if an invalid buffer pointer is passed in *DataBuffer* parameter.
WSAEINVAL	An invalid parameter was passed to the function. This error is returned if the *DataLength* parameter is zero.

Remarks

The **RIORegisterBuffer** function creates a registered buffer identifier for a specified buffer. When a buffer is registered, the virtual memory pages containing the buffer will be locked into physical memory.

If several small, non-contiguous buffers are registered, the physical memory footprint for the buffers may effectively be as large as an entire memory page per registration. In these cases it may be beneficial to allocate

multiple request buffers together.

There is also a small amount of overhead in physical memory used for the buffer registration itself. So if there are many allocations aggregated into single larger allocation, the physical memory footprint may be reduced further by aggregating the buffer registrations as well. In this case the application may need to take extra care to ensure that the buffers are eventually deregistered, but not while any send or receive requests are outstanding.

A portion of a registered buffer is passed to the **RIOSend**, **RIOSendEx**, **RIOReceive**, and **RIOReceiveEx** functions in the *pData* parameter for sending or receiving data.

When the buffer identifier is no longer needed, call the **RIODeregisterBuffer** function to deregister the buffer identifier.

> **NOTE**
>
> The function pointer to the **RIORegisterBuffer** function must be obtained at run time by making a call to the **WSAIoctl** function with the **SIO_GET_MULTIPLE_EXTENSION_FUNCTION_POINTER** opcode specified. The input buffer passed to the **WSAIoctl** function must contain **WSAID_MULTIPLE_RIO**, a globally unique identifier (GUID) whose value identifies the Winsock registered I/O extension functions. On success, the output returned by the **WSAIoctl** function contains a pointer to the **RIO_EXTENSION_FUNCTION_TABLE** structure that contains pointers to the Winsock registered I/O extension functions. The **SIO_GET_MULTIPLE_EXTENSION_FUNCTION_POINTER** IOCTL is defined in the *Ws2def.h* header file. The **WSAID_MULTIPLE_RIO** GUID is defined in the *Mswsock.h* header file.

Windows Phone 8: This function is supported for Windows Phone Store apps on Windows Phone 8 and later.

Windows 8.1 and **Windows Server 2012 R2**: This function is supported for Windows Store apps on Windows 8.1, Windows Server 2012 R2, and later.

Requirements

Header	mswsock.h

LPFN_RIORESIZECOMPLETIONQUEUE callback function (mswsock.h)

10/22/2022 • 2 minutes to read • Edit Online

The **RIOResizeCompletionQueue** function resizes an I/O completion queue to be either larger or smaller for use with the Winsock registered I/O extensions.

Syntax

```
LPFN_RIORESIZECOMPLETIONQUEUE LpfnRioresizecompletionqueue;

BOOL LpfnRioresizecompletionqueue(
  RIO_CQ CQ,
  DWORD QueueSize
)
{...}
```

Parameters

`CQ`

A descriptor that identifies an existing I/O completion queue to resize.

`QueueSize`

Return value

If no error occurs, the **RIOResizeCompletionQueue** function returns **TRUE**. Otherwise, a value of **FALSE** is returned, and a specific error code can be retrieved by calling the **WSAGetLastError** function.

RETURN CODE	DESCRIPTION
WSAEFAULT	The system detected an invalid pointer address in attempting to use a pointer argument in a call. This error is returned if the completion queue specified in the *CQ* parameter contains an invalid pointer.
WSAEINVAL	An invalid parameter was passed to the function. This error is returned if the *CQ* parameter is not valid (RIO_INVALID_CQ, for example). This error is also returned if the size of the queue specified in the *QueueSize* parameter is greater than **RIO_CQ_MAX_SIZE**.
WSAENOBUFS	Sufficient memory could not be allocated. This error is returned if memory could not be allocated for the queue specified in the *QueueSize* parameter.
WSAETOOMANYREFS	There are too many operations that still reference the I/O completion queue. Resizing of this I/O completion queue to be smaller is not possible at this time.

The **RIOResizeCompletionQueue** function resizes an I/O completion queue to be either larger or smaller. If

the I/O completion queue already contains completions, those completions will be copied over to the new completion queue.

I/O completion queues have a required minimum size that is dependent on the number of request queues associated with the completion queue and the number of sends and receives on the request queues. If an application calls the **RIOResizeCompletionQueue** function and tries to set the queue too small for the number of existing completions in the I/O completion queue, the call will fail and the queue will not be resized.

> **NOTE**
>
> The function pointer to the **RIOResizeCompletionQueue** function must be obtained at run time by making a call to the **WSAIoctl** function with the **SIO_GET_MULTIPLE_EXTENSION_FUNCTION_POINTER** opcode specified. The input buffer passed to the **WSAIoctl** function must contain **WSAID_MULTIPLE_RIO**, a globally unique identifier (GUID) whose value identifies the Winsock registered I/O extension functions. On success, the output returned by the **WSAIoctl** function contains a pointer to the **RIO_EXTENSION_FUNCTION_TABLE** structure that contains pointers to the Winsock registered I/O extension functions. The **SIO_GET_MULTIPLE_EXTENSION_FUNCTION_POINTER** IOCTL is defined in the *Ws2def.h* header file. The **WSAID_MULTIPLE_RIO** GUID is defined in the *Mswsock.h* header file.

Windows Phone 8: This function is supported for Windows Phone Store apps on Windows Phone 8 and later.

Windows 8.1 and **Windows Server 2012 R2**: This function is supported for Windows Store apps on Windows 8.1, Windows Server 2012 R2, and later.

Thread Safety

If multiple threads attempt to access the same **RIO_CQ** using the **RIODequeueCompletion** or **RIOResizeCompletionQueue** function, 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

Header	mswsock.h

LPFN_RIORESIZEREQUESTQUEUE callback function (mswsock.h)

10/22/2022 • 2 minutes to read • Edit Online

The **RIOResizeRequestQueue** function resizes a request queue to be either larger or smaller for use with the Winsock registered I/O extensions.

Syntax

```
LPFN_RIORESIZEREQUESTQUEUE LpfnRioresizerequestqueue;

BOOL LpfnRioresizerequestqueue(
  RIO_RQ RQ,
  DWORD MaxOutstandingReceive,
  DWORD MaxOutstandingSend
)
{...}
```

Parameters

`RQ`

A descriptor that identifies an existing registered I/O socket descriptor (request queue) to resize.

`MaxOutstandingReceive`

The maximum number of outstanding sends allowed on the socket. This value can be larger or smaller than the original number.

This parameter is usually a small number for most applications.

`MaxOutstandingSend`

The maximum number of outstanding receives allowed on the socket. This value can be larger or smaller than the original number.

Return value

If no error occurs, the **RIOResizeRequestQueue** function returns **TRUE**. Otherwise, a value of **FALSE** is returned, and a specific error code can be retrieved by calling the **WSAGetLastError** function.

RETURN CODE	DESCRIPTION
WSAEINVAL	An invalid parameter was passed to the function. This error is returned if the *RQ* parameter is not valid (RIO_INVALID_RQ, for example). This error is also returned if both the *MaxOutstandingReceive* and *MaxOutstandingSend* parameters are zero.
WSAENOBUFS	Sufficient memory could not be allocated. This error is returned if memory could not be allocated for the resized request queue.

RETURN CODE	DESCRIPTION
WSAETOOMANYREFS	There are too many operations that still reference the request queue. Resizing of this request queue to be smaller is not possible at this time.

Remarks

The **RIOResizeRequestQueue** function resizes a request queue to be either larger or smaller. If the request queue already contains entries, those entries will be copied over to the new request queue.

A request queue has a required minimum size that is dependent on the current number of entries (number of sends and receives on the request queue). If an application calls the **RIOResizeRequestQueue** function and tries to set the queue too small for the number of existing entries, the call will fail and the queue will not be resized.

> **NOTE**
>
> The function pointer to the **RIOResizeRequestQueue** function must be obtained at run time by making a call to the **WSAIoctl** function with the **SIO_GET_MULTIPLE_EXTENSION_FUNCTION_POINTER** opcode specified. The input buffer passed to the **WSAIoctl** function must contain **WSAID_MULTIPLE_RIO**, a globally unique identifier (GUID) whose value identifies the Winsock registered I/O extension functions. On success, the output returned by the **WSAIoctl** function contains a pointer to the **RIO_EXTENSION_FUNCTION_TABLE** structure that contains pointers to the Winsock registered I/O extension functions. The **SIO_GET_MULTIPLE_EXTENSION_FUNCTION_POINTER** IOCTL is defined in the *Ws2def.h* header file. The **WSAID_MULTIPLE_RIO** GUID is defined in the *Mswsock.h* header file.

Windows Phone 8: This function is supported for Windows Phone Store apps on Windows Phone 8 and later.

Windows 8.1 and **Windows Server 2012 R2**: This function is supported for Windows Store apps on Windows 8.1, Windows Server 2012 R2, and later.

Thread Safety

If multiple threads attempt to access the same **RIO_RQ** using the **RIODequeueCompletion** or **RIOResizeRequestQueue** function, 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

Header	mswsock.h

LPFN_RIOSEND callback function (mswsock.h)

10/22/2022 • 4 minutes to read • Edit Online

The **RIOSend** function 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.

Syntax

```
LPFN_RIOSEND LpfnRiosend;

BOOL LpfnRiosend(
  RIO_RQ SocketQueue,
  PRIO_BUF pData,
  ULONG DataBufferCount,
  DWORD Flags,
  PVOID RequestContext
)
{...}
```

Parameters

`SocketQueue`

A descriptor that identifies a connected registered I/O TCP socket or a bound registered I/O UDP socket.

`pData`

A description of the portion of the registered buffer from which to send data.

This parameter may be NULL for a bound registered I/O UDP socket if the application does not need to send a data payload in the UDP datagram.

`DataBufferCount`

A data buffer count parameter that indicates if data is to be sent in the buffer pointed to by the *pData* parameter.

This parameter should be set to zero if the *pData* is NULL. Otherwise, this parameter should be set to 1.

`Flags`

A set of flags that modify the behavior of the **RIOSend** function.

The *Flags* parameter can contain a combination of the following options defined in the *Mswsockdef.h* header file:

and *RequestContext* parameters must be NULL and the *DataBufferCount* parameter must be zero.
This flag would normally be used occasionally after a number of requests were issued with the **RIO_MSG_DEFER** flag set. This eliminates the need when using the **RIO_MSG_DEFER** flag to make the last request without the **RIO_MSG_DEFER** flag, which causes the last request to complete much slower than other requests.
Unlike other calls to the **RIOSend** function, when the **RIO_MSG_COMMIT_ONLY** flag is set calls to the **RIOSend** function do not need to be serialized. For a single [**RIO_RQ**](/windows/win32/winsock/riorqueue), the **RIOSend** function can be called with **RIO_MSG_COMMIT_ONLY** on one thread while calling the **RIOSend** function on another thread.
RIO_RQ](/windows/win32/winsock/riorqueue) passed in the *SocketQueue* parameter without the

RIO_MSG_DEFER flag set. To trigger execution for all sends in a send queue, call the **RIOSend** or [**RIOSendEx**](./nc-mswsock-lpfn_riosendex.md) function without the **RIO_MSG_DEFER** flag set.

> [!Note]
> The send request is charged against the outstanding I/O capacity on the [**RIO_RQ**](/windows/win32/winsock/riorqueue) passed in the *SocketQueue* parameter regardless of whether **RIO_MSG_DEFER** is set.

`RequestContext`

The request context to associate with this send operation.

Return value

If no error occurs, the **RIOSend** function returns **TRUE**. In this case, the send operation is successfully initiated and the completion will have already been queued or the operation has been successfully initiated and the completion will be queued at a later time.

A value of **FALSE** indicates the function failed, the operation was not successfully initiated and no completion indication will be queued. A specific error code can be retrieved by calling the **WSAGetLastError** function.

RETURN CODE	DESCRIPTION
WSAEFAULT	The system detected an invalid pointer address in attempting to use a pointer argument in a call. This error is returned if a buffer identifier is deregistered or a buffer is freed for any of the **RIO_BUF** structures passed in parameters before the operation is queued or invoked.
WSAEINVAL	An invalid parameter was passed to the function. This error is returned if the *SocketQueue* parameter is not valid, the *Flags* parameter contains a value not valid for a send operation, or the integrity of the completion queue has been compromised. This error can also be returned for other issues with parameters.
WSAENOBUFS	Sufficient memory could not be allocated. This error is returned if the I/O completion queue associated with the *SocketQueue* parameter is full or the I/O completion queue was created with zero send entries.
WSA_IO_PENDING	The operation has been successfully initiated and the completion will be queued at a later time.

Remarks

An application can use the **RIOSend** function to send network data from any buffer completely contained within a single registered buffer. The **Offset** and **Length** members of the **RIO_BUF** structure pointed to by the *pData* parameter determine the network data to be sent from the buffer.

The buffer associated with a send operation must not be used concurrently with another send or receive operation. The buffer, and buffer registration, must remain valid for the duration of a send operation. This means that you should not pass the same PRIO_BUF to a RIOSend(Ex) request when one is already pending. Only after an in-flight RIOSend(Ex) request is complete should you re-use the same PRIO_BUF (either with the same offset

or with a different offset and length). Furthermore, when send data references a registered buffer (either a portion or the entire buffer), the entire registered buffer must not be used until the send has completed. This includes using a portion of the registered buffer for a receive operation or another send operation.

The *Flags* parameter can be used to influence the behavior of the **RIOSend** function beyond the options specified for the associated socket. The behavior of this function is determined by a combination of any socket options set on the socket associated with the *SocketQueue* parameter and the values specified in the *Flags* parameter.

> **NOTE**
>
> The function pointer to the **RIOSend** function must be obtained at run time by making a call to the **WSAIoctl** function with the **SIO_GET_MULTIPLE_EXTENSION_FUNCTION_POINTER** opcode specified. The input buffer passed to the **WSAIoctl** function must contain **WSAID_MULTIPLE_RIO**, a globally unique identifier (GUID) whose value identifies the Winsock registered I/O extension functions. On success, the output returned by the **WSAIoctl** function contains a pointer to the **RIO_EXTENSION_FUNCTION_TABLE** structure that contains pointers to the Winsock registered I/O extension functions. The **SIO_GET_MULTIPLE_EXTENSION_FUNCTION_POINTER** IOCTL is defined in the *Ws2def.h* header file. The **WSAID_MULTIPLE_RIO** GUID is defined in the *Mswsock.h* header file.

Windows Phone 8: This function is supported for Windows Phone Store apps on Windows Phone 8 and later.

Windows 8.1 and **Windows Server 2012 R2**: This function is supported for Windows Store apps on Windows 8.1, Windows Server 2012 R2, and later.

Requirements

Header	mswsock.h

LPFN_RIOSENDEX callback function (mswsock.h)

10/22/2022 • 6 minutes to read • Edit Online

The **RIOSendEx** function 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.

Syntax

```
LPFN_RIOSENDEX LpfnRiosendex;

BOOL LpfnRiosendex(
  RIO_RQ SocketQueue,
  PRIO_BUF pData,
  ULONG DataBufferCount,
  PRIO_BUF pLocalAddress,
  PRIO_BUF pRemoteAddress,
  PRIO_BUF pControlContext,
  PRIO_BUF pFlags,
  DWORD Flags,
  PVOID RequestContext
)
{...}
```

Parameters

`SocketQueue`

A descriptor that identifies a connected registered I/O TCP socket or a bound registered I/O UDP socket.

`pData`

A buffer segment from a registered buffer from which to send data. The **RIO_BUF** structure pointed to by this parameter can represent a portion of a registered buffer or a complete registered buffer.

This parameter may be NULL for a bound registered I/O UDP socket if the application does not need to send a data payload in the UDP datagram.

`DataBufferCount`

A data buffer count parameter that indicates if data is to be sent in the buffer pointed to by the *pData* parameter.

This parameter should be set to zero if the *pData* is NULL. Otherwise, this parameter should be set to 1.

`pLocalAddress`

This parameter is reserved and must be **NULL**.

`pRemoteAddress`

A buffer segment from a registered buffer that on input holds the remote address to which the network data is to be sent.

This parameter may be **NULL** if the socket is connected.

`pControlContext`

A buffer slice that on completion will hold additional control information about the send operation.

This parameter may be **NULL** if the application does not want to receive the additional control information.

`pFlags`

A buffer slice that on completion will hold additional information about the set of flags for the send operation.

This parameter may be **NULL** if the application does not want to receive the additional flags information.

`Flags`

A set of flags that modify the behavior of the **RIOSendEx** function.

The *Flags* parameter can contain a combination of the following options defined in the *Mswsockdef.h* header file:

, *pLocalAddress*, *pRemoteAddress*, *pControlContext*, *pFlags*, and *RequestContext* parameters must be NULL and the *DataBufferCount* parameter must be zero.
This flag would normally be used occasionally after a number of requests were issued with the **RIO_MSG_DEFER** flag set. This eliminates the need when using the **RIO_MSG_DEFER** flag to make the last request without the **RIO_MSG_DEFER** flag, which causes the last request to complete much slower than other requests.
Unlike other calls to the **RIOSendEx** function, when the **RIO_MSG_COMMIT_ONLY** flag is set calls to the **RIOSendEx** function do not need to be serialized. For a single [**RIO_RQ**](/windows/win32/winsock/riorqueue), the **RIOSendEx** function can be called with **RIO_MSG_COMMIT_ONLY** on one thread while calling the **RIOSendEx** function on another thread.
RIO_RQ](/windows/win32/winsock/riorqueue) passed in the *SocketQueue* parameter without the **RIO_MSG_DEFER** flag set. To trigger execution for all sends in a send queue, call the [**RIOSend**](./nc-mswsock-lpfn_riosend.md) or **RIOSendEx** function without the **RIO_MSG_DEFER** flag set.

> [!Note]
> The send request is charged against the outstanding I/O capacity on the [**RIO_RQ**](/windows/win32/winsock/riorqueue) passed in the *SocketQueue* parameter regardless of whether **RIO_MSG_DEFER** is set.

`RequestContext`

The request context to associate with this send operation.

Return value

If no error occurs, the **RIOSendEx** function returns **TRUE**. In this case, the send operation is successfully initiated and the completion will have already been queued or the operation has been successfully initiated and the completion will be queued at a later time.

A value of **FALSE** indicates the function failed, the operation was not successfully initiated and no completion indication will be queued. A specific error code can be retrieved by calling the **WSAGetLastError** function.

RETURN CODE	DESCRIPTION
WSAEFAULT	The system detected an invalid pointer address in attempting to use a pointer argument in a call. This error is returned if a buffer identifier is deregistered or a buffer is freed for any of the **RIO_BUF** structures passed in parameters before the operation is queued or invoked.

RETURN CODE	DESCRIPTION
WSAEINVAL	An invalid parameter was passed to the function. This error is returned if the *SocketQueue* parameter is not valid, the *Flags* parameter contains a value not valid for a send operation, or the integrity of the completion queue has been compromised. This error can also be returned for other issues with parameters.
WSAENOBUFS	Sufficient memory could not be allocated. This error is returned if the I/O completion queue associated with the *SocketQueue* parameter is full or the I/O completion queue was created with zero send entries.
WSA_IO_PENDING	The operation has been successfully initiated and the completion will be queued at a later time.

Remarks

An application can use the **RIOSendEx** function to send network data from any buffer completely contained within a single registered buffer. The **Offset** and **Length** members of the RIO_BUF structure pointed to by the *pData* parameter determine the network data to be sent from the buffer.

The buffer associated with a send operation must not be used concurrently with another send or receive operation. The buffer, and buffer registration, must remain valid for the duration of a send operation. This means that you should not pass the same PRIO_BUF to a RIOSend(Ex) request when one is already pending. Only after an in-flight RIOSend(Ex) request is complete should you re-use the same PRIO_BUF (either with the same offset or with a different offset and length). Furthermore, when send data references a registered buffer (either a portion or the entire buffer), the entire registered buffer must not be used until the send has completed. This includes using a portion of the registered buffer for a receive operation or another send operation.

The *pLocalAddress* parameter can be used to retrieve the local address from which the data was sent. The *pRemoteAddress* parameter can be used to retrieve the remote address to which the data was sent. The local and remote addresses are returned as SOCKADDR_INET structures. As a result, the **Length** member of the RIO_BUF pointed to by *pLocalAddress* or *pRemoteAddress* parameters should be equal or greater than the size of a **SOCKADDR_INET** structure.

The following table summarizes the various uses of control data available for use with the control information in the *pControlContext* member.

PROTOCOL	CMSG_LEVEL	CMSG_TYPE	DESCRIPTION
IPv4	IPPROTO_IP	IP_PKTINFO	Specifies/receives packet information. For more information, see the IPPROTO_IP Socket Options for the IP_PKTINFO socket option.
IPv6	IPPROTO_IPV6	IPV6_DSTOPTS	Specifies/receives destination options.

PROTOCOL	CMSG_LEVEL	CMSG_TYPE	DESCRIPTION
IPv6	IPPROTO_IPV6	IPV6_HOPLIMIT	Specifies/receives hop limit. For more information, see the **IPPROTO_IPV6 Socket Options** for the IPV6_HOPLIMIT socket option.
IPv6	IPPROTO_IPV6	IPV6_HOPOPTS	Specifies/receives hop-by-hop options.
IPv6	IPPROTO_IPV6	IPV6_NEXTHOP	Specifies next-hop address.
IPv6	IPPROTO_IPV6	IPV6_PKTINFO	Specifies/receives packet information. For more information, see the **IPPROTO_IPV6 Socket Options** for the IPV6_PKTINFO socket option.
IPv6	IPPROTO_IPV6	IPV6_RTHDR	Specifies/receives routing header.

Control data is made up of one or more control data objects, each beginning with a **WSACMSGHDR** structure, defined as the following:

```
} WSACMSGHDR;
```

The members of the **WSACMSGHDR** structure are as follows:

TERM	DESCRIPTION
cmsg_len	The number of bytes of data starting from the beginning of the **WSACMSGHDR** to the end of data (excluding padding bytes that may follow data).
cmsg_level	The protocol that originated the control information.
cmsg_type	The protocol-specific type of control information.

The *Flags* parameter can be used to influence the behavior of the **RIOSendEx** function beyond the options specified for the associated socket. The behavior of this function is determined by a combination of any socket options set on the socket associated with the *SocketQueue* parameter and the values specified in the *Flags* parameter.

> **NOTE**
> The function pointer to the **RIOSendEx** function must be obtained at run time by making a call to the **WSAIoctl** function with the **SIO_GET_MULTIPLE_EXTENSION_FUNCTION_POINTER** opcode specified. The input buffer passed to the **WSAIoctl** function must contain **WSAID_MULTIPLE_RIO**, a globally unique identifier (GUID) whose value identifies the Winsock registered I/O extension functions. On success, the output returned by the **WSAIoctl** function contains a pointer to the **RIO_EXTENSION_FUNCTION_TABLE** structure that contains pointers to the Winsock registered I/O extension functions. The **SIO_GET_MULTIPLE_EXTENSION_FUNCTION_POINTER** IOCTL is defined in the *Ws2def.h* header file. The **WSAID_MULTIPLE_RIO** GUID is defined in the *Mswsock.h* header file.

Windows Phone 8: This function is supported for Windows Phone Store apps on Windows Phone 8 and later.

Windows 8.1 and **Windows Server 2012 R2**: This function is supported for Windows Store apps on Windows 8.1, Windows Server 2012 R2, and later.

Requirements

Header	mswsock.h

LPFN_TRANSMITPACKETS callback function (mswsock.h)

10/22/2022 • 8 minutes to read • Edit Online

The **TransmitPackets** function transmits in-memory data or file data over a connected socket. The **TransmitPackets** function uses the operating system cache manager to retrieve file data, locking memory for the minimum time required to transmit and resulting in efficient, high-performance transmission.

> **Note** This function is a Microsoft-specific extension to the Windows Sockets specification.

Syntax

```
LPFN_TRANSMITPACKETS LpfnTransmitpackets;

BOOL LpfnTransmitpackets(
  SOCKET hSocket,
  LPTRANSMIT_PACKETS_ELEMENT lpPacketArray,
  DWORD nElementCount,
  DWORD nSendSize,
  LPOVERLAPPED lpOverlapped,
  DWORD dwFlags
)
{...}
```

Parameters

`hSocket`

A handle to the connected socket to be used in the transmission. Although the socket does not need to be a connection-oriented circuit, the default destination/peer should have been established using the connect, WSAConnect, accept, WSAAccept, AcceptEx, or WSAJoinLeaf function.

`lpPacketArray`

An array of type TRANSMIT_PACKETS_ELEMENT, describing the data to be transmitted.

`nElementCount`

The number of elements in *lpPacketArray*.

`nSendSize`

The size, in bytes, of the data block used in the send operation. Set *nSendSize* to zero to let the sockets layer select a default **send** size.

Setting *nSendSize* to 0xFFFFFFF enables the caller to control the size and content of each send request, achieved by using the TP_ELEMENT_EOP flag in the TRANSMIT_PACKETS_ELEMENT array pointed to in the *lpPacketArray* parameter. This capability is useful for message protocols that place limitations on the size of individual **send** requests.

`lpOverlapped`

A pointer to an OVERLAPPED structure. If the socket handle specified in the *hSocket* parameter has been opened as overlapped, use this parameter to achieve asynchronous (overlapped) I/O operation. Socket handles are opened as overlapped by default.

`dwFlags`

A set of flags used to customize processing of the **TransmitPackets** function. The following table outlines the use of the *dwFlags* parameter.

VALUE	MEANING
TF_DISCONNECT	Starts a transport-level disconnect after all the file data has been queued for transmission. Applies only to connection-oriented sockets. Specifying this flag for sockets that do not support disconnect semantics (such as datagram sockets) results in an error.
TF_REUSE_SOCKET	Prepares the socket handle to be reused. When the **TransmitPackets** function completes, the socket handle can be passed to the AcceptEx function. Valid only when a connection-oriented socket and TF_DISCONNECT are specified. **Note** The socket level packet transmit is subject to the behavior of the underlying transport. For example, a TCP socket may be subject to the TCP TIME_WAIT state, causing the **TransmitPackets** call to be delayed.
TF_USE_DEFAULT_WORKER	Directs Winsock to use the system's default thread to process long **TransmitPackets** requests. Long **TransmitPackets** requests are defined as requests that require more than a single read from the file or a cache; the long request definition therefore depends on the size of the file and the specified length of the send packet. The system default thread can be adjusted using the following registry parameter as a REG_DWORD:**HKEY_LOCAL_MACHINE\CurrentControlSet\Services\AFD\Parameters\TransmitWorker**
TF_USE_SYSTEM_THREAD	Directs Winsock to use system threads to process long **TransmitPackets** requests. Long **TransmitPackets** requests are defined as requests that require more than a single read from the file or a cache; the long request definition therefore depends on the size of the file and the specified length of the send packet.
TF_USE_KERNEL_APC	Directs Winsock to use kernel Asynchronous Procedure Calls (APCs) instead of worker threads to process long **TransmitPackets** requests. Long **TransmitPackets** requests are defined as requests that require more than a single read from the file or a cache; the long request definition therefore depends on the size of the file and the specified length of the send packet. See Remarks for more information.

Return value

If the **TransmitPackets** function succeeds, the return value is **TRUE**. Otherwise, the return value is **FALSE**. To get extended error information, call WSAGetLastError. An error code of WSA_IO_PENDING or ERROR_IO_PENDING indicates that the overlapped operation has been successfully initiated and that completion will be indicated at a later time. Any other error code indicates that the overlapped operation was not successfully initiated and no completion indication will occur. Applications should handle either ERROR_IO_PENDING or WSA_IO_PENDING in this case.

RETURN CODE	DESCRIPTION
WSAECONNABORTED	An established connection was aborted by the software in your host machine. This error is returned if the virtual circuit was terminated due to a time-out or other failure.
WSAECONNRESET	An existing connection was forcibly closed by the remote host. This error is returned for a stream socket when the virtual circuit was reset by the remote side. The application should close the socket as it is no longer usable.
WSAEFAULT	The system detected an invalid pointer address in attempting to use a pointer argument in a call. This error is returned if the *lpPacketArray* or the *lpOverlapped* parameter is not totally contained in a valid part of the user address space.
WSAEINVAL	An invalid argument was supplied. This error is returned if the *dwFlags* parameter has the **TF_REUSE_SOCKET** flag set, but the **TF_DISCONNECT** flag was not set. This error is also returned if the offset specified in the OVERLAPPED structure pointed to by the *lpOverlapped* is not within the file. This error is also returned if the total number of bytes to be transmitted is a value greater than 2,147,483,646, the maximum value for a 32-bit integer minus 1.
WSAENETDOWN	A socket operation encountered a dead network.This error is returned if the network subsystem has failed.
WSAENETRESET	The connection has been broken due to keep-alive activity detecting a failure while the operation was in progress. This error is returned for a stream socket where the connection was broken due to keep-alive activity detecting a failure.
WSAENOBUFS	An operation on a socket could not be performed because the system lacked sufficient buffer space or because a queue was full. This error is also returned if the Windows Sockets provider reports a buffer deadlock.
WSAENOTCONN	A request to send or receive data was disallowed because the socket is not connected. This error is returned for a stream socket.
WSAENOTSOCK	An operation was attempted on something that is not a socket. This error is returned if the *hSocket* parameter is not a socket.

WSAESHUTDOWN	A request to send or receive data was disallowed because the socket had already been shut down in that direction with a previous shutdown call. This error is returned if a stream socket has been shut down for sending. It is not possible to call TransmitFile on a stream socket after the shutdown function has been called on the socket with the *how* parameter set to **SD_SEND** or **SD_BOTH**.
WSANOTINITIALISED	Either the application has not called the WSAStartup function, or **WSAStartup** failed. A successful **WSAStartup** call must occur before using the TransmitFile function.
WSA_IO_PENDING	An overlapped I/O operation is in progress. This value is returned if an overlapped I/O operation was successfully initiated and indicates that 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 the overlapped operation has been canceled due to the closure of the socket, the execution of the "SIO_FLUSH" command in WSAIoctl, or the thread that initiated the overlapped request exited before the operation completed. **Note** All I/O initiated by a given thread is canceled when that thread exits. For overlapped sockets, pending asynchronous operations can fail if the thread is closed before the asynchronous operations complete. For more information, see ExitThread.

Remarks

The **TransmitPackets** function is optimized according to the operating system on which it is used:

- On Windows server editions, the **TransmitPackets** function is optimized for high performance.
- On Windows client editions, the **TransmitPackets** function is optimized for minimum memory and resource utilization.

The maximum number of bytes that can be transmitted using a single call to the **TransmitPackets** function is 2,147,483,646, the maximum value for a 32-bit integer minus 1. If an application needs to transmit data larger than 2,147,483,646 bytes, then multiple calls to the **TransmitPackets** function can be used with each call transferring no more than 2,147,483,646 bytes.

> **Note** The function pointer for the **TransmitPackets** function must be obtained at run time by making a call to the WSAIoctl function with the **SIO_GET_EXTENSION_FUNCTION_POINTER** opcode specified. The input buffer passed to the **WSAIoctl** function must contain **WSAID_TRANSMITPACKETS**, a globally unique identifier (GUID) whose value identifies the **TransmitPackets** extension function. On success, the output returned by the **WSAIoctl** function contains a pointer to the **TransmitPackets** function. The **WSAID_TRANSMITPACKETS** GUID is defined in the *Mswsock.h* header file.

Expect better performance results when using the **TransmitPackets** function on Windows Server 2003.

When *lpOverlapped* is not **NULL**, overlapped I/O might not finish before the **TransmitPackets** function returns. When this occurs, the **TransmitPackets** function returns fails, and a call to the WSAGetLastError function returns ERROR_IO_PENDING, allowing the caller to continue processing while the transmission completes.

> **Note** All I/O initiated by a given thread is canceled when that thread exits. For overlapped sockets, pending asynchronous operations can fail if the thread is closed before the operations complete. See ExitThread for more information.

When the **TransmitPackets** function returns **TRUE** or returns **FALSE** and WSAGetLastError returns ERROR_IO_PENDING, Windows sets the event specified by the **hEvent** member of the OVERLAPPED structure or the socket specified by *hSocket* to the signaled state, and upon completion, delivers notification to any completion port associated with the socket. Use GetOverlappedResult, or WSAGetOverlappedResult, or GetQueuedCompletionStatus to retrieve final status and number of bytes transmitted.

TransmitPackets and Asynchronous Procedure Calls (APCs)

Use of the TF_USE_KERNEL_APC flag can deliver significant performance benefits. If the thread initiating the **TransmitPackets** function call is being used for heavy computations, it is possible, though unlikely, that APCs could be prevented from launching.

> **Note** There is a difference between kernel and user-mode APCs:
> - Kernel APCs launch when a thread is in a wait state.
> - User-mode APCs launch when a thread is in an alertable wait state.

Windows Phone 8: This function is supported for Windows Phone Store apps on Windows Phone 8 and later.

Windows 8.1 and **Windows Server 2012 R2**: This function is supported for Windows Store apps on Windows 8.1, Windows Server 2012 R2, and later.

Requirements

Minimum supported client	Windows 8.1, Windows Vista [desktop apps | UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps | UWP apps]
Target Platform	Windows
Header	mswsock.h

See also

AcceptEx

GetOverlappedResult

GetQueuedCompletionStatus

OVERLAPPED

TRANSMIT_PACKETS_ELEMENT

TransmitFile

WSAAccept

WSAConnect

WSAGetOverlappedResult

WSAJoinLeaf

Winsock Functions

Winsock Reference

accept

connect

send

LPFN_WSARECVMSG callback function (mswsock.h)

10/22/2022 • 13 minutes to read • Edit Online

LPFN_WSARECVMSG is a function pointer type. You implement a matching **WSARecvMsg** callback function in your app. The system uses your callback function to transmit to you in-memory data, or file data, over a connected socket.

Your **WSARecvMsg** callback function receives ancillary data/control information with a message, from connected and unconnected sockets.

> **NOTE**
> This function is a Microsoft-specific extension to the Windows Sockets specification.

Syntax

```
LPFN_WSARECVMSG LpfnWsarecvmsg;

INT LpfnWsarecvmsg(
  SOCKET s,
  LPWSAMSG lpMsg,
  LPDWORD lpdwNumberOfBytesRecvd,
  LPWSAOVERLAPPED lpOverlapped,
  LPWSAOVERLAPPED_COMPLETION_ROUTINE lpCompletionRoutine
)
{...}
```

Parameters

`s`

Type: _In_ **SOCKET**

A descriptor that identifies the socket.

`lpMsg`

Type: _Inout_ **LPWSAMSG**

A pointer to a **WSAMSG** structure based on the Posix.1g specification for the msghdr structure.

`lpdwNumberOfBytesRecvd`

Type: _Out_opt_ **LPDWORD**

A pointer to a **DWORD** containing number of bytes received by this call if the **WSARecvMsg** operation completes immediately.

To avoid potentially erroneous results, pass **NULL** for this parameter if the *lpOverlapped* parameter is not **NULL**. This parameter can be **NULL** only if the *lpOverlapped* parameter is not **NULL**.

`lpOverlapped`

Type: _Inout_opt_ **LPWSAOVERLAPPED**

A pointer to a **WSAOVERLAPPED** structure. Ignored for non-overlapped structures.

`lpCompletionRoutine`

Type: _In_opt_ **LPWSAOVERLAPPED_COMPLETION_ROUTINE**

A pointer to the completion routine called when the receive operation completes. Ignored for non-overlapped structures.

Return value

If no error occurs and the receive operation has completed immediately, **WSARecvMsg** returns zero. In this case, the completion routine will have already been scheduled to be called once the calling thread is in the alertable state. Otherwise, a value of SOCKET_ERROR is returned, and a specific error code can be retrieved by calling **WSAGetLastError**. The error code **WSA_IO_PENDING** indicates that the overlapped operation has been successfully initiated and that completion will be indicated at a later time.

Any other error code indicates that the operation was not successfully initiated and no completion indication will occur if an overlapped operation was requested.

ERROR CODE	MEANING
WSAECONNRESET	For a UDP datagram socket, this error would indicate that a previous send operation resulted in an ICMP "Port Unreachable" message.
WSAEFAULT	The *lpBuffers*, *lpFlags*, *lpFrom*, *lpNumberOfBytesRecvd*, *lpFromlen*, *lpOverlapped*, or *lpCompletionRoutine* parameter is not totally contained in a valid part of the user address space: the *lpFrom* buffer was too small to accommodate the peer address. This error is also returned if a **name** member of the **WSAMSG** structure pointed to by the *lpMsg* parameter was a **NULL** pointer and the **namelen** member of the **WSAMSG** structure was not set to zero. This error is also returned if a **Control.buf** member of the **WSAMSG** structure pointed to by the *lpMsg* parameter was a **NULL** pointer and the **Control.len** member of the **WSAMSG** structure was not set to zero.
WSAEINPROGRESS	A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function.
WSAEINTR	A blocking Windows Socket 1.1 call was canceled through WSACancelBlockingCall.
WSAEINVAL	The socket has not been bound (with **bind**, for example).
WSAEMSGSIZE	The message was too large to fit into the specified buffer and (for unreliable protocols only) any trailing portion of the message that did not fit into the buffer has been discarded.

ERROR CODE	MEANING
WSAENETDOWN	The network subsystem has failed.
WSAENETRESET	For a datagram socket, this error indicates that the time to live has expired.
WSAENOTCONN	The socket is not connected (connection-oriented sockets only).
WSAETIMEDOUT	The socket timed out. This error is returned if the socket had a wait timeout specified using the **SO_RCVTIMEO** socket option and the timeout was exceeded.
WSAEOPNOTSUPP	The socket operation is not supported. This error is returned if the **dwFlags** member of the WSAMSG structure pointed to by the *lpMsg* parameter includes the **MSG_PEEK** control flag on a non-datagram socket.
WSAEWOULDBLOCK	**Windows NT:** Overlapped sockets: There are too many outstanding overlapped I/O requests. Non-overlapped sockets: The socket is marked as nonblocking and the receive operation cannot be completed immediately.
WSANOTINITIALISED	A successful WSAStartup call must occur before using this function.
WSA_IO_PENDING	An overlapped operation was successfully initiated and completion will be indicated at a later time.
WSA_OPERATION_ABORTED	The overlapped operation has been canceled due to the closure of the socket.

Remarks

The **WSARecvMsg** function can be used in place of the WSARecv and WSARecvFrom functions to receive data and optional control information from connected and unconnected sockets. The **WSARecvMsg** function can only be used with datagrams and raw sockets. The socket descriptor in the *s* parameter must be opened with the socket type set to **SOCK_DGRAM** or **SOCK_RAW**.

Note The function pointer for the **WSARecvMsg** function must be obtained at run time by making a call to the WSAIoctl function with the **SIO_GET_EXTENSION_FUNCTION_POINTER** opcode specified. The input buffer passed to the **WSAIoctl** function must contain **WSAID_WSARECVMSG**, a globally unique identifier (GUID) whose value identifies the **WSARecvMsg** extension function. On success, the output returned by the **WSAIoctl** function contains a pointer to the **WSARecvMsg** function. The **WSAID_WSARECVMSG** GUID is defined in the *Mswsock.h* header file.

The **dwFlags** member of the **WSAMSG** structure pointed to by the *lpMsg* parameter may only contain the **MSG_PEEK** control flag on input.

Overlapped sockets are created with a **WSASocket** function call that has the **WSA_FLAG_OVERLAPPED** flag set. For overlapped sockets, receiving information uses overlapped I/O unless both the *lpOverlapped* and *lpCompletionRoutine* parameters are **NULL**. The socket is treated as a non-overlapped socket when both the *lpOverlapped* and *lpCompletionRoutine* parameters are **NULL**.

A completion indication occurs with overlapped sockets. Once the buffer or buffers have been consumed by the transport, a completion routine is triggered or an event object is set. If the operation does not complete immediately, the final completion status is retrieved through the completion routine or by calling the **WSAGetOverlappedResult** function.

For overlapped sockets, **WSARecvMsg** is used to post one or more buffers into which incoming data will be placed as it becomes available, after which the application-specified completion indication (invocation of the completion routine or setting of an event object) occurs. If the operation does not complete immediately, the final completion status is retrieved through the completion routine or the **WSAGetOverlappedResult** function.

For non-overlapped sockets, the blocking semantics are identical to that of the standard **recv** function and the *lpOverlapped* and *lpCompletionRoutine* parameters are ignored. Any data that has already been received and buffered by the transport will be copied into the specified user buffers. In the case of a blocking socket with no data currently having been received and buffered by the transport, the call will block until data is received. Windows Sockets 2 does not define any standard blocking time-out mechanism for this function. For protocols acting as byte-stream protocols the stack tries to return as much data as possible subject to the available buffer space and amount of received data available. However, receipt of a single byte is sufficient to unblock the caller. There is no guarantee that more than a single byte will be returned. For protocols acting as message-oriented, a full message is required to unblock the caller.

Note The **SO_RCVTIMEO** socket option applies only to blocking sockets.

The buffers are filled in the order in which they appear in the array pointed to by the **lpBuffers** member of the **WSAMSG** structure pointed to by the *lpMsg* parameter, and the buffers are packed so that no holes are created.

If this function is completed in an overlapped manner, it is the Winsock service provider's responsibility to capture this **WSABUF** structure before returning from this call. This enables applications to build stack-based **WSABUF** arrays pointed to by the **lpBuffers** member of the **WSAMSG** structure pointed to by the *lpMsg* parameter.

For message-oriented sockets (a socket type of **SOCK_DGRAM** or **SOCK_RAW**), an incoming message is placed into the buffers up to the total size of the buffers, and the completion indication occurs for overlapped sockets. If the message is larger than the buffers, the buffers are filled with the first part of the message and the excess data is lost, and **WSARecvMsg** generates the error WSAEMSGSIZE.

When the IP_PKTINFO socket option is enabled on an IPv4 socket of type **SOCK_DGRAM** or **SOCK_RAW**, the **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 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.

When the IPV6_PKTINFO socket option is enabled on an IPv6 socket of type **SOCK_DGRAM** or **SOCK_RAW**, the **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 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 **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.

Note All I/O initiated by a given thread is canceled when that thread exits. For overlapped sockets, pending asynchronous operations can fail if the thread is closed before the operations complete. For more information, see **ExitThread**.

Windows Phone 8: This function is supported for Windows Phone Store apps on Windows Phone 8 and later.

Windows 8.1 and **Windows Server 2012 R2**: This function is supported for Windows Store apps on Windows 8.1, Windows Server 2012 R2, and later.

dwFlags

On input, the **dwFlags** member of the **WSAMSG** structure pointed to by the *lpMsg* parameter can be used to influence the behavior of the function invocation beyond the socket options specified for the associated socket. That is, the semantics of this function are determined by the socket options and the **dwFlags** member of the **WSAMSG** structure. The only possible input value for the **dwFlags** member of the **WSAMSG** structure pointed to by the *lpMsg* parameter is **MSG_PEEK**.

VALUE	MEANING
MSG_PEEK	Peek at the incoming data. The data is copied into the buffer, but is not removed from the input queue. This flag is valid only for non-overlapped sockets.

The possible values for **dwFlags** member on input are defined in the *Winsock2.h* header file.

On output, the **dwFlags** member of the **WSAMSG** structure pointed to by the *lpMsg* parameter may return a combination of any of the following values.

VALUE	MEANING
MSG_BCAST	The datagram was received as a link-layer broadcast or with a destination IP address that is a broadcast address.
MSG_CTRUNC	The control (ancillary) data was truncated. More control data was present than the process allocated room for.

VALUE	MEANING
MSG_MCAST	The datagram was received with a destination IP address that is a multicast address.
MSG_TRUNC	The datagram was truncated. More data was present than the process allocated room for.

On the Microsoft Windows Software Development Kit (SDK) released for Windows Vista and later, the organization of header files has changed and the possible values for the **dwFlags** member on output are defined in the *Ws2def.h* header file which is automatically included by the *Winsock2.h* header file.

On versions of the Platform Software Development Kit (SDK) for Windows Server 2003 and earlier, the possible values for the **dwFlags** member on output are defined in the *Mswsock.h* header file.

Note When issuing a blocking Winsock call such as **WSARecvMsg** with the *lpOverlapped* parameter set to NULL, Winsock may need to wait for a network event before the call can complete. Winsock performs an alertable wait in this situation, which can be interrupted by an asynchronous procedure call (APC) scheduled on the same thread. Issuing another blocking Winsock call inside an APC that interrupted an ongoing blocking Winsock call on the same thread will lead to undefined behavior, and must never be attempted by Winsock clients.

Overlapped Socket I/O

If an overlapped operation completes immediately, **WSARecvMsg** returns a value of zero and the *lpNumberOfBytesRecvd* parameter is updated with the number of bytes received and the flag bits indicated by the *lpFlags* parameter are also updated. If the overlapped operation is successfully initiated and will complete later, **WSARecvMsg** returns **SOCKET_ERROR** and indicates error code WSA_IO_PENDING. In this case, *lpNumberOfBytesRecvd* is not updated. When the overlapped operation completes, the amount of data transferred is indicated either through the *cbTransferred* parameter in the completion routine (if specified), or through the *lpcbTransfer* parameter in **WSAGetOverlappedResult**. Flag values are obtained by examining the *lpdwFlags* parameter of **WSAGetOverlappedResult**.

The **WSARecvMsg** function using overlapped I/O can be called from within the completion routine of a previous **WSARecv**, **WSARecvFrom**, **WSARecvMsg**, **WSASend**, **WSASendMsg**, or **WSASendTo** function. For a given socket, I/O completion routines will not be nested. This permits time-sensitive data transmissions to occur entirely within a preemptive context.

The *lpOverlapped* parameter must be valid for the duration of the overlapped operation. If multiple I/O operations are simultaneously outstanding, each must reference a separate **WSAOVERLAPPED** structure.

If the *lpCompletionRoutine* parameter is **NULL**, the *hEvent* parameter of *lpOverlapped* is signaled when the overlapped operation completes if it contains a valid event object handle. An application can use **WSAWaitForMultipleEvents** or **WSAGetOverlappedResult** to wait or poll on the event object.

If *lpCompletionRoutine* is not **NULL**, the *hEvent* parameter is ignored and can be used by the application to pass context information to the completion routine. A caller that passes a non-**NULL** *lpCompletionRoutine* and later calls **WSAGetOverlappedResult** for the same overlapped I/O request may not set the *fWait* parameter for that invocation of **WSAGetOverlappedResult** to **TRUE**. In this case the usage of the *hEvent* parameter is undefined, and attempting to wait on the *hEvent* parameter would produce unpredictable results.

The completion routine follows the same rules as stipulated for Windows file I/O completion routines. The completion routine will not be invoked until the thread is in an alertable wait state such as can occur when the

function **WSAWaitForMultipleEvents** with the *fAlertable* parameter set to **TRUE** is invoked.

The prototype of the completion routine is as follows:

```
void CALLBACK CompletionRoutine(
  IN DWORD dwError,
  IN DWORD cbTransferred,
  IN LPWSAOVERLAPPED lpOverlapped,
  IN DWORD dwFlags
);
```

The **CompletionRoutine** is a placeholder for an application-defined or library-defined function name. The *dwError* parameter specifies the completion status for the overlapped operation as indicated by the *lpOverlapped* parameter. The *cbTransferred* parameter specifies the number of bytes received. The *dwFlags* parameter contains information that is also returned in **dwFlags** member of the **WSAMSG** structure pointed to by the *lpMsg* parameter if the receive operation had completed immediately. The **CompletionRoutine** function does not return a value.

Returning from this function allows invocation of another pending completion routine for this socket. When using **WSAWaitForMultipleEvents**, all waiting completion routines are called before the alertable thread's wait is satisfied with a return code of **WSA_IO_COMPLETION**. The completion routines can be called in any order, not necessarily in the same order the overlapped operations are completed. However, the posted buffers are guaranteed to be filled in the same order in which they are specified.

If you are using I/O completion ports, be aware that the order of calls made to **WSARecvMsg** is also the order in which the buffers are populated. The **WSARecvMsg** function should not be called on the same socket simultaneously from different threads, because it can result in an unpredictable buffer order.

Requirements

Minimum supported client	Windows 10 Build 20348
Minimum supported server	Windows 10 Build 20348
Header	mswsock.h

See also

ExitThread

IP_PKTINFO

IPV6_PKTINFO

Winsock reference

Winsock functions

WSABUF

WSAGetLastError

WSAIoctl

WSAMSG

WSAOVERLAPPED

WSARecv

WSARecvFrom

WSASend

WSASendMsg

WSASendTo

WSAStartup

WSAWaitForMultipleEvents

RIO_EXTENSION_FUNCTION_TABLE structure (mswsock.h)

10/22/2022 • 2 minutes to read • Edit Online

The **RIO_EXTENSION_FUNCTION_TABLE** structure contains information on the functions that implement the Winsock registered I/O extensions.

Syntax

```
typedef struct _RIO_EXTENSION_FUNCTION_TABLE {
  DWORD                          cbSize;
  LPFN_RIORECEIVE                RIOReceive;
  LPFN_RIORECEIVEEX              RIOReceiveEx;
  LPFN_RIOSEND                   RIOSend;
  LPFN_RIOSENDEX                 RIOSendEx;
  LPFN_RIOCLOSECOMPLETIONQUEUE   RIOCloseCompletionQueue;
  LPFN_RIOCREATECOMPLETIONQUEUE  RIOCreateCompletionQueue;
  LPFN_RIOCREATEREQUESTQUEUE     RIOCreateRequestQueue;
  LPFN_RIODEQUEUECOMPLETION      RIODequeueCompletion;
  LPFN_RIODEREGISTERBUFFER       RIODeregisterBuffer;
  LPFN_RIONOTIFY                 RIONotify;
  LPFN_RIOREGISTERBUFFER         RIORegisterBuffer;
  LPFN_RIORESIZECOMPLETIONQUEUE  RIOResizeCompletionQueue;
  LPFN_RIORESIZEREQUESTQUEUE     RIOResizeRequestQueue;
} RIO_EXTENSION_FUNCTION_TABLE, *PRIO_EXTENSION_FUNCTION_TABLE;
```

Members

`cbSize`

The size, in bytes, of the structure.

`RIOReceive`

A pointer to the RIOReceive function.

`RIOReceiveEx`

A pointer to the RIOReceiveEx function.

`RIOSend`

A pointer to the RIOSend function.

`RIOSendEx`

A pointer to the RIOSendEx function.

`RIOCloseCompletionQueue`

A pointer to the RIOCloseCompletionQueue function.

`RIOCreateCompletionQueue`

A pointer to the RIOCreateCompletionQueue function.

`RIOCreateRequestQueue`

A pointer to the RIOCreateRequestQueue function.

`RIODequeueCompletion`

A pointer to the RIODequeueCompletion function.

`RIODeregisterBuffer`

A pointer to the RIODeregisterBuffer function.

`RIONotify`

A pointer to the RIONotify function.

`RIORegisterBuffer`

A pointer to the RIORegisterBuffer function.

`RIOResizeCompletionQueue`

A pointer to the RIOResizeCompletionQueue function.

`RIOResizeRequestQueue`

A pointer to the RIOResizeRequestQueue function.

Remarks

The **RIO_EXTENSION_FUNCTION_TABLE** structure contains information on the functions that implement the Winsock registered I/O extensions.

The function pointers for the Winsock registered I/O extension functions must be obtained at run time by making a call to the WSAIoctl function with the **SIO_GET_MULTIPLE_EXTENSION_FUNCTION_POINTER** opcode specified. The input buffer passed to the **WSAIoctl** function must contain **WSAID_MULTIPLE_RIO**, a globally unique identifier (GUID) whose value identifies the Winsock registered I/O extension functions. On success, the output returned by the **WSAIoctl** function contains a pointer to the **RIO_EXTENSION_FUNCTION_TABLE** structure that contains pointers to the Winsock registered I/O extension functions. The **SIO_GET_MULTIPLE_EXTENSION_FUNCTION_POINTER** IOCTL is defined in the *Ws2def.h* header file.The **WSAID_MULTIPLE_RIO** GUID is defined in the *Mswsock.h* header file.

Requirements

Minimum supported client	Windows 8 [desktop apps only]
Minimum supported server	Windows Server 2012 [desktop apps only]
Header	mswsock.h (include Mswsock.h)

See also

RIOCloseCompletionQueue

RIOCreateCompletionQueue

RIOCreateRequestQueue

RIODequeueCompletion

RIODeregisterBuffer

RIONotify

RIOReceive

RIOReceiveEx

RIORegisterBuffer

RIOResizeCompletionQueue

RIOResizeRequestQueue

RIOSend

RIOSendEx

WSAIoctl

RIO_NOTIFICATION_COMPLETION structure (mswsock.h)

10/22/2022 • 3 minutes to read • Edit Online

The **RIO_NOTIFICATION_COMPLETION** structure specifies the method for I/O completion to be used with a RIONotify function for sending or receiving network data with the Winsock registered I/O extensions.

Syntax

```
typedef struct _RIO_NOTIFICATION_COMPLETION {
  RIO_NOTIFICATION_COMPLETION_TYPE Type;
  union {
    struct {
      HANDLE EventHandle;
      BOOL   NotifyReset;
    } Event;
    struct {
      HANDLE IocpHandle;
      PVOID  CompletionKey;
      PVOID  Overlapped;
    } Iocp;
  };
} RIO_NOTIFICATION_COMPLETION, *PRIO_NOTIFICATION_COMPLETION;
```

Members

`Type`

The type of completion to use with the RIONotify function when sending or receiving data.

`Event`

`Event.EventHandle`

The handle for the event to set following a completed RIONotify request.

This value is valid when the **Type** member is set to **RIO_EVENT_COMPLETION**.

`Event.NotifyReset`

The boolean value that causes the associated event to be reset when the RIONotify function is called. A non-zero value cause the associated event to be reset.

This value is valid when the **Type** member is set to **RIO_EVENT_COMPLETION**.

`Iocp`

`Iocp.IocpHandle`

The handle for the I/O completion port to use for queuing a RIONotify request completion.

This value is valid when the **Type** member is set to **RIO_IOCP_COMPLETION**.

`Iocp.CompletionKey`

The value to use for *lpCompletionKey* parameter returned by the GetQueuedCompletionStatus or

GetQueuedCompletionStatusEx function when queuing a RIONotify request.

This value is valid when the **Type** member is set to **RIO_IOCP_COMPLETION**.

`Iocp.Overlapped`

A pointer to the OVERLAPPED structure to use when queuing a RIONotify request completion. This member must point to a valid **OVERLAPPED** structure.

This value is valid when the **Type** member is set to **RIO_IOCP_COMPLETION**.

Remarks

The **RIO_NOTIFICATION_COMPLETION** structure is used to specify the behavior of the RIONotify function used with the Winsock registered I/O extensions.

The **RIO_NOTIFICATION_COMPLETION** structure is passed to the RIOCreateCompletionQueue function when a RIO_CQ is created. If an application does not call the RIONotify function for a completion queue, the completion queue can be created without a **RIO_NOTIFICATION_COMPLETION** object.

For completion queues using an event, the **Type** member of the **RIO_NOTIFICATION_COMPLETION** structure is set to **RIO_EVENT_COMPLETION**. The **Event.EventHandle** member of the **RIO_NOTIFICATION_COMPLETION** structure 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 of the **RIO_NOTIFICATION_COMPLETION** structure to a non-zero value. This causes the event to be reset by the **RIONotify** function when notification occurs. This mitigates the need to call the WSAResetEvent function to reset the event between calls to the **RIONotify** function.

For completion queues using an I/O completion port, the **Type** member of the **RIO_NOTIFICATION_COMPLETION** structure is set to **RIO_IOCP_COMPLETION**. The **Iocp.IocpHandle** member of the **RIO_NOTIFICATION_COMPLETION** structure 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 **Iocp.CompletionKey** member to distinguish **RIONotify** requests on the completion queue from other I/O completions including **RIONotify** completions for other completion queues.

An application using thread pools can use thread pool wait objects to get RIONotify completions via its thread pool. In that case, the call to the SetThreadpoolWait function should immediately follow the call to **RIONotify**. If the **SetThreadpoolWait** function is called before **RIONotify** and the application relies on **RIONotify** to clear the event object, this may result in spurious executions of the wait object callback.

Requirements

Minimum supported client	Windows 8 [desktop apps only]
Minimum supported server	Windows Server 2012 [desktop apps only]
Header	mswsock.h

See also

CreateEvent

CreateIoCompletionPort

GetQueuedCompletionStatus

GetQueuedCompletionStatusEx

OVERLAPPED

RIOCreateCompletionQueue

RIONotify

RIO_CQ

RIO_NOTIFICATION_COMPLETION_TYPE

SetThreadpoolWait

WSACreateEvent

WSAResetEvent

WSAWaitForMultipleEvents

RIO_NOTIFICATION_COMPLETION_TYPE enumeration (mswsock.h)

10/22/2022 • 2 minutes to read • Edit Online

The **RIO_NOTIFICATION_COMPLETION_TYPE** enumeration specifies the type of completion queue notifications to use with the RIONotify function when sending or receiving data using the Winsock registered I/O extensions.

Syntax

```
typedef enum _RIO_NOTIFICATION_COMPLETION_TYPE {
  RIO_EVENT_COMPLETION = 1,
  RIO_IOCP_COMPLETION = 2
} RIO_NOTIFICATION_COMPLETION_TYPE, *PRIO_NOTIFICATION_COMPLETION_TYPE;
```

Constants

`RIO_EVENT_COMPLETION` Value: *1* An event handle is used to signal completion queue notifications. An event handle is provided as the **EventNotify.EventHandle** member in the RIO_NOTIFICATION_COMPLETION structure passed to the RIOCreateCompletionQueue function when the RIO_CQ is created. The completion of the RIONotify function for this **RIO_CQ** will signal the event. The **Event.NotifyReset** member in the **RIO_NOTIFICATION_COMPLETION** structure passed to the **RIOCreateCompletionQueue** function when the **RIO_CQ** is created indicates whether or not the event should be reset as part of a call to the **RIONotify** function.
`RIO_IOCP_COMPLETION` Value: *2* An I/O completion port handle is used to signal completion queue notifications. An I/O completion port handle is provided as the **Iocp.IocpHandle** member in the RIO_NOTIFICATION_COMPLETION structure passed to the RIOCreateCompletionQueue function when the RIO_CQ is created. The completion of the RIONotify function for this **RIO_CQ** will queue an entry to the I/O completion port which can be retrieved using the GetQueuedCompletionStatus or GetQueuedCompletionStatusEx function. The queued entry will have the returned *lpCompletionKey* parameter value set to the value specified in the **Iocp.CompletionKey** member of the **RIO_NOTIFICATION_COMPLETION** and the returned *lpOverlapped* parameter value set to the value specified in the **Iocp.Overlapped** member in **RIO_NOTIFICATION_COMPLETION** structure. The **Iocp.Overlapped** member in the **RIO_NOTIFICATION_COMPLETION** will be a non-NULL value.

Remarks

The **RIO_NOTIFICATION_COMPLETION_TYPE** enumeration is used with the Winsock registered I/O extensions to specify the type of I/O completion to use with a RIO_CQ. An enumeration value is set in the RIO_NOTIFICATION_COMPLETION structure passed to the RIOCreateCompletionQueue function when the **RIO_CQ** is created.

When creating a RIO_CQ, the RIO_NOTIFICATION_COMPLETION structure determines how the application will receive completion queue notifications. If the **RIO_NOTIFICATION_COMPLETION** structure is provided when

creating the completion queue, the application may call the RIONotify function to request a completion queue notification. Normally this notification occurs when the completion queue is not empty. This may happen immediately or when the next completion entry is inserted into the completion queue. Once a completion queue notification is issued, the application must call **RIONotify** in order to receive another completion queue notification.

Two options are available for completion queue notification.

- Event handles.
- I/O completion ports

If the **Type** member of the RIO_NOTIFICATION_COMPLETION structure is set to **RIO_EVENT_COMPLETION**, an event handle is used to signal completion queue notifications. An event handle is provided as the **EventNotify.EventHandle** member in the **RIO_NOTIFICATION_COMPLETION** structure passed to the RIOCreateCompletionQueue function.

If the **Type** member of the RIO_NOTIFICATION_COMPLETION structure is set to **RIO_IOCP_COMPLETION**, an I/O completion port is used to signal completion queue notifications. An I/O completion port handle is provided as the **Iocp.IocpHandle** member in the **RIO_NOTIFICATION_COMPLETION** structure passed to the RIOCreateCompletionQueue function. The completion of the RIONotify function for this RIO_CQ will queue an entry to the I/O completion port which can be retrieved using the GetQueuedCompletionStatus or GetQueuedCompletionStatusEx function.

Requirements

Minimum supported client	Windows 8 [desktop apps only]
Minimum supported server	Windows Server 2012 [desktop apps only]
Header	mswsock.h

See also

RIOCreateCompletionQueue

RIONotify

RIO_CQ

RIO_NOTIFICATION_COMPLETION

TRANSMIT_FILE_BUFFERS structure (mswsock.h)

10/22/2022 • 2 minutes to read • Edit Online

The **TRANSMIT_FILE_BUFFERS** structure specifies data to be transmitted before and after file data during a TransmitFile function file transfer operation.

Syntax

```
typedef struct _TRANSMIT_FILE_BUFFERS {
  LPVOID Head;
  DWORD  HeadLength;
  LPVOID Tail;
  DWORD  TailLength;
} TRANSMIT_FILE_BUFFERS, *PTRANSMIT_FILE_BUFFERS, *LPTRANSMIT_FILE_BUFFERS;
```

Members

`Head`

Pointer to a buffer that contains data to be transmitted before the file data is transmitted.

`HeadLength`

Size of the buffer pointed to by **Head**, in bytes, to be transmitted.

`Tail`

Pointer to a buffer that contains data to be transmitted after the file data is transmitted.

`TailLength`

Size of the buffer pointed to **Tail**, in bytes, to be transmitted.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	mswsock.h (include Winsock.h)

See also

TransmitFile

TRANSMIT_PACKETS_ELEMENT structure (mswsock.h)

10/22/2022 • 2 minutes to read • Edit Online

The **TRANSMIT_PACKETS_ELEMENT** structure specifies a single data element to be transmitted by the TransmitPackets function.

Syntax

```
typedef struct _TRANSMIT_PACKETS_ELEMENT {
  ULONG dwElFlags;
  ULONG cLength;
  union {
    struct {
      LARGE_INTEGER nFileOffset;
      HANDLE        hFile;
    };
    PVOID pBuffer;
  };
} TRANSMIT_PACKETS_ELEMENT, *PTRANSMIT_PACKETS_ELEMENT, *LPTRANSMIT_PACKETS_ELEMENT;
```

Members

`dwElFlags`

Type: **ULONG**

Flags used to describe the contents of the packet array element, and to customize **TransmitPackets** function processing. The following table lists valid flags:

FLAG	MEANING
TP_ELEMENT_FILE	Specifies that data resides in a file. Default setting for **dwElFlags**. Mutually exclusive with TP_ELEMENT_MEMORY.
TP_ELEMENT_MEMORY	Specifies that data resides in memory. Mutually exclusive with TP_ELEMENT_FILE.
TP_ELEMENT_EOP	Specifies that this element should not be combined with the next element in a single send request from the sockets layer to the transport. This flag is used for granular control of the content of each message on a datagram or message-oriented socket.

`cLength`

Type: **ULONG**

The number of bytes to transmit. If zero, the entire file is transmitted.

`nFileOffset`

Type: **LARGE_INTEGER**

The file offset, in bytes, at which to begin the transfer. Valid only if TP_ELEMENT_FILE is specified in **dwEIFlags**. When set to –1, transmission begins at the current byte offset.

`hFile`

Type: **HANDLE**

A handle to an open file to be transmitted. Valid only if TP_ELEMENT_FILE is specified in **dwEIFlags**. Windows reads the file sequentially; caching performance is improved by opening this handle with FILE_FLAG_SEQUENTIAL_SCAN.

`pBuffer`

Type: **PVOID**

A pointer to the data in memory to be sent. Valid only if TP_ELEMENT_MEMORY is specified in **dwEIFlags**.

Requirements

Minimum supported client	Windows XP [desktop apps only]
Minimum supported server	Windows Server 2003 [desktop apps only]
Header	mswsock.h

See also

TransmitPackets

LPFN_WSARECVMSG (WSARecvMsg)

send

TransmitFile function (mswsock.h)

10/22/2022 • 11 minutes to read • Edit Online

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

> **Note** This function is a Microsoft-specific extension to the Windows Sockets specification.

Syntax

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

Parameters

`hSocket`

A handle to a connected socket. The **TransmitFile** function will transmit the file data over this socket. The socket specified by the *hSocket* parameter must be a connection-oriented socket of type **SOCK_STREAM**, **SOCK_SEQPACKET**, or **SOCK_RDM**.

`hFile`

A handle to the open file that the **TransmitFile** function transmits. Since the operating system reads the file data sequentially, you can improve caching performance by opening the handle with FILE_FLAG_SEQUENTIAL_SCAN.

The *hFile* parameter is optional. If the *hFile* parameter is **NULL**, only data in the header and/or the tail buffer is transmitted. Any additional action, such as socket disconnect or reuse, is performed as specified by the *dwFlags* parameter.

`nNumberOfBytesToWrite`

The number of bytes in the file to transmit. The **TransmitFile** function completes when it has sent the specified number of bytes, or when an error occurs, whichever occurs first.

Set this parameter to zero in order to transmit the entire file.

`nNumberOfBytesPerSend`

The size, in bytes, of each block of data sent in each send operation. This parameter is used by Windows' sockets layer to determine the block size for send operations. To select the default send size, set this parameter to zero.

The *nNumberOfBytesPerSend* parameter is useful for protocols that have limitations on the size of individual

send requests.

`lpOverlapped`

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

You can use the *lpOverlapped* parameter to specify a 64-bit offset within the file at which to start the file data transfer by setting the **Offset** and **OffsetHigh** member of the OVERLAPPED structure. If *lpOverlapped* is a **NULL** pointer, the transmission of data always starts at the current byte offset in the file.

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

`lpTransmitBuffers`

A pointer to a TRANSMIT_FILE_BUFFERS data structure that contains pointers to data to send before and after the file data is sent. This parameter should be set to a **NULL** pointer if you want to transmit only the file data.

`dwReserved`

A set of flags used to modify the behavior of the **TransmitFile** function call. The *dwFlags* parameter can contain a combination of the following options defined in the *Mswsock.h* header file:

FLAG	MEANING
TF_DISCONNECT	Start a transport-level disconnect after all the file data has been queued for transmission.
TF_REUSE_SOCKET	Prepare the socket handle to be reused. This flag is valid only if **TF_DISCONNECT** is also specified. When the **TransmitFile** request completes, the socket handle can be passed to the function call previously used to establish the connection, such as AcceptEx or ConnectEx. Such reuse is mutually exclusive; for example, if the **AcceptEx** function was called for the socket, reuse is allowed only for subsequent calls to the **AcceptEx** function, and not allowed for a subsequent call to **ConnectEx**. **Note** The socket level file transmit is subject to the behavior of the underlying transport. For example, a TCP socket may be subject to the TCP TIME_WAIT state, causing the **TransmitFile** call to be delayed.
TF_USE_DEFAULT_WORKER	Directs the Windows Sockets service provider to use the system's default thread to process long **TransmitFile** requests. The system default thread can be adjusted using the following registry parameter as a **REG_DWORD**: **HKEY_LOCAL_MACHINE\CurrentControlSet\Services\AFD\Parameters\TransmitWorker**

TF_USE_SYSTEM_THREAD	Directs the Windows Sockets service provider to use system threads to process long **TransmitFile** requests.
TF_USE_KERNEL_APC	Directs the driver to use kernel asynchronous procedure calls (APCs) instead of worker threads to process long **TransmitFile** requests. Long **TransmitFile** requests are defined as requests that require more than a single read from the file or a cache; the request therefore depends on the size of the file and the specified length of the send packet. Use of TF_USE_KERNEL_APC can deliver significant performance benefits. It is possible (though unlikely), however, that the thread in which context **TransmitFile** is initiated is being used for heavy computations; this situation may prevent APCs from launching. Note that the Winsock kernel mode driver uses normal kernel APCs, which launch whenever a thread is in a wait state, which differs from user-mode APCs, which launch whenever a thread is in an alertable wait state initiated in user mode).
TF_WRITE_BEHIND	Complete the **TransmitFile** request immediately, without pending. If this flag is specified and **TransmitFile** succeeds, then the data has been accepted by the system but not necessarily acknowledged by the remote end. Do not use this setting with the TF_DISCONNECT and TF_REUSE_SOCKET flags. **Note** If the file being sent is not in the file system cache, the request pends.

Return value

If the **TransmitFile** function succeeds, the return value is **TRUE**. Otherwise, the return value is **FALSE**. To get extended error information, call WSAGetLastError. An error code of WSA_IO_PENDING or ERROR_IO_PENDING indicates that the overlapped operation has been successfully initiated and that completion will be indicated at a later time. Any other error code indicates that the overlapped operation was not successfully initiated and no completion indication will occur. Applications should handle either ERROR_IO_PENDING or WSA_IO_PENDING in this case.

RETURN CODE	DESCRIPTION
WSAECONNABORTED	An established connection was aborted by the software in your host machine. This error is returned if the virtual circuit was terminated due to a time-out or other failure.
WSAECONNRESET	An existing connection was forcibly closed by the remote host. This error is returned for a stream socket when the virtual circuit was reset by the remote side. The application should close the socket as it is no longer usable.

WSAEFAULT	The system detected an invalid pointer address in attempting to use a pointer argument in a call. This error is returned if the *lpTransmitBuffers* or *lpOverlapped* parameter is not totally contained in a valid part of the user address space.
WSAEINVAL	An invalid argument was supplied. This error is returned if the *hSocket* parameter specified a socket of type **SOCK_DGRAM** or **SOCK_RAW**. This error is returned if the *dwFlags* parameter has the **TF_REUSE_SOCKET** flag set, but the **TF_DISCONNECT** flag was not set. This error is also returned if the offset specified in the OVERLAPPED structure pointed to by the *lpOverlapped* is not within the file. This error is also returned if the *nNumberOfBytesToWrite* parameter is set to a value greater than 2,147,483,646, the maximum value for a 32-bit integer minus 1.
WSAENETDOWN	A socket operation encountered a dead network.This error is returned if the network subsystem has failed.
WSAENETRESET	The connection has been broken due to keep-alive activity detecting a failure while the operation was in progress.
WSAENOBUFS	An operation on a socket could not be performed because the system lacked sufficient buffer space or because a queue was full. This error is also returned if the Windows Sockets provider reports a buffer deadlock.
WSAENOTCONN	A request to send or receive data was disallowed because the socket is not connected.
WSAENOTSOCK	An operation was attempted on something that is not a socket. This error is returned if the *hSocket* parameter is not a socket.
WSAESHUTDOWN	A request to send or receive data was disallowed because the socket had already been shut down in that direction with a previous shutdown call. This error is returned if the socket has been shut down for sending. It is not possible to call TransmitFile on a socket after the shutdown function has been called on the socket with the *how* parameter set to **SD_SEND** or **SD_BOTH**.
WSANOTINITIALISED	Either the application has not called the WSAStartup function, or **WSAStartup** failed. A successful **WSAStartup** call must occur before using the TransmitFile function.
WSA_IO_PENDING	An overlapped I/O operation is in progress. This value is returned if an overlapped I/O operation was successfully initiated and indicates that 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 the overlapped operation has been canceled due to the closure of the socket, the execution of the "SIO_FLUSH" command in WSAIoctl, or the thread that initiated the overlapped request exited before the operation completed. **Note** All I/O initiated by a given thread is canceled when that thread exits. For overlapped sockets, pending asynchronous operations can fail if the thread is closed before the asynchronous operations complete. For more information, see ExitThread.

Remarks

The **TransmitFile** function uses the operating system's cache manager to retrieve the file data, and provide high-performance file data transfer over sockets.

The **TransmitFile** function only supports connection-oriented sockets of type **SOCK_STREAM**, **SOCK_SEQPACKET**, and **SOCK_RDM**. Sockets of type **SOCK_DGRAM** and **SOCK_RAW** are not supported. The TransmitPackets function can be used with sockets of type **SOCK_DGRAM**.

The maximum number of bytes that can be transmitted using a single call to the **TransmitFile** function is 2,147,483,646, the maximum value for a 32-bit integer minus 1. The maximum number of bytes to send in a single call includes any data sent before or after the file data pointed to by the *lpTransmitBuffers* parameter plus the value specified in the *nNumberOfBytesToWrite* parameter for the length of file data to send. If an application needs to transmit a file larger than 2,147,483,646 bytes, then multiple calls to the **TransmitFile** function can be used with each call transferring no more than 2,147,483,646 bytes. Setting the *nNumberOfBytesToWrite* parameter to zero for a file larger than 2,147,483,646 bytes will also fail since in this case the **TransmitFile** function will use the size of the file as the value for the number of bytes to transmit.

> **Note** The function pointer for the **TransmitFile** function must be obtained at run time by making a call to the WSAIoctl function with the **SIO_GET_EXTENSION_FUNCTION_POINTER** opcode specified. The input buffer passed to the **WSAIoctl** function must contain **WSAID_TRANSMITFILE**, a globally unique identifier (GUID) whose value identifies the **TransmitFile** extension function. On success, the output returned by the **WSAIoctl** function contains a pointer to the **TransmitFile** function. The **WSAID_TRANSMITFILE** GUID is defined in the *Mswsock.h* header file.

> **Note** **TransmitFile** is not functional on transports that perform their own buffering. Transports with the TDI_SERVICE_INTERNAL_BUFFERING flag set, such as ADSP, perform their own buffering. Because **TransmitFile** achieves its performance gains by sending data directly from the file cache. Transports that run out of buffer space on a particular connection are not handled by **TransmitFile**, and as a result of running out of buffer space on the connection, **TransmitFile** returns STATUS_DEVICE_NOT_READY.

The **TransmitFile** function was primarily added to Winsock for use by high-performance server applications (web and ftp servers, for example).

Workstation and client versions of Windows optimize the **TransmitFile** function for minimum memory and resource utilization by limiting the number of concurrent **TransmitFile** operations allowed on the system to a maximum of two. On Windows Vista, Windows XP, Windows 2000 Professional, and Windows NT Workstation 3.51 and later only two outstanding **TransmitFile** requests are handled simultaneously; the third request will wait until one of the previous requests is completed.

Server versions of Windows optimize the **TransmitFile** function for high performance. On server versions, there are no default limits placed on the number of concurrent **TransmitFile** operations allowed on the system. Expect better performance results when using **TransmitFile** on server versions of Windows. On server versions of Windows, it is possible to set a limit on the maximum number of concurrent **TransmitFile** operations by creating a registry entry and setting a value for the following **REG_DWORD**:

HKEY_LOCAL_MACHINE\CurrentControlSet\Services\AFD\Parameters\MaxActiveTransmitFileCount

If the **TransmitFile** function is called with TCP socket (protocol of IPPROTO_TCP) with both the **TF_DISCONNECT** and **TF_REUSE_SOCKET** flags specified, the call will not complete until the two following conditions are met.

- All pending receive data sent by remote side (received prior to a FIN from the remote side) on the TCP socket has been read.
- The remote side has closed the connection (completed the graceful TCP connection closure).

If the **TransmitFile** function is called with the *lpOverlapped* parameter set to **NULL**, the operation is executed as synchronous I/O. The function will not complete until the file has been sent.

Windows Phone 8: This function is supported for Windows Phone Store apps on Windows Phone 8 and later.

Windows 8.1 and **Windows Server 2012 R2**: This function is supported for Windows Store apps on Windows 8.1, Windows Server 2012 R2, and later.

Notes for QoS

The **TransmitFile** function allows the setting of two flags, TF_DISCONNECT or TF_REUSE_SOCKET, that return the socket to a "disconnected, reusable" state after the file has been transmitted. These flags should not be used on a socket where quality of service has been requested, since the service provider may immediately delete any quality of service associated with the socket before the file transfer has completed. The best approach for a QoS-enabled socket is to simply call the closesocket function when the file transfer has completed, rather than relying on these flags.

Requirements

Minimum supported client	Windows 8.1, Windows Vista [desktop apps | UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps | UWP apps]
Target Platform	Windows
Header	mswsock.h (include Mswsock.h)
Library	Mswsock.lib
DLL	Mswsock.dll

See also

ExitThread

OVERLAPPED

TRANSMIT_FILE_BUFFERS

TransmitPackets

WSASend

closesocket

WSARecvEx function (mswsock.h)

10/22/2022 • 5 minutes to read • Edit Online

The **WSARecvEx** function receives data from a connected socket or a bound connectionless socket. The **WSARecvEx** function is similar to the recv function, except that the *flags* parameter is used only to return information. When a partial message is received while using datagram protocol, the MSG_PARTIAL bit is set in the *flags* parameter on return from the function.

> **Note** The **WSARecvEx** function is a Microsoft-specific extension to the Windows Sockets specification.

Syntax

```
int WSARecvEx(
  [in]      SOCKET s,
  [out]     char   *buf,
  [in]      int    len,
  [in, out] int    *flags
);
```

Parameters

`[in] s`

A descriptor that identifies a connected socket.

`[out] buf`

A pointer to the buffer to receive the incoming data.

`[in] len`

The length, in bytes, of the buffer pointed to by the *buf* parameter.

`[in, out] flags`

An indicator specifying whether the message is fully or partially received for datagram sockets.

Return value

If no error occurs, **WSARecvEx** returns the number of bytes received. If the connection has been closed, it returns zero. Additionally, if a partial message was received, the MSG_PARTIAL bit is set in the *flags* parameter. If a complete message was received, MSG_PARTIAL is not set in *flags*

Otherwise, a value of SOCKET_ERROR is returned, and a specific error code can be retrieved by calling WSAGetLastError.

> **Important** For a stream oriented-transport protocol, MSG_PARTIAL is never set on return from **WSARecvEx**. This function behaves identically to the recv function for stream-transport protocols.

ERROR CODE	MEANING
WSAECONNABORTED	The virtual circuit was terminated due to a time-out or other failure. The application should close the socket as it is no longer usable.
WSAECONNRESET	The virtual circuit was reset by the remote side executing a hard or abortive close. The application should close the socket as it is no longer usable. On a UPD-datagram socket this error would indicate that a previous send operation resulted in an ICMP "Port Unreachable" message.
WSAEFAULT	The *buf* parameter is not completely contained in a valid part of the user address space.
WSAEINPROGRESS	A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function.
WSAEINTR	The (blocking) call was canceled by the WSACancelBlockingCall call.
WSAEINVAL	The socket has not been bound with bind, or an unknown flag was specified, or MSG_OOB was specified for a socket with SO_OOBINLINE enabled or (for byte stream sockets only) *len* was zero or negative.
WSAENETDOWN	The network subsystem has failed.
WSAENETRESET	For a connection-oriented socket, this error indicates that the connection has been broken due to *keep-alive* activity that detected a failure while the operation was in progress. For a datagram socket, this error indicates that the time to live has expired.
WSAENOTCONN	The socket is not connected.
WSAENOTSOCK	The descriptor is not a socket.
WSAEOPNOTSUPP	MSG_OOB was specified, but the socket is not stream-style such as type SOCK_STREAM, OOB data is not supported in the communication domain associated with this socket, or the socket is unidirectional and supports only send operations.
WSAESHUTDOWN	The socket has been shut down; it is not possible to use WSARecvEx on a socket after shutdown has been invoked with *how* set to SD_RECEIVE or SD_BOTH.
WSAETIMEDOUT	The connection has been dropped because of a network failure or because the peer system failed to respond.

WSAEWOULDBLOCK	The socket is marked as nonblocking and the receive operation would block.
WSANOTINITIALISED	A successful WSAStartup call must occur before using this function.

Remarks

The **WSARecvEx** function that is part of the Microsoft implementation of Windows Sockets 2 is similar to the more common recv function except that the *flags* parameter is used for a single specific purpose. The *flags* parameter is used to indicate whether a partial or complete message is received when a message-oriented protocol is being used.

The value pointed to by the *flags* parameter is ignored on input. So no flags can be passed to the **WSARecvEx** function to modify its behavior. The value pointed to by the *flags* parameter is set on output. This differs from the recv and WSARecv functions where the value pointed to by the *flags* parameter on input can modify the behavior of the function.

The **WSARecvEx** and recv functions behave identically for stream-oriented protocols.

The *flags* parameter accommodates two common situations in which a partial message will be received:

- When the application's data buffer size is smaller than the message size and the message coincidentally arrives in two pieces.
- When the message is rather large and must arrive in several pieces.

The MSG_PARTIAL bit is set in the value pointed to by the *flags* parameter on return from **WSARecvEx** when a partial message was received. If a complete message was received, MSG_PARTIAL is not set in the value pointed to by the *flags* parameter.

The recv function is different from the **WSARecvEx** and WSARecv functions in that the **recv** function always receives a single message for each call for message-oriented transport protocols. The **recv** function also does not have a means to indicate to the application that the data received is only a partial message. An application must build its own protocol for checking whether a message is partial or complete by checking for the error code WSAEMSGSIZE after each call to **recv**. When the application buffer is smaller than the data being sent, as much of the message as will fit is copied into the user's buffer and **recv** returns with the error code WSAEMSGSIZE. A subsequent call to **recv** will get the next part of the message.

Applications written for message-oriented transport protocols should be coded for this possibility if message sizing is not guaranteed by the application's data transfer protocol. An application can use recv and manage the protocol itself. Alternatively, an application can use **WSARecvEx** and check that the MSG_PARTIAL bit is set in the *flags* parameter.

The **WSARecvEx** function provides the developer with a more effective way of checking whether a message received is partial or complete when a very large message arrives incrementally. For example, if an application sends a one-megabyte message, the transport protocol must break up the message in order to send it over the physical network. It is theoretically possible for the transport protocol on the receiving side to buffer all the data in the message, but this would be quite expensive in terms of resources. Instead, **WSARecvEx** can be used, minimizing overhead and eliminating the need for an application-based protocol.

> **Note** All I/O initiated by a given thread is canceled when that thread exits. For overlapped sockets, pending asynchronous operations can fail if the thread is closed before the operations complete. See the ExitThread function for more information.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	mswsock.h (include Mswsock.h)
Library	Mswsock.lib
DLL	Mswsock.dll

See also

WSAAsyncSelect

WSARecv

Winsock Functions

Winsock Reference

recv

recvfrom

select

send

socket

mswsockdef.h header

10/22/2022 • 2 minutes to read • Edit Online

This header is used by Windows Sockets 2. For more information, see:

- Windows Sockets 2

mswsockdef.h contains the following programming interfaces:

Structures

RIO_BUF Specifies a portion of a registered buffer used for sending or receiving network data with the Winsock registered I/O extensions.
RIORESULT Contains data used to indicate request completion results used with the Winsock registered I/O extensions.

RIO_BUF structure (mswsockdef.h)

10/22/2022 • 2 minutes to read • Edit Online

The **RIO_BUF** structure specifies a portion of a registered buffer used for sending or receiving network data with the Winsock registered I/O extensions.

Syntax

```
typedef struct _RIO_BUF {
  RIO_BUFFERID BufferId;
  ULONG        Offset;
  ULONG        Length;
} RIO_BUF, *PRIO_BUF;
```

Members

`BufferId`

The registered buffer descriptor for a Winsock registered I/O buffer used with send and receive requests.

`Offset`

The offset, in bytes, into the buffer specified by the **BufferId** member. An **Offset** value of zero points to the beginning of the buffer

`Length`

A length, in bytes, of the buffer to use from the **Offset** member.

Remarks

The Winsock registered I/O extensions often operate on portions of registered buffers sometimes called buffer slices. The **RIO_BUF** structure is used by an application that needs to use a small amount of registered memory for sending or receiving network data. The application can often increase performance by registering one large buffer and then using small chunks of the buffer as needed. The **RIO_BUF** structure may describe any contiguous segment of memory contained in a single buffer registration.

A pointer to a **RIO_BUF** structure is passed as the *pData* parameter to the RIOSend, RIOSendEx, RIOReceive, and RIOReceiveEx functions to send or receive network data.

An application cannot resize a registered buffer simply by using a buffer slice with values larger than the original buffer that was registered using the RIORegisterBuffer function.

The **RIO_BUF** structure 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

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

RIODeregisterBuffer

RIOReceive

RIOReceiveEx

RIORegisterBuffer

RIOSend

RIOSendEx

RIO_BUFFERID

RIORESULT structure (mswsockdef.h)

10/22/2022 • 2 minutes to read • Edit Online

The **RIORESULT** structure contains data used to indicate request completion results used with the Winsock registered I/O extensions.

Syntax

```
typedef struct _RIORESULT {
  LONG      Status;
  ULONG     BytesTransferred;
  ULONGLONG SocketContext;
  ULONGLONG RequestContext;
} RIORESULT, *PRIORESULT;
```

Members

`Status`

The completion status of the Winsock registered I/O request.

`BytesTransferred`

The number of bytes sent or received in the I/O request.

`SocketContext`

An application-provided context specified in call to the RIOCreateRequestQueue function.

`RequestContext`

An application-provided context specified with the registered I/O request to the RIOReceive, RIOReceiveEx, RIOSend, and RIOSendEx functions.

Remarks

The **RIORESULT** structure defines the data format used to indicate request completion by the Winsock registered I/O extensions. An application requests completion indications by allocating an array of **RIORESULT** structures and passing the array of **RIORESULT** structures to the RIODequeueCompletion function along with the element count. The application need not perform any initialization of the **RIORESULT** structure elements before calling the **RIODequeueCompletion** function.

The **SocketContext** member of the **RIORESULT** structure can be used by an application to identify the RIO_CQ object or the associated application object on which the Winsock registered I/O request was issued. The **RequestContext** member of the **RIORESULT** structure can similarly be used to identify the particular Winsock registered I/O request that was completed.

The **RIORESULT** structure 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

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

RIODequeueCompletion

RIOReceive

RIOReceiveEx

RIOSend

RIOSendEx

nsemail.h header

10/22/2022 • 2 minutes to read • Edit Online

This header is used by Windows Sockets 2. For more information, see:

- Windows Sockets 2

nsemail.h contains the following programming interfaces:

Structures

NAPI_DOMAIN_DESCRIPTION_BLOB Describes a domain handled by a namespace provider for the NS_EMAIL namespace.
NAPI_PROVIDER_INSTALLATION_BLOB Contains the information required to install a namespace provider for the NS_EMAIL namespace.

Enumerations

NAPI_PROVIDER_LEVEL Specifies the provider authority level of a NS_EMAIL namespace provider for a given domain.
NAPI_PROVIDER_TYPE Specifies the type of hosting expected for a namespace provider.

NAPI_DOMAIN_DESCRIPTION_BLOB structure (nsemail.h)

10/22/2022 • 2 minutes to read • Edit Online

The **NAPI_DOMAIN_DESCRIPTION_BLOB** structure describes a domain handled by a namespace provider for the NS_EMAIL namespace.

Syntax

```
typedef struct napi_domain_description_blob_tag {
  DWORD AuthLevel;
  DWORD cchDomainName;
  DWORD OffsetNextDomainDescription;
  DWORD OffsetThisDomainName;
} NAPI_DOMAIN_DESCRIPTION_BLOB;
```

Members

`AuthLevel`

The authority level of the namespace provider for this domain. This member can be one of the values from the NAPI_PROVIDER_LEVEL enumeration type defined in the *Nsemail.h* header file.

`cchDomainName`

The length, in Unicode characters, of the Unicode string that contains the domain name represented by the **OffsetThisDomainName** member. The **NULL** terminator is not counted when calculating the length.

`OffsetNextDomainDescription`

The offset, in bytes, to the next **NAPI_DOMAIN_DESCRIPTION_BLOB** structure in the NAPI_PROVIDER_INSTALLATION_BLOB structure.

`OffsetThisDomainName`

The offset, in bytes, to a Unicode string that contains a domain name handled by this namespace provider for the NS_EMAIL namespace. The domain name must be at least **cchDomainName** Unicode characters in length. **NULL**-termination of the Unicode string that contains the domain name is recommended, but not required. This offset must be aligned on a minimum of a two-byte boundary.

Remarks

This structure is supported on Windows Vista and later.

The **NAPI_DOMAIN_DESCRIPTION_BLOB** structure describes a domain handled by a namespace provider for the NS_EMAIL namespace. A typical domain name represented by the **OffsetThisDomainName** member in this structure might be msn.com or yahoo.com.

Each namespace provider registered in the NS_EMAIL namespace can support multiple domains. The list of supported domains is specified in the provider registration blob as a list of **NAPI_DOMAIN_DESCRIPTION_BLOB** structures. Each supported domain specification contains a NAPI_PROVIDER_LEVEL value in the **AuthLevel** member of the **NAPI_DOMAIN_DESCRIPTION_BLOB** that

describes the type of support provided by the provider for that domain.

The **NAPI_DOMAIN_DESCRIPTION_BLOB** structure is a member of the NAPI_PROVIDER_INSTALLATION_BLOB structure used to describe and register a NS_EMAIL namespace provider. There may be multiple **NAPI_DOMAIN_DESCRIPTION_BLOB** structures in the **NAPI_PROVIDER_INSTALLATION_BLOB** structure for an NS_EMAIL namespace provider.

The WSCInstallNameSpaceEx and WSCInstallNameSpaceEx32 functions are used to install a namespace provider for the NS_EMAIL namespace using a NAPI_PROVIDER_INSTALLATION_BLOB structure.

The WSAEnumNameSpaceProvidersEx and WSCEnumNameSpaceProvidersEx32 functions are used to enumerate all namespace providers (including NS_EMAIL namespace providers) and to retrieve the NAPI_PROVIDER_INSTALLATION_BLOB structure for a provider if the provider registered a blob upon installation.

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Header	nsemail.h

See also

NAPI_PROVIDER_INSTALLATION_BLOB

NAPI_PROVIDER_LEVEL

WSAEnumNameSpaceProvidersEx

WSCEnumNameSpaceProvidersEx32

WSCInstallNameSpaceEx

WSCInstallNameSpaceEx32

NAPI_PROVIDER_INSTALLATION_BLOB structure (nsemail.h)

10/22/2022 • 3 minutes to read • Edit Online

The **NAPI_PROVIDER_INSTALLATION_BLOB** structure contains the information required to install a namespace provider for the NS_EMAIL namespace.

Syntax

```
typedef struct napi_provider_installation_blob_tag {
  DWORD dwVersion;
  DWORD dwProviderType;
  DWORD fSupportsWildCard;
  ULONG cDomains;
  ULONG OffsetFirstDomain;
} NAPI_PROVIDER_INSTALLATION_BLOB;
```

Members

`dwVersion`

Type: **DWORD**

The version number of the NS_EMAIL namespace provider. This member is specific to the namespace provider.

`dwProviderType`

Type: **DWORD**

The type of namespace provider for the NS_EMAIL namespace. This member can be one of the values from the NAPI_PROVIDER_TYPE enumeration type defined in the *Nsemail.h* header file.

`fSupportsWildCard`

Type: **DWORD**

A Boolean value that indicates if this NS_EMAIL namespace provider supports wildcard names. If this member is nonzero, then an NS_EMAIL provider claims to be potentially able to resolve or register any name that does not belong to any domains the provider is specifically registered for as primary or secondary. If this member is nonzero, then the NS_EMAIL provider may be called to resolve or register any address, if no primary or secondary provider for the domain is available.

There may be multiple providers that claim to be able to resolve any address (the **fSupportsWildCard** set to nonzero). If there are namespace providers with this value that also registered as a wildcard provider, the providers are called in the order that they appear in the Winsock namespace catalog.

`cDomains`

Type: **DWORD**

The number of NAPI_DOMAIN_DESCRIPTION_BLOB structures the starting at the **OffsetFirstDomain** member used to describe domains that are supported by this NS_EMAIL namespace provider.

`OffsetFirstDomain`

Type: **DWORD**

The offset, in bytes, to the first of multiple NAPI_DOMAIN_DESCRIPTION_BLOB structures used to describe domains that are supported by this NS_EMAIL namespace provider. This offset must be aligned on a minimum of a four-byte boundary.

Remarks

This structure is supported on Windows Vista and later.

The **NAPI_PROVIDER_INSTALLATION_BLOB** structure contains the information required to install a namespace provider for the NS_EMAIL namespace. There may be multiple namespace providers for the NS_EMAIL namespace install on a local system.

Each namespace provider registered in the NS_EMAIL namespace can support multiple domains. As a result, there may be multiple NAPI_DOMAIN_DESCRIPTION_BLOB structures in the **NAPI_PROVIDER_INSTALLATION_BLOB** structure for an NS_EMAIL namespace provider. The list of supported domains is specified in the provider registration blob as a list of **NAPI_DOMAIN_DESCRIPTION_BLOB** structures. Each supported domain specification contains a NAPI_PROVIDER_LEVEL value in the **AuthLevel** member of the **NAPI_DOMAIN_DESCRIPTION_BLOB** that describes the level of authority provided by the provider for that domain.

Namespace providers are called in the following order to resolve or register an address in a domain. If a namespace provider registered as the primary provider for the domain, then this primary provider is called first. There are two cases depending on whether authoritative results are requested in the namespace query. The default for a query is to request authoritative results.

When authoritative results are requested in the query, then namespace providers are called as follows. If the primary provider is unavailable or is unable to resolve or register the address, then the first secondary provider in the Winsock catalog is called. If the secondary provider is unavailable or is unable to resolve or register the address, then the next secondary provider in the Winsock catalog is called. If all of the secondary providers are unavailable or are unable to resolve or register the address, then the first wildcard provider in the Winsock catalog is called. If the first wildcard provider is unavailable or is unable to resolve or register the address, then the next wildcard provider in the Winsock catalog is called.

When non-authoritative results are requested in the query, then namespace providers are called as follows. The primary provider, all secondary providers, and all wildcard providers are called and results from all of the queries are returned. The primary provider is called first. Secondary providers are called next, based on the order in the Winsock catalog. Wildcard providers are called next, based on the order in the Winsock catalog. The results that are returned are based on the order of the queries.

The WSCInstallNameSpaceEx and WSCInstallNameSpaceEx32 functions are used to install a namespace provider for the NS_EMAIL namespace using a **NAPI_PROVIDER_INSTALLATION_BLOB** structure.

The WSAEnumNameSpaceProvidersEx and WSCEnumNameSpaceProvidersEx32 functions are used to enumerate all namespace providers (including NS_EMAIL namespace providers) and to retrieve the **NAPI_PROVIDER_INSTALLATION_BLOB** structure for a provider if the provider registered a blob upon installation.

Requirements

Minimum supported client	Windows Vista [desktop apps only]

Minimum supported server	Windows Server 2008 [desktop apps only]
Header	nsemail.h

See also

NAPI_DOMAIN_DESCRIPTION_BLOB

NAPI_PROVIDER_LEVEL

NAPI_PROVIDER_TYPE

WSAAdvertiseProvider

WSAEnumNameSpaceProvidersEx

WSCEnumNameSpaceProvidersEx32

WSCInstallNameSpaceEx

WSCInstallNameSpaceEx32

NAPI_PROVIDER_LEVEL enumeration (nsemail.h)

10/22/2022 • 4 minutes to read • Edit Online

The **NAPI_PROVIDER_LEVEL** enumeration specifies the provider authority level of a NS_EMAIL namespace provider for a given domain.

Syntax

```
typedef enum napi_provider_level_tag {
  ProviderLevel_None = 0,
  ProviderLevel_Secondary,
  ProviderLevel_Primary
} NAPI_PROVIDER_LEVEL;
```

Constants

`ProviderLevel_None` Value: *0* The namespace provider does not support the current domain. This value can be used to temporarily turn off the support for a domain without removing it from the list of supported domains. If **ProviderLevel_None** is set in the **AuthLevel** member of the NAPI_DOMAIN_DESCRIPTION_BLOB for a given domain when the provider is installed and registered, the namespace provider will not be called to resolve or register an address in that domain unless the provider registered as a wildcard provider. There may be multiple NS_EMAIL namespace providers for a domain with a value of **ProviderLevel_None**. If there are namespace providers with this value that also registered as a wildcard provider, the providers are called in the order that they appear in the Winsock catalog.
`ProviderLevel_Secondary` The namespace provider is a secondary provider for a domain in the NS_EMAIL namespace. A namespace provider can be a secondary provider in the target domain if the provider can resolve and register NS_EMAIL names for this domain and give the same answer that a primary provider would provide. If **ProviderLevel_Secondary** is set in **AuthLevel** member of the NAPI_DOMAIN_DESCRIPTION_BLOB for a given domain when the provider is installed and registered, this provider is called when a primary provider for the domain is not currently available or the primary provider could not resolve or register the address in that domain. There may be multiple secondary NS_EMAIL namespace providers for a domain with a value of **ProviderLevel_Secondary**. If there are multiple secondary namespace providers, the providers are called in the order that they appear in the Winsock catalog.

`ProviderLevel_Primary`

The namespace provider is the primary provider for a domain in the NS_EMAIL namespace. A namespace provider can claim to be the primary provider for a domain if it owns all of the NS_EMAIL names for that domain and thus has access to the master data for all such names.

There should be only a single primary NS_EMAIL namespace provider for a domain registered on the local system.

> **Note** There should never be two NS_EMAIL namespace providers that claim to be the primary provider for the same domain. If multiple providers try to register as the primary provider for the same domain, the first provider found in the Winsock namespace catalog for the domain as the primary provider will be called. All other provider claims to be the primary provider are ignored.

Remarks

This enumeration is supported on Windows Vista and later.

The **NAPI_PROVIDER_LEVEL** enumeration is used by the NAPI_DOMAIN_DESCRIPTION_BLOB structure to specify the authority level of a NS_EMAIL namespace provider for a domain. Each namespace provider registered in the NS_EMAIL namespace can support multiple domains. The list of supported domains is specified in the provider registration blob as a list of **NAPI_DOMAIN_DESCRIPTION_BLOB** structures. Each supported domain specification contains a **NAPI_PROVIDER_LEVEL** value in the **AuthLevel** member of the **NAPI_DOMAIN_DESCRIPTION_BLOB** that describes the type of support provided by the provider for that domain.

In addition to the specified domain, a NS_EMAIL namespace provider can also register as a wildcard provider to try and support any domain, by specifying the **fSupportsWildCard** member as nonzero in the NAPI_PROVIDER_INSTALLATION_BLOB passed when the provider is installed.

Namespace providers are called in the following order to resolve or register an address in a domain. If a namespace provider registered as the primary provider for the domain, then this primary provider is called first. There are two cases depending on whether authoritative results are requested in the namespace query. The default for a query is to request authoritative results.

When authoritative results are requested in the query, then namespace providers are called as follows. If the primary provider is unavailable or is unable to resolve or register the address, then the first secondary provider in the Winsock catalog is called. If the secondary provider is unavailable or is unable to resolve or register the address, then the next secondary provider in the Winsock catalog is called. If all of the secondary providers are unavailable or are unable to resolve or register the address, then the first wildcard provider in the Winsock catalog is called. If the first wildcard provider is unavailable or is unable to resolve or register the address, then the next wildcard provider in the Winsock catalog is called.

When non-authoritative results are requested in the query, then namespace providers are called as follows. The primary provider, all secondary providers, and all wildcard providers are called and results from all of the queries are returned. The primary provider is called first. Secondary providers are called next, based on the order in the Winsock catalog. Wildcard providers are called next, based on the order in the Winsock catalog. The results that are returned are based on the order of the queries.

The NAPI_DOMAIN_DESCRIPTION_BLOB structure is used in the NAPI_PROVIDER_INSTALLATION_BLOB structure to describe a NS_EMAIL namespace provider.

The WSCInstallNameSpaceEx and WSCInstallNameSpaceEx32 functions are used to install a namespace

provider for the NS_EMAIL namespace using a NAPI_PROVIDER_INSTALLATION_BLOB structure.

The WSAEnumNameSpaceProvidersEx and WSCEnumNameSpaceProvidersEx32 functions are used to enumerate namespace providers for the NS_EMAIL namespace and retrieve the NAPI_PROVIDER_INSTALLATION_BLOB structure for a provider.

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Header	nsemail.h

See also

NAPI_DOMAIN_DESCRIPTION_BLOB

NAPI_PROVIDER_INSTALLATION_BLOB

WSAEnumNameSpaceProvidersEx

WSCEnumNameSpaceProvidersEx32

WSCInstallNameSpaceEx

WSCInstallNameSpaceEx32

NAPI_PROVIDER_TYPE enumeration (nsemail.h)

10/22/2022 • 2 minutes to read • Edit Online

The **NAPI_PROVIDER_TYPE** enumeration specifies the type of hosting expected for a namespace provider.

Syntax

```
typedef enum napi_provider_type_tag {
  ProviderType_Application = 1,
  ProviderType_Service
} NAPI_PROVIDER_TYPE;
```

Constants

`ProviderType_Application` Value: *1* The namespace provider is expected to be hosted by an application. There may be multiple namespace providers of type **ProviderType_Application** running at the same time on a local system. There may also be multiple instances of the same namespace provider running at the same time on a local system as long as the following conditions are met. Only one instance of the same namespace provider application can be running in a single user session at the same time on the local system. The Windows Sockets infrastructure will select the particular target instance of a namespace provider based on the identity of the client and the user session where it is running. Clients running as user **MyUser** in a user session will only be able to contact an instance of the same namespace provider running as **MyUser** in the same session.
`ProviderType_Service` The namespace provider is expected to be hosted by a service. This hosting model is not currently supported.

Remarks

This enumeration is supported on Windows Vista and later.

On Windows Vista and Windows Server 2008, the **NAPI_PROVIDER_TYPE** enumeration applies only to NS_EMAIL namespace providers. Windows Vista and Windows Server 2008 currently support only namespace providers of type **ProviderType_Application** providers. On Windows Vista and Windows Server 2008, this value should always be set to **ProviderType_Application**.

The **NAPI_PROVIDER_TYPE** enumeration is used by the NAPI_PROVIDER_INSTALLATION_BLOB structure to specify the provide type of an NS_EMAIL namespace provider. Examples of a NS_EMAIL namespace provider of type **ProviderType_Application** would be instant messaging or email clients. An example of a NS_EMAIL namespace provider of type **ProviderType_Service** would be the Peer Name Resolution Protocol (PNRP) namespace provider.

The WSCInstallNameSpaceEx and WSCInstallNameSpaceEx32 functions are used to install a namespace provider for the NS_EMAIL namespace using a NAPI_PROVIDER_INSTALLATION_BLOB structure.

The WSAEnumNameSpaceProvidersEx and WSCEnumNameSpaceProvidersEx32 functions are used to enumerate namespace providers for the NS_EMAIL namespace and retrieve the

NAPI_PROVIDER_INSTALLATION_BLOB structure for a provider.

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Header	nsemail.h

See also

NAPI_PROVIDER_INSTALLATION_BLOB

WSAEnumNameSpaceProvidersEx

WSCEnumNameSpaceProvidersEx32

WSCInstallNameSpaceEx

WSCInstallNameSpaceEx32

nspapi.h header

10/22/2022 • 2 minutes to read • Edit Online

This header is used by Windows Sockets 2. For more information, see:

- Windows Sockets 2

nspapi.h contains the following programming interfaces:

Functions

EnumProtocolsA The EnumProtocols function retrieves information about a specified set of network protocols that are active on a local host. (ANSI)
EnumProtocolsW The EnumProtocols function retrieves information about a specified set of network protocols that are active on a local host. (Unicode)
GetAddressByNameA GetAddressByName is no longer available for use as of Windows Sockets 2. (ANSI)
GetAddressByNameW GetAddressByName is no longer available for use as of Windows Sockets 2. (Unicode)
GetNameByTypeA The GetNameByType function retrieves the name of a network service for the specified service type. (ANSI)
GetNameByTypeW The GetNameByType function retrieves the name of a network service for the specified service type. (Unicode)
GetServiceA The GetService function retrieves information about a network service in the context of a set of default namespaces or a specified namespace. (ANSI)
GetServiceW The GetService function retrieves information about a network service in the context of a set of default namespaces or a specified namespace. (Unicode)
GetTypeByNameA The GetTypeByName function retrieves a service type GUID for a network service specified by name. (ANSI)

GetTypeByNameW

The GetTypeByName function retrieves a service type GUID for a network service specified by name. (Unicode)

SetServiceA

The SetService function registers or removes from the registry a network service within one or more namespaces. (ANSI)

SetServiceW

The SetService function registers or removes from the registry a network service within one or more namespaces. (Unicode)

Structures

BLOB

The BLOB structure (nspapi.h), which is derived from Binary Large Object, contains information about a block of data.

CSADDR_INFO

The CSADDR_INFO structure (nspapi.h) contains Windows Sockets address information for a socket, network service, or namespace provider.

NS_SERVICE_INFOA

Contains information about a network service or a network service type in the context of a specified namespace, or a set of default namespaces. (ANSI)

NS_SERVICE_INFOW

Contains information about a network service or a network service type in the context of a specified namespace, or a set of default namespaces. (Unicode)

PROTOCOL_INFOA

Contains information about a protocol. (ANSI)

PROTOCOL_INFOW

Contains information about a protocol. (Unicode)

SERVICE_ADDRESS

Contains address information for a service. The structure can accommodate many types of interprocess communications (IPC) mechanisms and their address forms, including remote procedure calls (RPC), named pipes, and sockets.

SERVICE_ADDRESSES

The SERVICE_ADDRESSES structure contains an array of SERVICE_ADDRESS data structures.

SERVICE_INFOA

Contains information about a network service or a network service type. (ANSI)

SERVICE_INFOW

Contains information about a network service or a network service type. (Unicode)

SERVICE_TYPE_INFO_ABSA

The SERVICE_TYPE_INFO_ABS structure contains information about a network service type. Use SERVICE_TYPE_INFO_ABS to add a network service type to a namespace. (ANSI)

SERVICE_TYPE_INFO_ABSW

The SERVICE_TYPE_INFO_ABS structure contains information about a network service type. Use SERVICE_TYPE_INFO_ABS to add a network service type to a namespace. (Unicode)

SERVICE_TYPE_VALUE_ABSA

Contains information about a network-service type value. This information may be specific to a namespace. (ANSI)

SERVICE_TYPE_VALUE_ABSW

Contains information about a network-service type value. This information may be specific to a namespace. (Unicode)

BLOB structure (nspapi.h)

10/22/2022 • 2 minutes to read • Edit Online

The **BLOB** structure, derived from Binary Large Object, contains information about a block of data.

Syntax

```
typedef struct _BLOB {
  ULONG cbSize;
#if ...
  BYTE  *pBlobData;
#else
  BYTE  *pBlobData;
#endif
} BLOB, *LPBLOB;
```

Members

`cbSize`

Size of the block of data pointed to by **pBlobData**, in bytes.

`pBlobData`

Pointer to a block of data.

Remarks

The structure name **BLOB** comes from the acronym BLOB, which stands for Binary Large Object.

This structure does not describe the nature of the data pointed to by **pBlobData**.

> **Note** Windows Sockets defines a similar **BLOB** structure in Wtypes.h. Using both header files in the same source code file creates redefinition–compile time errors.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	nspapi.h (include Wtypes.h, Nspapi.h, Winsock2.h, Wtypes.h, Nspapi.h, Winsock2.h)

See also

Bluetooth and BLOB

CSADDR_INFO structure (nspapi.h)

10/22/2022 • 3 minutes to read • Edit Online

The **CSADDR_INFO** structure contains Windows Sockets address information for a socket, network service, or namespace provider.

Syntax

```
typedef struct _CSADDR_INFO {
  SOCKET_ADDRESS LocalAddr;
  SOCKET_ADDRESS RemoteAddr;
  INT            iSocketType;
  INT            iProtocol;
} CSADDR_INFO, *PCSADDR_INFO, *LPCSADDR_INFO;
```

Members

`LocalAddr`

Type: **SOCKET_ADDRESS**

The Windows Sockets local address.

In a client application, pass this address to the **bind** function to obtain access to a network service.

In a network service, pass this address to the **bind** function so that the service is bound to the appropriate local address.

`RemoteAddr`

Type: **SOCKET_ADDRESS**

Windows Sockets remote address.

There are several uses for this remote address:

- You can use this remote address to connect to the service through the connect function. This is useful if an application performs send/receive operations that involve connection-oriented protocols.
- You can use this remote address with the sendto function when you are communicating over a connectionless (datagram) protocol. If you are using a connectionless protocol, such as UDP, **sendto** is typically the way you pass data to the remote system.

`iSocketType`

Type: **INT**

The type of Windows socket. Possible values for the socket type are defined in the *Winsock2.h* header file.

The following table lists the possible values supported for Windows Sockets 2:

VALUE	MEANING

SOCK_STREAM	A stream socket. This is a protocol that sends data as a stream of bytes, with no message boundaries. This socket type 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	A datagram socket. This socket type 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). Services use recvfrom function to obtain datagrams. The listen and accept functions do not work with datagrams.
SOCK_RDM	A reliable message datagram socket. This socket type preserves message boundaries in data. An example of this type is the Pragmatic General Multicast (PGM) multicast protocol implementation in Windows, often referred to as reliable multicast programming.
SOCK_SEQPACKET	A sequenced packet stream socket. This socket type provides a pseudo-stream packet based on datagrams.

`iProtocol`

Type: **INT**

The protocol used. The possible options for the *protocol* parameter are specific to the address family and socket type specified. Possible values are defined in the *Winsock2.h* and *Wsrm.h* header files.

On the Windows SDK released for Windows Vista and later, the organization of header files has changed and this parameter can be one of the values from 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.

The table below lists common values for the *protocol* although many other values are possible.

PROTOCOL	MEANING
IPPROTO_TCP 6	The Transmission Control Protocol (TCP). This is a possible value when the address family is **AF_INET** or **AF_INET6** and the **iSocketType** member is **SOCK_STREAM**.
IPPROTO_UDP 17	The User Datagram Protocol (UDP). This is a possible value when the address family is **AF_INET** or **AF_INET6** and the **iSocketType** member is **SOCK_DGRAM**.
IPPROTO_RM 113	The PGM protocol for reliable multicast. This is a possible value when the address family is **AF_INET** and the **iSocketType** member is **SOCK_RDM**. On the Windows SDK released for Windows Vista and later, this value is also called **IPPROTO_PGM**.

Remarks

The GetAddressByName function obtains Windows Sockets address information using **CSADDR_INFO** structures.

The getsockopt function called with the SO_BSP_STATE socket option retrieves a **CSADDR_INFO** structure for the specified socket.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	nspapi.h (include Nspapi.h)

See also

GetAddressByName

SOCKET_ADDRESS

SO_BSP_STATE

bind

connect

getsockopt

recv

send

sendto

EnumProtocolsA function (nspapi.h)

10/22/2022 • 4 minutes to read • Edit Online

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

> **Note** The **EnumProtocols** function is a Microsoft-specific extension to the Windows Sockets 1.1 specification. This function is obsolete. For the convenience of Windows Sockets 1.1 developers, the reference material is included. The WSAEnumProtocols function provides equivalent functionality in Windows Sockets 2.

Syntax

```
INT EnumProtocolsA(
  [in, optional] LPINT   lpiProtocols,
  [out]          LPVOID  lpProtocolBuffer,
  [in, out]      LPDWORD lpdwBufferLength
);
```

Parameters

`[in, optional] lpiProtocols`

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

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

The following protocol identifier values are defined.

VALUE	MEANING
IPPROTO_TCP	The Transmission Control Protocol (TCP), a connection-oriented stream protocol.
IPPROTO_UDP	The User Datagram Protocol (UDP), a connectionless datagram protocol.
ISOPROTO_TP4	The ISO connection-oriented transport protocol.
NSPROTO_IPX	The Internet Packet Exchange (IPX) protocol, a connectionless datagram protocol.
NSPROTO_SPX	The Sequenced Packet Exchange (SPX) protocol, a connection-oriented stream protocol.

NSPROTO_SPXII	The Sequenced Packet Exchange (SPX) protocol version 2, a connection-oriented stream protocol.

`[out] lpProtocolBuffer`

A pointer to a buffer that the function fills with an array of PROTOCOL_INFO data structures.

`[in, out] lpdwBufferLength`

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

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

Return value

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

If the function fails, the return value is SOCKET_ERROR(–1). To get extended error information, call GetLastError, which returns the following extended error code.

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

Remarks

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

```
#define WIN32_LEAN_AND_MEAN

#include <windows.h>
#include <winsock2.h>
#include <ws2tcpip.h>
#include <Nspapi.h>
#include <stdlib.h>
#include <stdio.h>


// Need to link with Ws2_32.lib and Mswsock.lib
#pragma comment (lib, "Ws2_32.lib")
#pragma comment (lib, "Mswsock.lib")

int FindProtocol(BOOL Reliable,
    BOOL MessageOriented, BOOL StreamOriented,
    BOOL Connectionless, DWORD *ProtocolUsed);

int __cdecl main(int argc, char **argv)
{
    WSADATA wsaData;

    int ProtocolError = SOCKET_ERROR;
    int iResult;

    BOOLEAN bReliable = FALSE;
    BOOLEAN bMessageOriented = FALSE;
```

```
    BOOLEAN bStreamOriented = TRUE;
    BOOLEAN bConnectionless = FALSE;
    DWORD *pProtocols = NULL;

    // Validate the parameters
    if (argc != 2) {
        printf("usage: %s servicename\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;
    }

    ProtocolError = FindProtocol( bReliable, bMessageOriented,
        bStreamOriented, bConnectionless, pProtocols);
    if (ProtocolError == SOCKET_ERROR) {
        printf("Unable to find a protocol to support the parameters requested\n");
        return 1;
    }

    // Connect to the servicename ...

    return 0;

}

#define MAX_PROTOCOLS 1024

int FindProtocol (
    BOOL Reliable,
    BOOL MessageOriented,
    BOOL StreamOriented,
    BOOL Connectionless,
    DWORD *ProtocolUsed
    )
{
    // local variables
    INT protocols[MAX_PROTOCOLS+1];
    BYTE buffer[2048];
    DWORD bytesRequired;
    INT err;
    PPROTOCOL_INFO protocolInfo;
    INT protocolCount;
    INT i;
    DWORD protocolIndex;
//    PCSADDR_INFO csaddrInfo;
//    INT addressCount;
//    SOCKET s;

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

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

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

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

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

                continue;
            }

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

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

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

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

    *ProtocolUsed = (INT) protocolIndex;
    return 0;
}
```

NOTE

The nspapi.h header defines EnumProtocols as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see Conventions for Function Prototypes.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	nspapi.h
Library	Mswsock.lib
DLL	Mswsock.dll

See also

GetAddressByName

PROTOCOL_INFO

Winsock Functions

Winsock Reference

EnumProtocolsW function (nspapi.h)

10/22/2022 • 4 minutes to read • Edit Online

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

> **Note** The **EnumProtocols** function is a Microsoft-specific extension to the Windows Sockets 1.1 specification. This function is obsolete. For the convenience of Windows Sockets 1.1 developers, the reference material is included. The WSAEnumProtocols function provides equivalent functionality in Windows Sockets 2.

Syntax

```
INT EnumProtocolsW(
  [in, optional] LPINT   lpiProtocols,
  [out]          LPVOID  lpProtocolBuffer,
  [in, out]      LPDWORD lpdwBufferLength
);
```

Parameters

`[in, optional] lpiProtocols`

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

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

The following protocol identifier values are defined.

VALUE	MEANING
IPPROTO_TCP	The Transmission Control Protocol (TCP), a connection-oriented stream protocol.
IPPROTO_UDP	The User Datagram Protocol (UDP), a connectionless datagram protocol.
ISOPROTO_TP4	The ISO connection-oriented transport protocol.
NSPROTO_IPX	The Internet Packet Exchange (IPX) protocol, a connectionless datagram protocol.
NSPROTO_SPX	The Sequenced Packet Exchange (SPX) protocol, a connection-oriented stream protocol.

NSPROTO_SPXII	The Sequenced Packet Exchange (SPX) protocol version 2, a connection-oriented stream protocol.

`[out] lpProtocolBuffer`

A pointer to a buffer that the function fills with an array of PROTOCOL_INFO data structures.

`[in, out] lpdwBufferLength`

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

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

Return value

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

If the function fails, the return value is SOCKET_ERROR(–1). To get extended error information, call GetLastError, which returns the following extended error code.

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

Remarks

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

```
#define WIN32_LEAN_AND_MEAN

#include <windows.h>
#include <winsock2.h>
#include <ws2tcpip.h>
#include <Nspapi.h>
#include <stdlib.h>
#include <stdio.h>


// Need to link with Ws2_32.lib and Mswsock.lib
#pragma comment (lib, "Ws2_32.lib")
#pragma comment (lib, "Mswsock.lib")

int FindProtocol(BOOL Reliable,
    BOOL MessageOriented, BOOL StreamOriented,
    BOOL Connectionless, DWORD *ProtocolUsed);

int __cdecl main(int argc, char **argv)
{
    WSADATA wsaData;

    int ProtocolError = SOCKET_ERROR;
    int iResult;

    BOOLEAN bReliable = FALSE;
    BOOLEAN bMessageOriented = FALSE;
```

```
    BOOLEAN bStreamOriented = TRUE;
    BOOLEAN bConnectionless = FALSE;
    DWORD *pProtocols = NULL;

    // Validate the parameters
    if (argc != 2) {
        printf("usage: %s servicename\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;
    }

    ProtocolError = FindProtocol( bReliable, bMessageOriented,
        bStreamOriented, bConnectionless, pProtocols);
    if (ProtocolError == SOCKET_ERROR) {
        printf("Unable to find a protocol to support the parameters requested\n");
        return 1;
    }

    // Connect to the servicename ...

    return 0;

}

#define MAX_PROTOCOLS 1024

int FindProtocol (
    BOOL Reliable,
    BOOL MessageOriented,
    BOOL StreamOriented,
    BOOL Connectionless,
    DWORD *ProtocolUsed
    )
{
    // local variables
    INT protocols[MAX_PROTOCOLS+1];
    BYTE buffer[2048];
    DWORD bytesRequired;
    INT err;
    PPROTOCOL_INFO protocolInfo;
    INT protocolCount;
    INT i;
    DWORD protocolIndex;
//    PCSADDR_INFO csaddrInfo;
//    INT addressCount;
//    SOCKET s;

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

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

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

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

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

                continue;
            }

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

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

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

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

    *ProtocolUsed = (INT) protocolIndex;
    return 0;
}
```

NOTE

The nspapi.h header defines EnumProtocols as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see Conventions for Function Prototypes.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	nspapi.h
Library	Mswsock.lib
DLL	Mswsock.dll

See also

GetAddressByName

PROTOCOL_INFO

Winsock Functions

Winsock Reference

GetAddressByNameA function (nspapi.h)

10/22/2022 • 7 minutes to read • Edit Online

[**GetAddressByName** is no longer available for use as of Windows Sockets 2. Instead, use the functions detailed in Protocol-Independent Name Resolution.]

The **GetAddressByName** function 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 bind function.

Syntax

```
INT GetAddressByNameA(
  [in]           DWORD                dwNameSpace,
  [in]           LPGUID               lpServiceType,
  [in, optional] LPSTR                lpServiceName,
  [in, optional] LPINT                lpiProtocols,
  [in]           DWORD                dwResolution,
  [in, optional] LPSERVICE_ASYNC_INFO lpServiceAsyncInfo,
  [out]          LPVOID               lpCsaddrBuffer,
  [in, out]      LPDWORD              lpdwBufferLength,
  [in, out]      LPSTR                lpAliasBuffer,
  [in, out]      LPDWORD              lpdwAliasBufferLength
);
```

Parameters

`[in] dwNameSpace`

The namespace, or set of default namespaces, that the operating system should query for network address information.

Use one of the following constants to specify a namespace.

VALUE	MEANING
NS_DEFAULT	A set of default namespaces. The function queries each namespace within this set. The set of default namespaces typically includes all the namespaces installed on the system. System administrators, however, can exclude particular namespaces from the set. This is the value that most applications should use for *dwNameSpace*.
NS_DNS	The Domain Name System (DNS) used in the Internet for host name resolution.
NS_NETBT	The NetBIOS over TCP/IP layer. All operating systems register their computer names with NetBIOS. This namespace is used to convert a computer name to an IP address that uses this registration. Note that NS_NETBT can access a WINS server to perform the resolution.

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

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

`[in] lpServiceType`

A pointer to a globally unique identifier (GUID) that specifies the type of the network service. The Svcguid.h header file includes definitions of several GUID service types, and macros for working with them.

The Svcguid.h header file is not automatically included by the Winsock2.h header file.

`[in, optional] lpServiceName`

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

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

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

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

`[in, optional] lpiProtocols`

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

If *lpiProtocols* is set to **NULL**, the function retrieves information on all available protocols.

`[in] dwResolution`

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

VALUE	MEANING
RES_SERVICE	If set, the function retrieves the address to which a service of the specified type should bind. This is the equivalent to setting the *lpServiceName* parameter to **NULL**. If this flag is clear, normal name resolution occurs.

RES_FIND_MULTIPLE	If this flag is set, the operating system performs an extensive search of all namespaces for the service. It asks every appropriate namespace to resolve the service name. If this flag is clear, the operating system stops looking for service addresses as soon as one is found.
RES_SOFT_SEARCH	This flag is valid if the namespace supports multiple levels of searching. If this flag is valid and set, the operating system performs a simple and quick search of the namespace. This is useful if an application only needs to obtain easy-to-find addresses for the service. If this flag is valid and clear, the operating system performs a more extensive search of the namespace.

`[in, optional] lpServiceAsyncInfo`

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

`[out] lpCsaddrBuffer`

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

`[in, out] lpdwBufferLength`

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

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

`[in, out] lpAliasBuffer`

A pointer to a buffer to receive alias information for the network service.

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

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

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

`[in, out] lpdwAliasBufferLength`

A pointer to a variable that, upon input, specifies the size, in elements (characters), of the buffer pointed to by *lpAliasBuffer*.

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

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

Return value

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

If the function fails, the return value is SOCKET_ERROR(–1). To get extended error information, call GetLastError, which returns the following extended error value.

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

Remarks

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

> **Note** The gethostbyname function has been deprecated by the introduction of the getaddrinfo function. Developers creating Windows Sockets 2 applications are urged to use the **getaddrinfo** function instead of **gethostbyname**.

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

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

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

> **NOTE**
> The nspapi.h header defines GetAddressByName as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see Conventions for Function Prototypes.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]

Target Platform	Windows
Header	nspapi.h
Library	Mswsock.lib
DLL	Mswsock.dll

See also

CSADDR_INFO

Winsock Functions

Winsock Reference

getaddrinfo

gethostbyname

GetAddressByNameW function (nspapi.h)

10/22/2022 • 7 minutes to read • Edit Online

[**GetAddressByName** is no longer available for use as of Windows Sockets 2. Instead, use the functions detailed in Protocol-Independent Name Resolution.]

The **GetAddressByName** function 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 bind function.

Syntax

```
INT GetAddressByNameW(
  [in]           DWORD                dwNameSpace,
  [in]           LPGUID               lpServiceType,
  [in, optional] LPWSTR               lpServiceName,
  [in, optional] LPINT                lpiProtocols,
  [in]           DWORD                dwResolution,
  [in, optional] LPSERVICE_ASYNC_INFO lpServiceAsyncInfo,
  [out]          LPVOID               lpCsaddrBuffer,
  [in, out]      LPDWORD              lpdwBufferLength,
  [in, out]      LPWSTR               lpAliasBuffer,
  [in, out]      LPDWORD              lpdwAliasBufferLength
);
```

Parameters

`[in] dwNameSpace`

The namespace, or set of default namespaces, that the operating system should query for network address information.

Use one of the following constants to specify a namespace.

VALUE	MEANING
NS_DEFAULT	A set of default namespaces. The function queries each namespace within this set. The set of default namespaces typically includes all the namespaces installed on the system. System administrators, however, can exclude particular namespaces from the set. This is the value that most applications should use for *dwNameSpace*.
NS_DNS	The Domain Name System (DNS) used in the Internet for host name resolution.
NS_NETBT	The NetBIOS over TCP/IP layer. All operating systems register their computer names with NetBIOS. This namespace is used to convert a computer name to an IP address that uses this registration. Note that NS_NETBT can access a WINS server to perform the resolution.

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

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

`[in] lpServiceType`

A pointer to a globally unique identifier (GUID) that specifies the type of the network service. The Svcguid.h header file includes definitions of several GUID service types, and macros for working with them.

The Svcguid.h header file is not automatically included by the Winsock2.h header file.

`[in, optional] lpServiceName`

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

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

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

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

`[in, optional] lpiProtocols`

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

If *lpiProtocols* is set to **NULL**, the function retrieves information on all available protocols.

`[in] dwResolution`

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

VALUE	MEANING
RES_SERVICE	If set, the function retrieves the address to which a service of the specified type should bind. This is the equivalent to setting the *lpServiceName* parameter to **NULL**. If this flag is clear, normal name resolution occurs.

RES_FIND_MULTIPLE	If this flag is set, the operating system performs an extensive search of all namespaces for the service. It asks every appropriate namespace to resolve the service name. If this flag is clear, the operating system stops looking for service addresses as soon as one is found.
RES_SOFT_SEARCH	This flag is valid if the namespace supports multiple levels of searching. If this flag is valid and set, the operating system performs a simple and quick search of the namespace. This is useful if an application only needs to obtain easy-to-find addresses for the service. If this flag is valid and clear, the operating system performs a more extensive search of the namespace.

`[in, optional] lpServiceAsyncInfo`

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

`[out] lpCsaddrBuffer`

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

`[in, out] lpdwBufferLength`

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

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

`[in, out] lpAliasBuffer`

A pointer to a buffer to receive alias information for the network service.

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

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

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

`[in, out] lpdwAliasBufferLength`

A pointer to a variable that, upon input, specifies the size, in elements (characters), of the buffer pointed to by *lpAliasBuffer*.

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

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

Return value

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

If the function fails, the return value is SOCKET_ERROR(–1). To get extended error information, call GetLastError, which returns the following extended error value.

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

Remarks

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

> **Note** The gethostbyname function has been deprecated by the introduction of the getaddrinfo function. Developers creating Windows Sockets 2 applications are urged to use the **getaddrinfo** function instead of **gethostbyname**.

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

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

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

> **NOTE**
> The nspapi.h header defines GetAddressByName as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see Conventions for Function Prototypes.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]

Target Platform	Windows
Header	nspapi.h
Library	Mswsock.lib
DLL	Mswsock.dll

See also

CSADDR_INFO

Winsock Functions

Winsock Reference

getaddrinfo

gethostbyname

GetNameByTypeA function (nspapi.h)

10/22/2022 • 2 minutes to read • Edit Online

The **GetNameByType** function retrieves the name of a network service for the specified service type.

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

> **Note** The functions detailed in Protocol-Independent Name Resolution provide equivalent functionality in Windows Sockets 2.

Syntax

```
INT GetNameByTypeA(
  [in]  LPGUID lpServiceType,
  [out] LPSTR  lpServiceName,
  [in]  DWORD  dwNameLength
);
```

Parameters

`[in] lpServiceType`

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

The *Svcguid.h* header file is not automatically included by the *Winsock2.h* header file.

`[out] lpServiceName`

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

`[in] dwNameLength`

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

Return value

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

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

Remarks

> **NOTE**
> The nspapi.h header defines GetNameByType as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see Conventions for Function Prototypes.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	nspapi.h
Library	Mswsock.lib
DLL	Mswsock.dll

See also

GetTypeByName

Winsock Functions

Winsock Reference

GetNameByTypeW function (nspapi.h)

10/22/2022 • 2 minutes to read • Edit Online

The **GetNameByType** function retrieves the name of a network service for the specified service type.

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

> **Note** The functions detailed in Protocol-Independent Name Resolution provide equivalent functionality in Windows Sockets 2.

Syntax

```
INT GetNameByTypeW(
  [in]  LPGUID lpServiceType,
  [out] LPWSTR lpServiceName,
  [in]  DWORD  dwNameLength
);
```

Parameters

`[in] lpServiceType`

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

The *Svcguid.h* header file is not automatically included by the *Winsock2.h* header file.

`[out] lpServiceName`

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

`[in] dwNameLength`

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

Return value

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

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

Remarks

> **NOTE**
> The nspapi.h header defines GetNameByType as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see Conventions for Function Prototypes.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	nspapi.h
Library	Mswsock.lib
DLL	Mswsock.dll

See also

GetTypeByName

Winsock Functions

Winsock Reference

GetServiceA function (nspapi.h)

10/22/2022 • 5 minutes to read • Edit Online

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

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

> **Note** The functions detailed in Protocol-Independent Name Resolution provide equivalent functionality in Windows Sockets 2.

Syntax

```
INT GetServiceA(
  [in]           DWORD                dwNameSpace,
  [in]           LPGUID               lpGuid,
  [in]           LPSTR                lpServiceName,
  [in]           DWORD                dwProperties,
  [out]          LPVOID               lpBuffer,
  [in, out]      LPDWORD              lpdwBufferSize,
  [in, optional] LPSERVICE_ASYNC_INFO lpServiceAsyncInfo
);
```

Parameters

`[in] dwNameSpace`

The namespace, or a set of default namespaces, that the operating system should query for information about the specified network service.

Use one of the following constants to specify a namespace.

VALUE	MEANING
NS_DEFAULT	A set of default namespaces. The operating system queries each namespace within this set. The set of default namespaces typically includes all the namespaces installed on the system. System administrators, however, can exclude particular namespaces from the set. NS_DEFAULT is the value that most applications should use for *dwNameSpace*.
NS_DNS	The Domain Name System used in the Internet for host name resolution.

NS_NETBT	The NetBIOS over TCP/IP layer. All operating systems register their computer names with NetBIOS. This namespace is used to resolve a computer name into an IP address using this registration. Note that NS_NETBT can access a WINS server to perform the resolution.
NS_SAP	The NetWare Service Advertising Protocol. This can access the NetWare bindery if appropriate. NS_SAP is a dynamic namespace that allows registration of services.
NS_TCPIP_HOSTS	Looks up host names and IP addresses in the <systemroot>\system32\drivers\etc\hosts file.
NS_TCPIP_LOCAL	Local TCP/IP name resolution mechanisms, including comparisons against the local host name and looks up host names and IP addresses in cache of host to IP address mappings.

Most calls to **GetService** should use the special value NS_DEFAULT. This lets a client get by without knowing available namespaces on an internetwork. The system administrator determines namespace access. Namespaces can come and go without the client having to be aware of the changes.

`[in] lpGuid`

A pointer to a globally unique identifier (GUID) that specifies the type of the network service. The *Svcguid.h* header file includes GUID service types from many well-known services within the DNS and SAP namespaces.

The *Svcguid.h* header file is not automatically included by the *Winsock2.h* header file.

`[in] lpServiceName`

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

`[in] dwProperties`

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

VALUE	MEANING
PROP_COMMENT	If this flag is set, the function stores data in the **lpComment** member of the data structures stored in **lpBuffer*.
PROP_LOCALE	If this flag is set, the function stores data in the **lpLocale** member of the data structures stored in **lpBuffer*.
PROP_DISPLAY_HINT	If this flag is set, the function stores data in the **dwDisplayHint** member of the data structures stored in **lpBuffer*.
PROP_VERSION	If this flag is set, the function stores data in the **dwVersion** member of the data structures stored in **lpBuffer*.

PROP_START_TIME	If this flag is set, the function stores data in the **dwTime** member of the data structures stored in **lpBuffer*.
PROP_MACHINE	If this flag is set, the function stores data in the **lpMachineName** member of the data structures stored in **lpBuffer*.
PROP_ADDRESSES	If this flag is set, the function stores data in the **lpServiceAddress** member of the data structures stored in **lpBuffer*.
PROP_SD	If this flag is set, the function stores data in the **ServiceSpecificInfo** member of the data structures stored in **lpBuffer*.
PROP_ALL	If this flag is set, the function stores data in all of the members of the data structures stored in **lpBuffer*.

`[out] lpBuffer`

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

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

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

`[in, out] lpdwBufferSize`

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

`[in, optional] lpServiceAsyncInfo`

Reserved for future use. Must be set to **NULL**.

Return value

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

If the function fails, the return value is SOCKET_ERROR (– 1). To get extended error information, call GetLastError, which returns one of the following extended error values.

ERROR CODE	MEANING
ERROR_INSUFFICIENT_BUFFER	The buffer pointed to by *lpBuffer* is too small to receive all of the requested information. Call the function with a buffer at least as large as the value returned in **lpdwBufferSize*.

ERROR_SERVICE_NOT_FOUND	The specified service was not found, or the specified namespace is not in use. The function return value is zero in this case.

Remarks

> **NOTE**
>
> The nspapi.h header defines GetService as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see Conventions for Function Prototypes.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	nspapi.h
Library	Mswsock.lib
DLL	Mswsock.dll

See also

NS_SERVICE_INFO

SERVICE_INFO

SetService

Winsock Functions

Winsock Reference

GetServiceW function (nspapi.h)

10/22/2022 • 5 minutes to read • Edit Online

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

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

> **Note** The functions detailed in Protocol-Independent Name Resolution provide equivalent functionality in Windows Sockets 2.

Syntax

```
INT GetServiceW(
  [in]           DWORD                dwNameSpace,
  [in]           LPGUID               lpGuid,
  [in]           LPWSTR               lpServiceName,
  [in]           DWORD                dwProperties,
  [out]          LPVOID               lpBuffer,
  [in, out]      LPDWORD              lpdwBufferSize,
  [in, optional] LPSERVICE_ASYNC_INFO lpServiceAsyncInfo
);
```

Parameters

`[in] dwNameSpace`

The namespace, or a set of default namespaces, that the operating system should query for information about the specified network service.

Use one of the following constants to specify a namespace.

VALUE	MEANING
NS_DEFAULT	A set of default namespaces. The operating system queries each namespace within this set. The set of default namespaces typically includes all the namespaces installed on the system. System administrators, however, can exclude particular namespaces from the set. NS_DEFAULT is the value that most applications should use for *dwNameSpace*.
NS_DNS	The Domain Name System used in the Internet for host name resolution.

NS_NETBT	The NetBIOS over TCP/IP layer. All operating systems register their computer names with NetBIOS. This namespace is used to resolve a computer name into an IP address using this registration. Note that NS_NETBT can access a WINS server to perform the resolution.
NS_SAP	The NetWare Service Advertising Protocol. This can access the NetWare bindery if appropriate. NS_SAP is a dynamic namespace that allows registration of services.
NS_TCPIP_HOSTS	Looks up host names and IP addresses in the <systemroot>\system32\drivers\etc\hosts file.
NS_TCPIP_LOCAL	Local TCP/IP name resolution mechanisms, including comparisons against the local host name and looks up host names and IP addresses in cache of host to IP address mappings.

Most calls to **GetService** should use the special value NS_DEFAULT. This lets a client get by without knowing available namespaces on an internetwork. The system administrator determines namespace access. Namespaces can come and go without the client having to be aware of the changes.

`[in] lpGuid`

A pointer to a globally unique identifier (GUID) that specifies the type of the network service. The *Svcguid.h* header file includes GUID service types from many well-known services within the DNS and SAP namespaces.

The *Svcguid.h* header file is not automatically included by the *Winsock2.h* header file.

`[in] lpServiceName`

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

`[in] dwProperties`

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

VALUE	MEANING
PROP_COMMENT	If this flag is set, the function stores data in the **lpComment** member of the data structures stored in **lpBuffer*.
PROP_LOCALE	If this flag is set, the function stores data in the **lpLocale** member of the data structures stored in **lpBuffer*.
PROP_DISPLAY_HINT	If this flag is set, the function stores data in the **dwDisplayHint** member of the data structures stored in **lpBuffer*.
PROP_VERSION	If this flag is set, the function stores data in the **dwVersion** member of the data structures stored in **lpBuffer*.

PROP_START_TIME	If this flag is set, the function stores data in the **dwTime** member of the data structures stored in **lpBuffer*.
PROP_MACHINE	If this flag is set, the function stores data in the **lpMachineName** member of the data structures stored in **lpBuffer*.
PROP_ADDRESSES	If this flag is set, the function stores data in the **lpServiceAddress** member of the data structures stored in **lpBuffer*.
PROP_SD	If this flag is set, the function stores data in the **ServiceSpecificInfo** member of the data structures stored in **lpBuffer*.
PROP_ALL	If this flag is set, the function stores data in all of the members of the data structures stored in **lpBuffer*.

`[out] lpBuffer`

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

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

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

`[in, out] lpdwBufferSize`

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

`[in, optional] lpServiceAsyncInfo`

Reserved for future use. Must be set to **NULL**.

Return value

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

If the function fails, the return value is SOCKET_ERROR (– 1). To get extended error information, call GetLastError, which returns one of the following extended error values.

ERROR CODE	MEANING
ERROR_INSUFFICIENT_BUFFER	The buffer pointed to by *lpBuffer* is too small to receive all of the requested information. Call the function with a buffer at least as large as the value returned in **lpdwBufferSize*.

ERROR_SERVICE_NOT_FOUND	The specified service was not found, or the specified namespace is not in use. The function return value is zero in this case.

Remarks

> **NOTE**
>
> The nspapi.h header defines GetService as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see Conventions for Function Prototypes.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	nspapi.h
Library	Mswsock.lib
DLL	Mswsock.dll

See also

NS_SERVICE_INFO

SERVICE_INFO

SetService

Winsock Functions

Winsock Reference

GetTypeByNameA function (nspapi.h)

10/22/2022 • 2 minutes to read • Edit Online

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

> **Note** The **GetTypeByName** function is a Microsoft-specific extension to the Windows Sockets 1.1 specification. This function is obsolete. For the convenience of Windows Sockets 1.1 developers, this reference material is included. The functions detailed in Protocol-Independent Name Resolution provide equivalent functionality in Windows Sockets 2.

Syntax

```
INT GetTypeByNameA(
  [in]      LPSTR  lpServiceName,
  [in, out] LPGUID lpServiceType
);
```

Parameters

`[in] lpServiceName`

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

`[in, out] lpServiceType`

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

The *Svcguid.h* header file is not automatically included by the *Winsock2.h* header file.

Return value

If the function succeeds, the return value is zero.

If the function fails, the return value is SOCKET_ERROR(– 1). To get extended error information, call GetLastError, which returns the following extended error value.

VALUE	MEANING
ERROR_SERVICE_DOES_NOT_EXIST	The specified service type is unknown.

Remarks

> **NOTE**
> The nspapi.h header defines GetTypeByName as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see Conventions for Function Prototypes.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	nspapi.h
Library	Mswsock.lib
DLL	Mswsock.dll

See also

GetNameByType

Winsock Functions

Winsock Reference

GetTypeByNameW function (nspapi.h)

10/22/2022 • 2 minutes to read • Edit Online

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

> **Note** The **GetTypeByName** function is a Microsoft-specific extension to the Windows Sockets 1.1 specification. This function is obsolete. For the convenience of Windows Sockets 1.1 developers, this reference material is included. The functions detailed in Protocol-Independent Name Resolution provide equivalent functionality in Windows Sockets 2.

Syntax

```
INT GetTypeByNameW(
  [in]      LPWSTR lpServiceName,
  [in, out] LPGUID lpServiceType
);
```

Parameters

`[in] lpServiceName`

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

`[in, out] lpServiceType`

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

The *Svcguid.h* header file is not automatically included by the *Winsock2.h* header file.

Return value

If the function succeeds, the return value is zero.

If the function fails, the return value is SOCKET_ERROR(– 1). To get extended error information, call GetLastError, which returns the following extended error value.

VALUE	MEANING
ERROR_SERVICE_DOES_NOT_EXIST	The specified service type is unknown.

Remarks

> **NOTE**
> The nspapi.h header defines GetTypeByName as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see Conventions for Function Prototypes.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	nspapi.h
Library	Mswsock.lib
DLL	Mswsock.dll

See also

GetNameByType

Winsock Functions

Winsock Reference

NS_SERVICE_INFOA structure (nspapi.h)

10/22/2022 • 2 minutes to read • Edit Online

The **NS_SERVICE_INFO** structure contains information about a network service or a network service type in the context of a specified namespace, or a set of default namespaces.

Syntax

```
typedef struct _NS_SERVICE_INFOA {
  DWORD         dwNameSpace;
  SERVICE_INFOA ServiceInfo;
} NS_SERVICE_INFOA, *PNS_SERVICE_INFOA, *LPNS_SERVICE_INFOA;
```

Members

`dwNameSpace`

Type: **DWORD**

Namespace, or a set of default namespaces, to which this service information applies.

Use one of the following constant values to specify a namespace.

VALUE	MEANING
NS_DEFAULT	A set of default namespaces. The set of default namespaces typically includes all the namespaces installed on the system. System administrators, however, can exclude particular namespaces from the set.
NS_DNS	The Domain Name System used in the Internet to resolve the name of the host.
NS_MS	The Microsoft namespace.
NS_NDS	The NetWare 4 provider.
NS_NETBT	The NetBIOS over TCP/IP layer. The operating system registers their computer names with NetBIOS. This namespace is used to convert a computer name to an IP address that uses this registration.
NS_NIS	
NS_SAP	The NetWare Service Advertising Protocol. This can access the NetWare bindery, if appropriate. NS_SAP is a dynamic namespace that enables the registration of services.

NS_STDA	
NS_TCPIP_HOSTS	Lookup value in the <systemroot>\system32\drivers\etc\posts file.
NS_TCPIP_LOCAL	Local TCP/IP name resolution mechanisms, including comparisons against the local host name and lookup value in the cache of host to IP address mappings.
NS_WINS	The Windows Internet Name System (WINS) namespace.
NS_X500	The X.500 directory service namespace.

`ServiceInfo`

Type: **SERVICE_INFO**

A SERVICE_INFO structure that contains information about a network service or network service type.

Remarks

> **NOTE**
> The nspapi.h header defines NS_SERVICE_INFO as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see Conventions for Function Prototypes.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	nspapi.h

See also

SERVICE_INFO

NS_SERVICE_INFOW structure (nspapi.h)

10/22/2022 • 2 minutes to read • Edit Online

The **NS_SERVICE_INFO** structure contains information about a network service or a network service type in the context of a specified namespace, or a set of default namespaces.

Syntax

```
typedef struct _NS_SERVICE_INFOW {
  DWORD         dwNameSpace;
  SERVICE_INFOW ServiceInfo;
} NS_SERVICE_INFOW, *PNS_SERVICE_INFOW, *LPNS_SERVICE_INFOW;
```

Members

`dwNameSpace`

Type: **DWORD**

Namespace, or a set of default namespaces, to which this service information applies.

Use one of the following constant values to specify a namespace.

VALUE	MEANING
NS_DEFAULT	A set of default namespaces. The set of default namespaces typically includes all the namespaces installed on the system. System administrators, however, can exclude particular namespaces from the set.
NS_DNS	The Domain Name System used in the Internet to resolve the name of the host.
NS_MS	The Microsoft namespace.
NS_NDS	The NetWare 4 provider.
NS_NETBT	The NetBIOS over TCP/IP layer. The operating system registers their computer names with NetBIOS. This namespace is used to convert a computer name to an IP address that uses this registration.
NS_NIS	
NS_SAP	The NetWare Service Advertising Protocol. This can access the NetWare bindery, if appropriate. NS_SAP is a dynamic namespace that enables the registration of services.

NS_STDA	
NS_TCPIP_HOSTS	Lookup value in the <systemroot>\system32\drivers\etc\posts file.
NS_TCPIP_LOCAL	Local TCP/IP name resolution mechanisms, including comparisons against the local host name and lookup value in the cache of host to IP address mappings.
NS_WINS	The Windows Internet Name System (WINS) namespace.
NS_X500	The X.500 directory service namespace.

`ServiceInfo`

Type: **SERVICE_INFO**

A SERVICE_INFO structure that contains information about a network service or network service type.

Remarks

> **NOTE**
> The nspapi.h header defines NS_SERVICE_INFO as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see Conventions for Function Prototypes.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	nspapi.h

See also

SERVICE_INFO

PROTOCOL_INFOA structure (nspapi.h)

10/22/2022 • 3 minutes to read • Edit Online

The **PROTOCOL_INFO** structure contains information about a protocol.

Syntax

```
typedef struct _PROTOCOL_INFOA {
  DWORD dwServiceFlags;
  INT   iAddressFamily;
  INT   iMaxSockAddr;
  INT   iMinSockAddr;
  INT   iSocketType;
  INT   iProtocol;
  DWORD dwMessageSize;
  LPSTR lpProtocol;
} PROTOCOL_INFOA, *PPROTOCOL_INFOA, *LPPROTOCOL_INFOA;
```

Members

`dwServiceFlags`

Type: **DWORD**

A set of bit flags that specifies the services provided by the protocol. One or more of the following bit flags may be set.

VALUE	MEANING
XP_CONNECTIONLESS	If this flag is set, the protocol provides connectionless (datagram) service. If this flag is clear, the protocol provides connection-oriented data transfer.
XP_GUARANTEED_DELIVERY	If this flag is set, the protocol guarantees that all data sent will reach the intended destination. If this flag is clear, there is no such guarantee.
XP_GUARANTEED_ORDER	If this flag is set, the protocol guarantees that data will arrive in the order in which it was sent. Note that this characteristic does not guarantee delivery of the data, only its order. If this flag is clear, the order of data sent is not guaranteed.
XP_MESSAGE_ORIENTED	If this flag is set, the protocol is message-oriented. A message-oriented protocol honors message boundaries. If this flag is clear, the protocol is stream oriented, and the concept of message boundaries is irrelevant.

XP_PSEUDO_STREAM	If this flag is set, the protocol is a message-oriented protocol that ignores message boundaries for all receive operations. This optional capability is useful when you do not want the protocol to frame messages. An application that requires stream-oriented characteristics can open a socket with type SOCK_STREAM for transport protocols that support this functionality, regardless of the value of **iSocketType**.
XP_GRACEFUL_CLOSE	If this flag is set, the protocol supports two-phase close operations, also known as graceful close operations. If this flag is clear, the protocol supports only abortive close operations.
XP_EXPEDITED_DATA	If this flag is set, the protocol supports expedited data, also known as urgent data.
XP_CONNECT_DATA	If this flag is set, the protocol supports connect data.
XP_DISCONNECT_DATA	If this flag is set, the protocol supports disconnect data.
XP_SUPPORTS_BROADCAST	If this flag is set, the protocol supports a broadcast mechanism.
XP_SUPPORTS_MULTICAST	If this flag is set, the protocol supports a multicast mechanism.
XP_BANDWIDTH_ALLOCATION	If this flag is set, the protocol supports a mechanism for allocating a guaranteed bandwidth to an application.
XP_FRAGMENTATION	If this flag is set, the protocol supports message fragmentation; physical network MTU is hidden from applications.
XP_ENCRYPTS	If this flag is set, the protocol supports data encryption.

`iAddressFamily`

Type: **INT**

Value to pass as the *af* parameter when the socket function is called to open a socket for the protocol. This address family value uniquely defines the structure of protocol addresses, also known as **sockaddr** structures, used by the protocol.

`iMaxSockAddr`

Type: **INT**

Maximum length of a socket address supported by the protocol, in bytes.

`iMinSockAddr`

Type: **INT**

Minimum length of a socket address supported by the protocol, in bytes.

`iSocketType`

Type: **INT**

Value to pass as the *type* parameter when the **socket** function is called to open a socket for the protocol.

Note that if XP_PSEUDO_STREAM is set in **dwServiceFlags**, the application can specify SOCK_STREAM as the *type* parameter to **socket**, regardless of the value of **iSocketType**.

`iProtocol`

Type: **INT**

Value to pass as the *protocol* parameter when the **socket** function is called to open a socket for the protocol.

`dwMessageSize`

Type: **DWORD**

Maximum message size supported by the protocol, in bytes. This is the maximum size of a message that can be sent from or received by the host. For protocols that do not support message framing, the actual maximum size of a message that can be sent to a given address may be less than this value.

The following special message size values are defined.

VALUE	MEANING
0	The protocol is stream-oriented; the concept of message size is not relevant.
0xFFFFFFFF	The protocol is message-oriented, but there is no maximum message size.

`lpProtocol`

Type: **LPTSTR**

Pointer to a zero-terminated string that supplies a name for the protocol; for example, "SPX2."

Remarks

> **NOTE**
> The nspapi.h header defines PROTOCOL_INFO as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see Conventions for Function Prototypes.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]

Minimum supported server	Windows 2000 Server [desktop apps only]
Header	nspapi.h

See also

EnumProtocols

socket

PROTOCOL_INFOW structure (nspapi.h)

10/22/2022 • 3 minutes to read • Edit Online

The **PROTOCOL_INFO** structure contains information about a protocol.

Syntax

```
typedef struct _PROTOCOL_INFOW {
  DWORD  dwServiceFlags;
  INT    iAddressFamily;
  INT    iMaxSockAddr;
  INT    iMinSockAddr;
  INT    iSocketType;
  INT    iProtocol;
  DWORD  dwMessageSize;
  LPWSTR lpProtocol;
} PROTOCOL_INFOW, *PPROTOCOL_INFOW, *LPPROTOCOL_INFOW;
```

Members

`dwServiceFlags`

Type: **DWORD**

A set of bit flags that specifies the services provided by the protocol. One or more of the following bit flags may be set.

VALUE	MEANING
XP_CONNECTIONLESS	If this flag is set, the protocol provides connectionless (datagram) service. If this flag is clear, the protocol provides connection-oriented data transfer.
XP_GUARANTEED_DELIVERY	If this flag is set, the protocol guarantees that all data sent will reach the intended destination. If this flag is clear, there is no such guarantee.
XP_GUARANTEED_ORDER	If this flag is set, the protocol guarantees that data will arrive in the order in which it was sent. Note that this characteristic does not guarantee delivery of the data, only its order. If this flag is clear, the order of data sent is not guaranteed.
XP_MESSAGE_ORIENTED	If this flag is set, the protocol is message-oriented. A message-oriented protocol honors message boundaries. If this flag is clear, the protocol is stream oriented, and the concept of message boundaries is irrelevant.

XP_PSEUDO_STREAM	If this flag is set, the protocol is a message-oriented protocol that ignores message boundaries for all receive operations. This optional capability is useful when you do not want the protocol to frame messages. An application that requires stream-oriented characteristics can open a socket with type SOCK_STREAM for transport protocols that support this functionality, regardless of the value of **iSocketType**.
XP_GRACEFUL_CLOSE	If this flag is set, the protocol supports two-phase close operations, also known as graceful close operations. If this flag is clear, the protocol supports only abortive close operations.
XP_EXPEDITED_DATA	If this flag is set, the protocol supports expedited data, also known as urgent data.
XP_CONNECT_DATA	If this flag is set, the protocol supports connect data.
XP_DISCONNECT_DATA	If this flag is set, the protocol supports disconnect data.
XP_SUPPORTS_BROADCAST	If this flag is set, the protocol supports a broadcast mechanism.
XP_SUPPORTS_MULTICAST	If this flag is set, the protocol supports a multicast mechanism.
XP_BANDWIDTH_ALLOCATION	If this flag is set, the protocol supports a mechanism for allocating a guaranteed bandwidth to an application.
XP_FRAGMENTATION	If this flag is set, the protocol supports message fragmentation; physical network MTU is hidden from applications.
XP_ENCRYPTS	If this flag is set, the protocol supports data encryption.

`iAddressFamily`

Type: **INT**

Value to pass as the *af* parameter when the socket function is called to open a socket for the protocol. This address family value uniquely defines the structure of protocol addresses, also known as **sockaddr** structures, used by the protocol.

`iMaxSockAddr`

Type: **INT**

Maximum length of a socket address supported by the protocol, in bytes.

`iMinSockAddr`

Type: **INT**

Minimum length of a socket address supported by the protocol, in bytes.

`iSocketType`

Type: **INT**

Value to pass as the *type* parameter when the **socket** function is called to open a socket for the protocol.

Note that if XP_PSEUDO_STREAM is set in **dwServiceFlags**, the application can specify SOCK_STREAM as the *type* parameter to **socket**, regardless of the value of **iSocketType**.

`iProtocol`

Type: **INT**

Value to pass as the *protocol* parameter when the **socket** function is called to open a socket for the protocol.

`dwMessageSize`

Type: **DWORD**

Maximum message size supported by the protocol, in bytes. This is the maximum size of a message that can be sent from or received by the host. For protocols that do not support message framing, the actual maximum size of a message that can be sent to a given address may be less than this value.

The following special message size values are defined.

VALUE	MEANING
0	The protocol is stream-oriented; the concept of message size is not relevant.
0xFFFFFFFF	The protocol is message-oriented, but there is no maximum message size.

`lpProtocol`

Type: **LPTSTR**

Pointer to a zero-terminated string that supplies a name for the protocol; for example, "SPX2."

Remarks

> **NOTE**
> The nspapi.h header defines PROTOCOL_INFO as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see Conventions for Function Prototypes.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]

Minimum supported server	Windows 2000 Server [desktop apps only]
Header	nspapi.h

See also

EnumProtocols

socket

SERVICE_ADDRESS structure (nspapi.h)

10/22/2022 • 2 minutes to read • Edit Online

The **SERVICE_ADDRESS** structure contains address information for a service. The structure can accommodate many types of interprocess communications (IPC) mechanisms and their address forms, including remote procedure calls (RPC), named pipes, and sockets.

Syntax

```
typedef struct _SERVICE_ADDRESS {
  DWORD dwAddressType;
  DWORD dwAddressFlags;
  DWORD dwAddressLength;
  DWORD dwPrincipalLength;
#if ...
  BYTE  *lpAddress;
#else
  BYTE  *lpAddress;
#endif
#if ...
  BYTE  *lpPrincipal;
#else
  BYTE  *lpPrincipal;
#endif
} SERVICE_ADDRESS, *PSERVICE_ADDRESS, *LPSERVICE_ADDRESS;
```

Members

`dwAddressType`

Type: **DWORD**

The address family to which the socket address pointed to by **lpAddress** member belongs.

`dwAddressFlags`

Type: **DWORD**

A set of bit flags that specify properties of the address. The following bit flags are defined.

VALUE	MEANING
SERVICE_ADDRESS_FLAG_RPC_CN	If this bit flag is set, the service supports connection-oriented RPC over this transport protocol.
SERVICE_ADDRESS_FLAG_RPC_DG	If this bit flag is set, the service supports datagram-oriented RPC over this transport protocol.
SERVICE_ADDRESS_FLAG_RPC_NB	If this bit flag is set, the service supports NetBIOS RPC over this transport protocol.

`dwAddressLength`

Type: **DWORD**

The size, in bytes, of the address.

`dwPrincipalLength`

Type: **DWORD**

Reserved for future use. Must be zero.

`lpAddress`

Type: **BYTE***

A pointer to a socket address of the appropriate type.

`lpPrincipal`

Type: **BYTE***

Reserved for future use. Must be **NULL**.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	nspapi.h

See also

SERVICE_ADDRESSES

SERVICE_INFO

SERVICE_ADDRESSES structure (nspapi.h)

10/22/2022 • 2 minutes to read • Edit Online

The **SERVICE_ADDRESSES** structure contains an array of SERVICE_ADDRESS data structures.

Syntax

```
typedef struct _SERVICE_ADDRESSES {
  DWORD           dwAddressCount;
  SERVICE_ADDRESS *Addressses[];
  SERVICE_ADDRESS Addresses[1];
} SERVICE_ADDRESSES, *PSERVICE_ADDRESSES, *LPSERVICE_ADDRESSES;
```

Members

`dwAddressCount`

Number of SERVICE_ADDRESS structures in the **Addresses** array.

`Addressses[*]`

`Addresses[1]`

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	nspapi.h

See also

SERVICE_ADDRESS

SERVICE_INFO

SERVICE_INFOA structure (nspapi.h)

10/22/2022 • 3 minutes to read • Edit Online

The **SERVICE_INFO** structure contains information about a network service or a network service type.

Syntax

```
typedef struct _SERVICE_INFOA {
  LPGUID              lpServiceType;
  LPSTR               lpServiceName;
  LPSTR               lpComment;
  LPSTR               lpLocale;
  DWORD               dwDisplayHint;
  DWORD               dwVersion;
  DWORD               dwTime;
  LPSTR               lpMachineName;
  LPSERVICE_ADDRESSES lpServiceAddress;
  BLOB                ServiceSpecificInfo;
} SERVICE_INFOA, *PSERVICE_INFOA, *LPSERVICE_INFOA;
```

Members

`lpServiceType`

Type: **LPGUID**

A pointer to a GUID that is the type of the network service.

`lpServiceName`

Type: **LPTSTR**

A pointer to a **NULL**-terminated string that is the name of the network service.

If you are calling the **SetService** function with the *dwNameSpace* parameter set to NS_DEFAULT, the network service name must be a common name. A common name is what the network service is commonly known as. An example of a common name for a network service is "My SQL Server".

If you are calling the **SetService** function with the *dwNameSpace* parameter set to a specific service name, the network service name can be a common name or a distinguished name. A distinguished name distinguishes the service to a unique location with a directory service. An example of a distinguished name for a network service is "MS\SYS\NT\DEV\My SQL Server".

`lpComment`

Type: **LPTSTR**

A pointer to a **NULL**-terminated string that is a comment or description for the network service. For example, "Used for development upgrades."

`lpLocale`

Type: **LPTSTR**

A pointer to a **NULL**-terminated string that contains locale information.

`dwDisplayHint`

Type: **DWORD**

A hint as to how to display the network service in a network browsing user interface. This can be one of the following values.

VALUE	MEANING
RESOURCEDISPLAYTYPE_DOMAIN	Display the network service as a domain.
RESOURCEDISPLAYTYPE_FILE	Display the network service as a file.
RESOURCEDISPLAYTYPE_GENERIC	The method used to display the object does not matter.
RESOURCEDISPLAYTYPE_GROUP	Display the network service as a group.
RESOURCEDISPLAYTYPE_SERVER	Display the network service as a server.
RESOURCEDISPLAYTYPE_SHARE	Display the network service as a sharepoint.
RESOURCEDISPLAYTYPE_TREE	Display the network service as a tree.

`dwVersion`

Type: **DWORD**

The version for the network service. The high word of this value specifies a major version number. The low word of this value specifies a minor version number.

`dwTime`

Type: **DWORD**

Reserved for future use. Must be zero.

`lpMachineName`

Type: **LPTSTR**

A pointer to a **NULL**-terminated string that is the name of the computer on which the network service is running.

`lpServiceAddress`

Type: **LPSERVICE_ADDRESSES**

A pointer to a SERVICE_ADDRESSES structure that contains an array of SERVICE_ADDRESS structures. Each **SERVICE_ADDRESS** structure contains information about a network service address.

A network service can call the getsockname function to determine the local address of the system.

`ServiceSpecificInfo`

Type: **BLOB**

A BLOB structure that specifies service-defined information.

> **Note** In general, the data pointed to by the BLOB structure's **pBlobData** member must not contain any pointers. That is because only the network service knows the format of the data; copying the data without such knowledge would lead to pointer invalidation. If the data pointed to by **pBlobData** contains variable-sized elements, offsets from **pBlobData** can be used to indicate the location of those elements. There is one exception to this general rule: when **pBlobData** points to a SERVICE_TYPE_INFO_ABS structure. This is possible because both the **SERVICE_TYPE_INFO_ABS** structure, and any SERVICE_TYPE_VALUE_ABS structures it contains are predefined, and thus their formats are known to the operating system.

Remarks

> **NOTE**
>
> The nspapi.h header defines SERVICE_INFO as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see Conventions for Function Prototypes.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	nspapi.h

See also

BLOB

GetService

NS_SERVICE_INFO

SERVICE_ADDRESS

SERVICE_ADDRESSES

SERVICE_TYPE_INFO_ABS

SERVICE_TYPE_VALUE_ABS

SetService

SERVICE_INFOW structure (nspapi.h)

10/22/2022 • 3 minutes to read • Edit Online

The **SERVICE_INFO** structure contains information about a network service or a network service type.

Syntax

```
typedef struct _SERVICE_INFOW {
  LPGUID              lpServiceType;
  LPWSTR              lpServiceName;
  LPWSTR              lpComment;
  LPWSTR              lpLocale;
  DWORD               dwDisplayHint;
  DWORD               dwVersion;
  DWORD               dwTime;
  LPWSTR              lpMachineName;
  LPSERVICE_ADDRESSES lpServiceAddress;
  BLOB                ServiceSpecificInfo;
} SERVICE_INFOW, *PSERVICE_INFOW, *LPSERVICE_INFOW;
```

Members

`lpServiceType`

Type: **LPGUID**

A pointer to a GUID that is the type of the network service.

`lpServiceName`

Type: **LPTSTR**

A pointer to a **NULL**-terminated string that is the name of the network service.

If you are calling the **SetService** function with the *dwNameSpace* parameter set to NS_DEFAULT, the network service name must be a common name. A common name is what the network service is commonly known as. An example of a common name for a network service is "My SQL Server".

If you are calling the **SetService** function with the *dwNameSpace* parameter set to a specific service name, the network service name can be a common name or a distinguished name. A distinguished name distinguishes the service to a unique location with a directory service. An example of a distinguished name for a network service is "MS\SYS\NT\DEV\My SQL Server".

`lpComment`

Type: **LPTSTR**

A pointer to a **NULL**-terminated string that is a comment or description for the network service. For example, "Used for development upgrades."

`lpLocale`

Type: **LPTSTR**

A pointer to a **NULL**-terminated string that contains locale information.

`dwDisplayHint`

Type: **DWORD**

A hint as to how to display the network service in a network browsing user interface. This can be one of the following values.

VALUE	MEANING
RESOURCEDISPLAYTYPE_DOMAIN	Display the network service as a domain.
RESOURCEDISPLAYTYPE_FILE	Display the network service as a file.
RESOURCEDISPLAYTYPE_GENERIC	The method used to display the object does not matter.
RESOURCEDISPLAYTYPE_GROUP	Display the network service as a group.
RESOURCEDISPLAYTYPE_SERVER	Display the network service as a server.
RESOURCEDISPLAYTYPE_SHARE	Display the network service as a sharepoint.
RESOURCEDISPLAYTYPE_TREE	Display the network service as a tree.

`dwVersion`

Type: **DWORD**

The version for the network service. The high word of this value specifies a major version number. The low word of this value specifies a minor version number.

`dwTime`

Type: **DWORD**

Reserved for future use. Must be zero.

`lpMachineName`

Type: **LPTSTR**

A pointer to a **NULL**-terminated string that is the name of the computer on which the network service is running.

`lpServiceAddress`

Type: **LPSERVICE_ADDRESSES**

A pointer to a SERVICE_ADDRESSES structure that contains an array of SERVICE_ADDRESS structures. Each **SERVICE_ADDRESS** structure contains information about a network service address.

A network service can call the getsockname function to determine the local address of the system.

`ServiceSpecificInfo`

Type: **BLOB**

A BLOB structure that specifies service-defined information.

> **Note** In general, the data pointed to by the BLOB structure's **pBlobData** member must not contain any pointers. That is because only the network service knows the format of the data; copying the data without such knowledge would lead to pointer invalidation. If the data pointed to by **pBlobData** contains variable-sized elements, offsets from **pBlobData** can be used to indicate the location of those elements. There is one exception to this general rule: when **pBlobData** points to a SERVICE_TYPE_INFO_ABS structure. This is possible because both the **SERVICE_TYPE_INFO_ABS** structure, and any SERVICE_TYPE_VALUE_ABS structures it contains are predefined, and thus their formats are known to the operating system.

Remarks

> **NOTE**
>
> The nspapi.h header defines SERVICE_INFO as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see Conventions for Function Prototypes.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	nspapi.h

See also

BLOB

GetService

NS_SERVICE_INFO

SERVICE_ADDRESS

SERVICE_ADDRESSES

SERVICE_TYPE_INFO_ABS

SERVICE_TYPE_VALUE_ABS

SetService

SERVICE_TYPE_INFO_ABSA structure (nspapi.h)

10/22/2022 • 2 minutes to read • Edit Online

The **SERVICE_TYPE_INFO_ABS** structure contains information about a network service type. Use **SERVICE_TYPE_INFO_ABS** to add a network service type to a namespace.

Syntax

```
typedef struct _SERVICE_TYPE_INFO_ABSA {
  LPSTR                   lpTypeName;
  DWORD                   dwValueCount;
  SERVICE_TYPE_VALUE_ABSA Values[1];
} SERVICE_TYPE_INFO_ABSA, *PSERVICE_TYPE_INFO_ABSA, *LPSERVICE_TYPE_INFO_ABSA;
```

Members

`lpTypeName`

Pointer to a zero-terminated string that is the name of the network service type. This name is the same in all namespaces, and is used by the GetTypeByName and **GetNameByType** functions.

`dwValueCount`

Number of SERVICE_TYPE_VALUE_ABS structures in the **Values** member array that follows **dwValueCount**.

`Values[1]`

Array of SERVICE_TYPE_VALUE_ABS structures.

Each of these structures contains information about a service type value that the operating system or network service may need when an instance of this network service type is registered with a namespace.

The information in these structures may be specific to a namespace. For example, if a network service uses the SAP namespace, but does not have a **GUID** that contains the SAP identifier (SAPID), it defines the SAPID in a SERVICE_TYPE_VALUE_ABS structure.

Remarks

When you use the SetService function to add a network service type to a namespace, the **SERVICE_TYPE_INFO_ABS** structure is passed as the **ServiceSpecificInfo** BLOB member of a SERVICE_INFO structure. Although the **ServiceSpecificInfo** member generally should not contain pointers, an exception is made in the case of the **SERVICE_TYPE_INFO_ABS** and SERVICE_TYPE_VALUE_ABS structures.

> **NOTE**
>
> The nspapi.h header defines SERVICE_TYPE_INFO_ABS as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see Conventions for Function Prototypes.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	nspapi.h

See also

SERVICE_INFO

SERVICE_TYPE_VALUE_ABS

SetService

SERVICE_TYPE_INFO_ABSW structure (nspapi.h)

10/22/2022 • 2 minutes to read • Edit Online

The **SERVICE_TYPE_INFO_ABS** structure contains information about a network service type. Use **SERVICE_TYPE_INFO_ABS** to add a network service type to a namespace.

Syntax

```
typedef struct _SERVICE_TYPE_INFO_ABSW {
  LPWSTR                  lpTypeName;
  DWORD                   dwValueCount;
  SERVICE_TYPE_VALUE_ABSW Values[1];
} SERVICE_TYPE_INFO_ABSW, *PSERVICE_TYPE_INFO_ABSW, *LPSERVICE_TYPE_INFO_ABSW;
```

Members

`lpTypeName`

Pointer to a zero-terminated string that is the name of the network service type. This name is the same in all namespaces, and is used by the GetTypeByName and **GetNameByType** functions.

`dwValueCount`

Number of SERVICE_TYPE_VALUE_ABS structures in the **Values** member array that follows **dwValueCount**.

`Values[1]`

Array of SERVICE_TYPE_VALUE_ABS structures.

Each of these structures contains information about a service type value that the operating system or network service may need when an instance of this network service type is registered with a namespace.

The information in these structures may be specific to a namespace. For example, if a network service uses the SAP namespace, but does not have a **GUID** that contains the SAP identifier (SAPID), it defines the SAPID in a SERVICE_TYPE_VALUE_ABS structure.

Remarks

When you use the SetService function to add a network service type to a namespace, the **SERVICE_TYPE_INFO_ABS** structure is passed as the **ServiceSpecificInfo** BLOB member of a SERVICE_INFO structure. Although the **ServiceSpecificInfo** member generally should not contain pointers, an exception is made in the case of the **SERVICE_TYPE_INFO_ABS** and SERVICE_TYPE_VALUE_ABS structures.

> **NOTE**
>
> The nspapi.h header defines SERVICE_TYPE_INFO_ABS as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see Conventions for Function Prototypes.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	nspapi.h

See also

SERVICE_INFO

SERVICE_TYPE_VALUE_ABS

SetService

SERVICE_TYPE_VALUE_ABSA structure (nspapi.h)

10/22/2022 • 2 minutes to read • Edit Online

The **SERVICE_TYPE_VALUE_ABS** structure contains information about a network-service type value. This information may be specific to a namespace.

Syntax

```
typedef struct _SERVICE_TYPE_VALUE_ABSA {
  DWORD dwNameSpace;
  DWORD dwValueType;
  DWORD dwValueSize;
  LPSTR lpValueName;
  PVOID lpValue;
} SERVICE_TYPE_VALUE_ABSA, *PSERVICE_TYPE_VALUE_ABSA, *LPSERVICE_TYPE_VALUE_ABSA;
```

Members

`dwNameSpace`

Type: **DWORD**

A namespace, or a set of default namespaces, for which the network service type value is intended. Namespace providers will look only at values intended for their namespace.

Use one of the following constants to specify a namespace:

VALUE	MEANING
NS_DEFAULT	A set of default namespaces. The function queries each namespace within this set. The set of default namespaces typically includes all the namespaces installed on the system. System administrators, however, can exclude particular namespaces from the set. NS_DEFAULT is the value that most applications should use for **dwNameSpace**.
NS_DNS	The Domain Name System used in the Internet for host name resolution.
NS_NETBT	The NetBIOS over TCP/IP layer. All Windows operating systems register their computer names with NetBIOS. This namespace is used to convert a computer name to an IP address that uses this registration. Note that NS_NETBT may access a WINS server to perform the resolution.
NS_SAP	The NetWare Service Advertising Protocol. This may access the NetWare bindery if appropriate. NS_SAP is a dynamic namespace that allows registration of services.
NS_TCPIP_HOSTS	Lookup value in the <systemroot>\system32\drivers\etc\hosts file.

NS_TCPIP_LOCAL	Local TCP/IP name resolution mechanisms, including comparisons against the local host name and looks up host names and IP addresses in cache of host to IP address mappings.

`dwValueType`

Type: **DWORD**

The type of the value data. Specify one of the following types:

VALUE	MEANING
REG_BINARY	Binary data in any form.
REG_DWORD	A 32-bit number.
REG_MULTI_SZ	An array of null-terminated strings, terminated by two null characters.
REG_SZ	A null-terminated string.

`dwValueSize`

Type: **DWORD**

The size, in bytes, of the value pointed to by the **lpValue** member. In the case of REG_SZ and REG_MULTI_SZ string data, the terminating characters are counted as part of the size.

`lpValueName`

Type: **LPTSTR**

A pointer to a **NULL**-terminated string that is the name of the value. This name is specific to a namespace.

Several commonly used value name strings are associated with defined constants. These name strings include the following.

CONSTANT	MEANING
SERVICE_TYPE_VALUE_SAPID	"SapId"
SERVICE_TYPE_VALUE_CONN	"ConnectionOriented"
SERVICE_TYPE_VALUE_TCPPORT	"TcpPort"
SERVICE_TYPE_VALUE_UDPPORT	"UdpPort"

`lpValue`

Type: **PVOID**

A pointer to the value data.

Remarks

When you use the SetService function to add a network service type to a namespace, a SERVICE_TYPE_INFO_ABS structure is passed as the **ServiceSpecificInfo** BLOB member of a SERVICE_INFO structure. Although the **ServiceSpecificInfo** member generally should not contain pointers, an exception is made in the case of the **SERVICE_TYPE_INFO_ABS** and **SERVICE_TYPE_VALUE_ABS** structures.

> **NOTE**
> The nspapi.h header defines SERVICE_TYPE_VALUE_ABS as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see Conventions for Function Prototypes.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	nspapi.h

See also

SERVICE_INFO

SERVICE_TYPE_INFO_ABS

SetService

SERVICE_TYPE_VALUE_ABSW structure (nspapi.h)

10/22/2022 • 2 minutes to read • Edit Online

The **SERVICE_TYPE_VALUE_ABS** structure contains information about a network-service type value. This information may be specific to a namespace.

Syntax

```
typedef struct _SERVICE_TYPE_VALUE_ABSW {
  DWORD  dwNameSpace;
  DWORD  dwValueType;
  DWORD  dwValueSize;
  LPWSTR lpValueName;
  PVOID  lpValue;
} SERVICE_TYPE_VALUE_ABSW, *PSERVICE_TYPE_VALUE_ABSW, *LPSERVICE_TYPE_VALUE_ABSW;
```

Members

`dwNameSpace`

Type: **DWORD**

A namespace, or a set of default namespaces, for which the network service type value is intended. Namespace providers will look only at values intended for their namespace.

Use one of the following constants to specify a namespace:

VALUE	MEANING
NS_DEFAULT	A set of default namespaces. The function queries each namespace within this set. The set of default namespaces typically includes all the namespaces installed on the system. System administrators, however, can exclude particular namespaces from the set. NS_DEFAULT is the value that most applications should use for **dwNameSpace**.
NS_DNS	The Domain Name System used in the Internet for host name resolution.
NS_NETBT	The NetBIOS over TCP/IP layer. All Windows operating systems register their computer names with NetBIOS. This namespace is used to convert a computer name to an IP address that uses this registration. Note that NS_NETBT may access a WINS server to perform the resolution.
NS_SAP	The NetWare Service Advertising Protocol. This may access the NetWare bindery if appropriate. NS_SAP is a dynamic namespace that allows registration of services.
NS_TCPIP_HOSTS	Lookup value in the <systemroot>\system32\drivers\etc\hosts file.

NS_TCPIP_LOCAL	Local TCP/IP name resolution mechanisms, including comparisons against the local host name and looks up host names and IP addresses in cache of host to IP address mappings.

`dwValueType`

Type: **DWORD**

The type of the value data. Specify one of the following types:

VALUE	MEANING
REG_BINARY	Binary data in any form.
REG_DWORD	A 32-bit number.
REG_MULTI_SZ	An array of null-terminated strings, terminated by two null characters.
REG_SZ	A null-terminated string.

`dwValueSize`

Type: **DWORD**

The size, in bytes, of the value pointed to by the **lpValue** member. In the case of REG_SZ and REG_MULTI_SZ string data, the terminating characters are counted as part of the size.

`lpValueName`

Type: **LPTSTR**

A pointer to a **NULL**-terminated string that is the name of the value. This name is specific to a namespace.

Several commonly used value name strings are associated with defined constants. These name strings include the following.

CONSTANT	MEANING
SERVICE_TYPE_VALUE_SAPID	"SapId"
SERVICE_TYPE_VALUE_CONN	"ConnectionOriented"
SERVICE_TYPE_VALUE_TCPPORT	"TcpPort"
SERVICE_TYPE_VALUE_UDPPORT	"UdpPort"

`lpValue`

Type: **PVOID**

A pointer to the value data.

Remarks

When you use the SetService function to add a network service type to a namespace, a SERVICE_TYPE_INFO_ABS structure is passed as the **ServiceSpecificInfo** BLOB member of a SERVICE_INFO structure. Although the **ServiceSpecificInfo** member generally should not contain pointers, an exception is made in the case of the **SERVICE_TYPE_INFO_ABS** and **SERVICE_TYPE_VALUE_ABS** structures.

> **NOTE**
> The nspapi.h header defines SERVICE_TYPE_VALUE_ABS as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see Conventions for Function Prototypes.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	nspapi.h

See also

SERVICE_INFO

SERVICE_TYPE_INFO_ABS

SetService

SetServiceA function (nspapi.h)

10/22/2022 • 3 minutes to read • Edit Online

The **SetService** function registers or removes from the registry a network service within one or more namespaces. The function can also add or remove a network service type within one or more namespaces.

> **Note** The **SetService** function is obsolete. The functions detailed in Protocol-Independent Name Resolution provide equivalent functionality in Windows Sockets 2. For the convenience of Windows Sockets 1.1 developers, the reference material is as follows.

Syntax

```
INT SetServiceA(
  [in]           DWORD                dwNameSpace,
  [in]           DWORD                dwOperation,
  [in]           DWORD                dwFlags,
  [in]           LPSERVICE_INFOA      lpServiceInfo,
  [in, optional] LPSERVICE_ASYNC_INFO lpServiceAsyncInfo,
  [out]          LPDWORD              lpdwStatusFlags
);
```

Parameters

`[in] dwNameSpace`

The namespace, or a set of default namespaces, within which the function will operate.

Use one of the following constants to specify a namespace.

VALUE	MEANING
NS_DEFAULT	A set of default namespaces. The function queries each namespace within this set. The set of default namespaces typically includes all the namespaces installed on the system. System administrators, however, can exclude particular namespaces from the set. NS_DEFAULT is the value that most applications should use for *dwNameSpace*.
NS_DNS	The Domain Name System used in the Internet to resolve the name of the host.
NS_NDS	The NetWare 4 provider.
NS_NETBT	The NetBIOS over TCP/IP layer. All Windows systems register their computer names with NetBIOS. This namespace is used to convert a computer name to an IP address that uses this registration.

NS_SAP	The NetWare Service Advertising Protocol. This can access the NetWare bindery, if appropriate. NS_SAP is a dynamic namespace that enables the registration of services.
NS_TCPIP_HOSTS	Lookup value in the <systemroot>\system32\drivers\etc\posts file.
NS_TCPIP_LOCAL	Local TCP/IP name resolution mechanisms, including comparisons against the local host name and lookup value in the cache of host to IP address mappings.

`[in] dwOperation`

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

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

`[in] dwFlags`

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

VALUE	MEANING
SERVICE_FLAG_DEFER	This bit flag is valid only if the operation is SERVICE_REGISTER or SERVICE_DEREGISTER. If this bit flag is one, and it is valid, the namespace provider should defer the registration or deregistration operation until a SERVICE_FLUSH operation is requested.

SERVICE_FLAG_HARD	This bit flag is valid only if the operation is SERVICE_REGISTER or SERVICE_DEREGISTER. If this bit flag is one, and it is valid, the namespace provider updates any relevant persistent store information when the operation is performed. For example: If the operation involves deregistration in a namespace that uses a persistent store, the namespace provider would remove the relevant persistent store information.

`[in] lpServiceInfo`

A pointer to a SERVICE_INFO structure that contains information about the network service or service type.

`[in, optional] lpServiceAsyncInfo`

Reserved for future use. Must be set to **NULL**.

`[out] lpdwStatusFlags`

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

VALUE	MEANING
SET_SERVICE_ PARTIAL_SUCCESS	One or more namespace providers were unable to successfully perform the requested operation.

Return value

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

ERROR CODE	MEANING
ERROR_ALREADY_ REGISTERED	The function tried to register a service that was already registered.

Remarks

> **NOTE**
>
> The nspapi.h header defines SetService as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see Conventions for Function Prototypes.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]

Target Platform	Windows
Header	nspapi.h
Library	Mswsock.lib
DLL	Mswsock.dll

See also

GetService

SERVICE_INFO

SERVICE_TYPE_INFO_ABS

Winsock Functions

Winsock Reference

SetServiceW function (nspapi.h)

10/22/2022 • 3 minutes to read • Edit Online

The **SetService** function registers or removes from the registry a network service within one or more namespaces. The function can also add or remove a network service type within one or more namespaces.

> **Note** The **SetService** function is obsolete. The functions detailed in Protocol-Independent Name Resolution provide equivalent functionality in Windows Sockets 2. For the convenience of Windows Sockets 1.1 developers, the reference material is as follows.

Syntax

```
INT SetServiceW(
  [in]           DWORD                dwNameSpace,
  [in]           DWORD                dwOperation,
  [in]           DWORD                dwFlags,
  [in]           LPSERVICE_INFOW      lpServiceInfo,
  [in, optional] LPSERVICE_ASYNC_INFO lpServiceAsyncInfo,
  [out]          LPDWORD              lpdwStatusFlags
);
```

Parameters

`[in] dwNameSpace`

The namespace, or a set of default namespaces, within which the function will operate.

Use one of the following constants to specify a namespace.

VALUE	MEANING
NS_DEFAULT	A set of default namespaces. The function queries each namespace within this set. The set of default namespaces typically includes all the namespaces installed on the system. System administrators, however, can exclude particular namespaces from the set. NS_DEFAULT is the value that most applications should use for *dwNameSpace*.
NS_DNS	The Domain Name System used in the Internet to resolve the name of the host.
NS_NDS	The NetWare 4 provider.
NS_NETBT	The NetBIOS over TCP/IP layer. All Windows systems register their computer names with NetBIOS. This namespace is used to convert a computer name to an IP address that uses this registration.

NS_SAP	The NetWare Service Advertising Protocol. This can access the NetWare bindery, if appropriate. NS_SAP is a dynamic namespace that enables the registration of services.
NS_TCPIP_HOSTS	Lookup value in the <systemroot>\system32\drivers\etc\posts file.
NS_TCPIP_LOCAL	Local TCP/IP name resolution mechanisms, including comparisons against the local host name and lookup value in the cache of host to IP address mappings.

`[in] dwOperation`

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

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

`[in] dwFlags`

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

VALUE	MEANING
SERVICE_FLAG_DEFER	This bit flag is valid only if the operation is SERVICE_REGISTER or SERVICE_DEREGISTER. If this bit flag is one, and it is valid, the namespace provider should defer the registration or deregistration operation until a SERVICE_FLUSH operation is requested.

SERVICE_FLAG_HARD	This bit flag is valid only if the operation is SERVICE_REGISTER or SERVICE_DEREGISTER. If this bit flag is one, and it is valid, the namespace provider updates any relevant persistent store information when the operation is performed. For example: If the operation involves deregistration in a namespace that uses a persistent store, the namespace provider would remove the relevant persistent store information.

`[in] lpServiceInfo`

A pointer to a SERVICE_INFO structure that contains information about the network service or service type.

`[in, optional] lpServiceAsyncInfo`

Reserved for future use. Must be set to **NULL**.

`[out] lpdwStatusFlags`

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

VALUE	MEANING
SET_SERVICE_ PARTIAL_SUCCESS	One or more namespace providers were unable to successfully perform the requested operation.

Return value

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

ERROR CODE	MEANING
ERROR_ALREADY_ REGISTERED	The function tried to register a service that was already registered.

Remarks

> **NOTE**
>
> The nspapi.h header defines SetService as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see Conventions for Function Prototypes.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]

Target Platform	Windows
Header	nspapi.h
Library	Mswsock.lib
DLL	Mswsock.dll

See also

GetService

SERVICE_INFO

SERVICE_TYPE_INFO_ABS

Winsock Functions

Winsock Reference

socketapi.h header

10/22/2022 • 2 minutes to read • Edit Online

This header is used by Windows Sockets 2. For more information, see:

- Windows Sockets 2

socketapi.h contains the following programming interfaces:

Functions

SetSocketMediaStreamingMode Indicates whether the network is to be used for transferring streaming media that requires quality of service.

SetSocketMediaStreamingMode function (socketapi.h)

10/22/2022 • 2 minutes to read • Edit Online

The **SetSocketMediaStreamingMode** function indicates whether the network is to be used for transferring streaming media that requires quality of service.

Syntax

```
HRESULT SetSocketMediaStreamingMode(
  BOOL value
);
```

Parameters

`value`

Indicates whether the network is to be used for transferring streaming media that requires quality of service. This ensures that sockets opened as low latency will get the right quality of service over 802.11 wireless networks.

Return value

If no error occurs, **SetSocketMediaStreamingMode** returns S_OK. Otherwise, an error code is returned as an HRESULT.

Remarks

The **SetSocketMediaStreamingMode** function is used to indicate whether the network is to be used for transferring streaming media that requires quality of service. This function is normally used by Voice over IP (VoIP) or similar apps that require a consistent quality of service. The **SetSocketMediaStreamingMode** function can be used by Windows Store apps or desktop apps.

There can be quality of service issues for media streaming when used over an 802.11 wireless network. The 802.11 network driver will periodically scan for other nearby infrastructure networks (ESS) or ad-hoc networks (IBSS). This allows the wireless network adapter to find other networks and possibly connected to a network with a stronger signal. Most current 802.11 network drivers scan all of the available channels as a series at once. When the 802.11 network driver is scanning for other networks and listening on other channels, it cannot receive packets for the app. The time spent scanning for other networks can introduce a noticeable gap (100 milliseconds or more) when a VoIP app would be unable to receive the audio stream. This scanning process is longer for 802.11 network adapters that are dual band (2.4GHz and 5GHz) since even more channels are scanned. This can result in the audio to be perceived as stuttering.

When the **SetSocketMediaStreamingMode** function is called with the *value* parameter set to **TRUE** and the socket will be transferring over an 802.11 wireless network adapter, the system will notify the wireless network driver to stop scanning for other networks. This eliminates stuttering by VoIP and similar audio apps when used over 802.11 wireless networks, but also affects any apps running on the local computer or device.

There are cases where turning off scans may cause problems. When scans are disabled, the local computer stays

connected to the same network even if the signal becomes weaker and weaker as the user moves away from the network.

A VoIP or similar app should close all low latency sockets to restore the media streaming mode of the 802.11 wireless network driver. This will re-enable scanning for other wireless networks.

The **SetSocketMediaStreamingMode** function has no effect if the socket will not be sending or receiving packets over an 802.11 wireless adapter.

Requirements

Minimum supported client	Windows 8 [desktop apps | UWP apps]
Minimum supported server	Windows Server 2012 [desktop apps | UWP apps]
Target Platform	Windows
Header	socketapi.h
Library	Windows.Networking.lib
DLL	Windows.Networking.dll

See also

Adding support for networking

Windows.Networking.BackgroundTransfer

Windows.Networking.Sockets

sporder.h header

10/22/2022 • 2 minutes to read • Edit Online

This header is used by Windows Sockets 2. For more information, see:

- Windows Sockets 2

sporder.h contains the following programming interfaces:

Functions

WSCWriteNameSpaceOrder Changes the order of available Windows Sockets (Winsock) 2 namespace providers. The order of the namespace providers determines the priority of the namespace when enumerated or queried for name resolution.
WSCWriteNameSpaceOrder32 Changes the order of available Windows Sockets (Winsock) 2 namespace providers in a 32-bit catalog.
WSCWriteProviderOrder Used to reorder the available transport providers.
WSCWriteProviderOrder32 Used to reorder the available 32-bit transport providers.

WSCWriteNameSpaceOrder function (sporder.h)

10/22/2022 • 4 minutes to read • Edit Online

The **WSCWriteNameSpaceOrder** function changes the order of available Windows Sockets (Winsock) 2 namespace providers. The order of the namespace providers determines the priority of the namespace when enumerated or queried for name resolution.

Syntax

```
int WSCWriteNameSpaceOrder(
  [in] LPGUID lpProviderId,
  [in] DWORD  dwNumberOfEntries
);
```

Parameters

`[in] lpProviderId`

An array of **NSProviderId** elements as found in the WSANAMESPACE_INFO structure. The order of the **NSProviderId** elements is the new priority ordering for the namespace providers.

`[in] dwNumberOfEntries`

The number of elements in the **NSProviderId** array.

Return value

The function returns **ERROR_SUCCESS** (zero) if the routine is successful. Otherwise, it returns a specific error code.

ERROR CODE	MEANING
WSAEFAULT	The **NSProviderId** array is not fully contained within process address space.
WSAEINVAL	One or more of the arguments are input parameters were invalid, no action was taken.
WSANO_RECOVERY	A nonrecoverable error occurred. This error is returned under several conditions including the following: the user lacks the administrative privileges required to write to the Winsock registry or another application is currently writing to the namespace provider catalog.
WSASYSCALLFAILURE	A system call that should never fail has failed.
WSATRY_AGAIN	The function is called by another thread or process.

WSA_NOT_ENOUGH_MEMORY	Insufficient memory was available to perform the operation.
(other)	The function may return any registry error code.

Remarks

Namespace providers are installed using the WSCInstallNameSpace function. The order in which namespace providers are initially installed governs the default order in which they are enumerated through WSAEnumNameSpaceProviders. More importantly, this order also governs the order in which namespace providers are considered when a client requests name resolution. The order of namespace providers can be changed using the **WSCWriteNameSpaceOrder** function. On 64-bit platforms, the WSCWriteNameSpaceOrder32 function is provided to allow 64-bit processes to change the order of namespace providers in the 32-bit namespace provider catalog. On 64-bit platforms, namespace providers are installed in the 32-bit namespace provider catalog using the WSCInstallNameSpace32 function.

The current namespace provider catalog is stored in the registry under the following registry key:
HKEY_LOCAL_MACHINE\SYSTEM\Current Control Set\Services\Winsock2\Parameters\NameSpace_Catalog5

A client request for name resolution uses the WSALookupServiceBegin, WSALookupServiceNext, and WSALookupServiceEnd routines. The **dwNameSpace** member of the WSAQUERYSET structure passed to **WSALookupServiceBegin** is set to the identifier of a single namespace (NS_DNS, for example) in which to constrain the search, or **NS_ALL** to include all namespaces. If multiple namespace providers support a specific namespace (**NS_DNS**, for example), then the results from all namespace providers that match the requested **dwNameSpace** are returned unless the **lpNSProviderId** member is set to a specific namespace provider. The results from all namespace providers is returned if NS_ALL is specified for the **dwNameSpace** member. The order that the results are returned is dependent on the namespace provider order in the catalog.

The Windows SDK includes an application called SpOrder.exe that allows the catalog of installed namespace providers to be displayed. Windows Sockets 2 includes the ws2_32.dll that exports the **WSCWriteNameSpaceOrder** function for reordering namespace providers in the catalog. This interface can be imported by linking with WS2_32.lib. For computers running on Windows XP with Service Pack 2 (SP2) and Windows Server 2003 with Service Pack 1 (SP1) and later, the **netsh.exe winsock show catalog** command will display both the protocol and namespace providers installed on the system.

WSCWriteNameSpaceOrder can only be called by a user logged on as a member of the Administrators group. If **WSCWriteNameSpaceOrder** is called by a user that is not a member of the Administrators group, the function call will fail and **WSANO_RECOVERY** is returned in the *lpErrno* parameter.

For computers running on Windows Vista and Windows Vista, this function can also fail because of user account control (UAC). If an application that contains this function is executed by a user logged on as a member of the Administrators group other than the Administrator, this call will fail unless the application has been marked in the manifest file with a **requestedExecutionLevel** set to **requireAdministrator**. If the application on Windows Vista and Windows Vista lacks this setting in the manifest file used to build the executable file, a user logged on as a member of the Administrators group other than the Administrator must then be executing the application in an enhanced shell as the Administrator (**RunAs administrator**) for this function to succeed.

The following list describes scenarios in which the **WSCWriteNameSpaceOrder** function could fail:

- The *dwNumberOfEntries* parameter is not equal to the number of registered namespace providers.
- The **NSProviderId** array contains an invalid namespace provider identifier.
- The **NSProviderId** array does not contain all valid namespace provider identifiers exactly one time.

- The function cannot access the registry (for example, insufficient user permissions).
- Another process (or thread) is currently calling the function.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	sporder.h
Library	Sporder.lib
DLL	Ws2_32.dll

See also

WSAEnumNameSpaceProviders

WSANAMESPACE_INFO

WSAQUERYSET

WSCEnumNameSpaceProviders32

WSCInstallNameSpace

WSCInstallNameSpace32

WSCWriteNameSpaceOrder32

WSCWriteNameSpaceOrder32 function (sporder.h)

10/22/2022 • 4 minutes to read • Edit Online

The **WSCWriteNameSpaceOrder32** function changes the order of available Windows Sockets (Winsock) 2 namespace providers in a 32-bit catalog. The order of the namespace providers determines the priority of the namespace when enumerated or queried for name resolution.

> **Note** This call is a strictly 32-bit version of WSCWriteNameSpaceOrder for use on 64-bit platforms. It is provided to allow 64-bit processes to reorder the 32-bit namespace provider catalog.

Syntax

```
int WSCWriteNameSpaceOrder32(
  [in] LPGUID lpProviderId,
  [in] DWORD  dwNumberOfEntries
);
```

Parameters

`[in] lpProviderId`

An array of **NSProviderId** elements as found in the WSANAMESPACE_INFO structure. The order of the **NSProviderId** elements is the new priority ordering for the namespace providers.

`[in] dwNumberOfEntries`

The number of elements in the **NSProviderId** array.

Return value

The function returns **ERROR_SUCCESS** (zero) if the routine is successful. Otherwise, it returns a specific error code.

ERROR CODE	MEANING
WSAEFAULT	The **NSProviderId** array is not fully contained within process address space.
WSAEINVAL	Input parameters were invalid, no action was taken.
WSANO_RECOVERY	A nonrecoverable error occurred. This error is returned under several conditions including the following: the Winsock registry could not be opened, the user lacks the administrative privileges required to write to the Winsock registry, or another application is currently writing to the namespace provider catalog.

WSASYSCALLFAILURE	A system call that should never fail has failed.
WSATRY_AGAIN	The function is called by another thread or process.
WSA_NOT_ENOUGH_MEMORY	Insufficient memory was available to perform the operation.
(other)	The function may return any registry error code.

Remarks

Namespace providers are installed on 64-bit platforms in a 32-bit namespace provider catalog using the WSCInstallNameSpace32 function. The order in which namespace providers in a 32-bit catalog are initially installed governs the default order in which they are enumerated through WSCEnumNameSpaceProviders32. More importantly, this order also governs the order in which namespace providers are considered when a client requests name resolution. On 64-bit platforms, the **WSCWriteNameSpaceOrder32** function is provided to allow 64-bit processes to change the order of namespace providers in the 32-bit namespace provider catalog. The order of namespace providers in the native catalog can be changed using the WSCWriteNameSpaceOrder function.

The current namespace provider catalog is stored in the registry under the following registry key: **HKEY_LOCAL_MACHINE\SYSTEM\Current Control Set\Services\Winsock2\Parameters\NameSpace_Catalog5**

A client request for name resolution uses the WSALookupServiceBegin, WSALookupServiceNext, and WSALookupServiceEnd routines. The **dwNameSpace** member of the WSAQUERYSET structure passed to **WSALookupServiceBegin** is set to the identifier of a single namespace (**NS_DNS**, for example) in which to constrain the search, or **NS_ALL** to include all namespaces. If multiple namespace providers support a specific namespace (for example, **NS_DNS**), then the results from all namespace providers that match the requested **dwNameSpace** are returned unless the **lpNSProviderId** member is set to a specific namespace provider. The results from all namespace providers is returned if **NS_ALL** is specified for the **dwNameSpace** member. The order that the results are returned depends on the namespace provider order in the catalog.

The Windows SDK includes an application called SpOrder.exe that allows the catalog of installed namespace providers to be displayed. Winsock 2 includes the ws2_32.DLL on 64-bit platforms that exports the **WSCWriteNameSpaceOrder32** function for reordering namespace providers in the 32-bit namespace provider catalog. This interface can be imported by linking with WS2_32.lib. For computers running on Windows XP with Service Pack 2 (SP2) and Windows Server 2003 with Service Pack 1 (SP1) and later, the **netsh.exe winsock show catalog** command will display both the protocol and namespace providers installed. The native 64-bit catalog is displayed first followed by the 32-bit provider catalogs (denoted with a 32 after their name).

WSCWriteNameSpaceOrder32 can only be called by a user logged on as a member of the Administrators group. If **WSCWriteNameSpaceOrder32** is called by a user that is not a member of the Administrators group, the function call will fail and **WSANO_RECOVERY** is returned in the lpErrno parameter.

For computers running on Windows Vista and Windows Vista, this function can also fail because of user account control (UAC). If an application that contains this function is executed by a user logged on as a member of the Administrators group other than the Administrator, this call will fail unless the application has been marked in the manifest file with a **requestedExecutionLevel** set to **requireAdministrator**. If the application on

Windows Vista and Windows Vista lacks this setting in the manifest file used to build the executable file, a user logged on as a member of the Administrators group other than the Administrator must then be executing the application in an enhanced shell as the Administrator (**RunAs administrator**) for this function to succeed.

The following list describes scenarios in which the **WSCWriteNameSpaceOrder32** function could fail:

- The *dwNumberOfEntries* parameter is not equal to the number of registered namespace providers.
- The **NSProviderId** array contains an invalid namespace provider identifier.
- The **NSProviderId** array does not contain all valid namespace provider identifiers exactly one time.
- The function is not able to access the registry for some reason (insufficient user permissions, for example).
- Another process, or thread, is currently calling the function.

Requirements

Minimum supported client	Windows Vista, Windows XP Professional x64 Edition [desktop apps only]
Minimum supported server	Windows Server 2008, Windows Server 2003 x64 Edition [desktop apps only]
Target Platform	Windows
Header	sporder.h
Library	Sporder.lib
DLL	Ws2_32.dll

See also

WSAEnumNameSpaceProviders

WSANAMESPACE_INFO

WSAQUERYSET

WSCEnumNameSpaceProviders32

WSCInstallNameSpace

WSCInstallNameSpace32

WSCWriteNameSpaceOrder

WSCWriteProviderOrder function (sporder.h)

10/22/2022 • 2 minutes to read • Edit Online

The **WSCWriteProviderOrder** function is used to reorder the available transport providers. The order of the protocols determines the priority of a protocol when being enumerated or selected for use.

Syntax

```
int WSCWriteProviderOrder(
  [in] LPDWORD lpwdCatalogEntryId,
  [in] DWORD   dwNumberOfEntries
);
```

Parameters

`[in] lpwdCatalogEntryId`

A pointer to an array of **CatalogEntryId** elements found in the WSAPROTOCOL_INFO structure. The order of the **CatalogEntryId** elements is the new priority ordering for the protocols.

`[in] dwNumberOfEntries`

The number of elements in the *lpwdCatalogEntryId* array.

Return value

The function returns **ERROR_SUCCESS** (zero) if the routine is successful. Otherwise, it returns a specific error code.

ERROR CODE	MEANING
WSAEINVAL	One or more of the arguments are invalid, no action was taken.
WSANO_RECOVERY	A nonrecoverable error occurred. This error is returned under several conditions including the following: the user lacks the administrative privileges required to write to the Winsock registry, or a failure occurred when opening or writing a catalog entry.
WSA_NOT_ENOUGH_MEMORY	Insufficient memory was available. This error is returned when there is insufficient memory to allocate a new catalog entry.
(other)	The routine may return any registry error code.

Remarks

The order in which transport service providers are initially installed governs the order in which they are enumerated through WSCEnumProtocols 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 application 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 this procedural interface for reordering protocols. This interface can be imported by linking with *Sporder.lib*.

The following are scenarios in which the **WSCWriteProviderOrder** function could fail:

- The *dwNumberOfEntries* parameter is not equal to the number of registered service providers.
- The *lpwdCatalogEntryId* contains an invalid catalog identifier.
- The *lpwdCatalogEntryId* does not contain all valid catalog identifiers exactly one time.
- The routine is not able to access the registry for some reason (for example, inadequate user permissions).
- Another process (or thread) is currently calling the function.

On success, **WSCWriteProviderOrder** will attempt to alert all interested applications that have registered for notification of the change by calling WSAProviderConfigChange.

The **WSCWriteProviderOrder** function can only be called by a user logged on as a member of the Administrators group. If **WSCWriteProviderOrder** is called by a user that is not a member of the Administrators group, the function call will fail and WSANO_RECOVERY is returned. For computers running on Windows Vista or Windows Server 2008, this function can also fail because of user account control (UAC). If an application that contains this function 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 on Windows Vista or Windows Server 2008 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.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	sporder.h
Library	Sporder.lib
DLL	Ws2_32.dll

See also

WSAEnumProtocols

WSAPROTOCOL_INFO

WSAProviderConfigChange

WSCWriteProviderOrder32 function (sporder.h)

10/22/2022 • 3 minutes to read • Edit Online

The **WSCWriteProviderOrder32** function is used to reorder the available 32-bit transport providers. The order of the protocols determines the priority of a protocol when being enumerated or selected for use.

> **Note** This call is a strictly 32-bit version of WSCWriteProviderOrder for use on 64-bit platforms. It is provided to allow 64-bit processes to modify the 32-bit catalogs.

Syntax

```
int WSCWriteProviderOrder32(
  [in] LPDWORD lpwdCatalogEntryId,
  [in] DWORD   dwNumberOfEntries
);
```

Parameters

`[in] lpwdCatalogEntryId`

A pointer to an array of **CatalogEntryId** elements found in the WSAPROTOCOL_INFO structure. The order of the **CatalogEntryId** elements is the new priority ordering for the protocols.

`[in] dwNumberOfEntries`

The number of elements in the *lpwdCatalogEntryId* array.

Return value

The function returns **ERROR_SUCCESS** (zero) if the routine is successful. Otherwise, it returns a specific error code.

ERROR CODE	MEANING
WSAEINVAL	One or more of the arguments are invalid, no action was taken.
WSANO_RECOVERY	A nonrecoverable error occurred. This error is returned under several conditions including the following: the user lacks the administrative privileges required to write to the Winsock registry, or a failure occurred when opening or writing a catalog entry.
WSA_NOT_ENOUGH_MEMORY	Insufficient memory was available. This error is returned when there is insufficient memory to allocate a new catalog entry.

(other)	The routine may return any registry error code.

Remarks

The **WSCWriteProviderOrder32** function is a strictly 32-bit version of the WSCWriteProviderOrder function. On a 64-bit computer, all calls not specifically 32-bit (for example, all functions that do not end in "32") operate on the native 64-bit catalog. Processes that execute on a 64-bit computer must use the specific 32-bit function calls to operate on a strictly 32-bit catalog and preserve compatibility. The definitions and semantics of the specific 32-bit calls are the same as their native counterparts.

The order in which transport service providers are initially installed governs the order in which they are enumerated through 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 application 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 this procedural interface for reordering protocols. This interface can be imported by linking with *Sporder.lib*.

The following are scenarios in which the **WSCWriteProviderOrder32** function could fail:

- The *dwNumberOfEntries* parameter is not equal to the number of registered service providers.
- The *lpwdCatalogEntryId* contains an invalid catalog identifier.
- The *lpwdCatalogEntryId* does not contain all valid catalog identifiers exactly one time.
- The routine is not able to access the registry for some reason (for example, inadequate user permissions).
- Another process (or thread) is currently calling the function.

On success, **WSCWriteProviderOrder32** will attempt to alert all interested applications that have registered for notification of the change by calling WSAProviderConfigChange.

The **WSCWriteProviderOrder32** function can only be called by a user logged on as a member of the Administrators group. If **WSCWriteProviderOrder32** is called by a user that is not a member of the Administrators group, the function call will fail and WSANO_RECOVERY is returned. For computers running on Windows Vista or Windows Server 2008, this function can also fail because of user account control (UAC). If an application that contains this function 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 on Windows Vista or Windows Server 2008 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.

Requirements

Minimum supported client	Windows Vista, Windows XP Professional x64 Edition [desktop apps only]
Minimum supported server	Windows Server 2008, Windows Server 2003 x64 Edition [desktop apps only]

Target Platform	Windows
Header	sporder.h
Library	Sporder.lib
DLL	Ws2_32.dll

See also

WSAEnumProtocols

WSAPROTOCOL_INFO

WSAProviderConfigChange

WSCEnumProtocols32

transportsettingcommon.h header

10/22/2022 • 2 minutes to read • Edit Online

This header is used by Windows Sockets 2. For more information, see:

- Windows Sockets 2

transportsettingcommon.h contains the following programming interfaces:

Structures

TRANSPORT_SETTING_ID	The TRANSPORT_SETTING_ID (transportsettingcommon.h) structure specifies the transport setting ID used by the SIO_APPLY_TRANSPORT_SETTING and SIO_QUERY_TRANSPORT_SETTING IOCTLs.

TRANSPORT_SETTING_ID structure (transportsettingcommon.h)

10/22/2022 • 2 minutes to read • Edit Online

The **TRANSPORT_SETTING_ID** structure specifies the transport setting ID used by the SIO_APPLY_TRANSPORT_SETTING and SIO_QUERY_TRANSPORT_SETTING IOCTLs to apply or query the transport setting for a socket.

Syntax

```
typedef struct TRANSPORT_SETTING_ID {
  GUID Guid;
} TRANSPORT_SETTING_ID, *PTRANSPORT_SETTING_ID;
```

Members

`Guid`

The transport setting ID.

Remarks

The only transport setting defined for Windows 8 and Windows Server 2012 is for the **REAL_TIME_NOTIFICATION_CAPABILITY** capability on a TCP socket. For Windows 10 and Windows Server 2016, there is another transport setting defined as **ASSOCIATE_NAMERES_CONTEXT**.

The **TRANSPORT_SETTING_ID** structure is passed as input to the SIO_APPLY_TRANSPORT_SETTING and SIO_QUERY_TRANSPORT_SETTING IOCTLs. The **Guid** member determines what transport setting is applied or queried.

The only transport setting currently defines is for the **REAL_TIME_NOTIFICATION_CAPABILITY** capability on a TCP socket.

Requirements

Minimum supported client	Windows 8 [desktop apps only]
Minimum supported server	Windows Server 2012 [desktop apps only]
Header	transportsettingcommon.h (include Mstcpip.h)

See also

ControlChannelTrigger

REAL_TIME_NOTIFICATION_SETTING_OUTPUT

SIO_APPLY_TRANSPORT_SETTING

SIO_QUERY_TRANSPORT_SETTING

winsock.h header

10/22/2022 • 4 minutes to read • Edit Online

This header is used by Windows Sockets 2. For more information, see:

- Windows Sockets 2

winsock.h contains the following programming interfaces:

Functions

__WSAFDIsSet The __WSAFDIsSet function (winsock.h) specifies whether a socket is included in a set of socket descriptors.
AcceptEx The AcceptEx function (winsock.h) accepts a new connection, returns the local and remote address, and receives the first block of data sent by the client application.
bind The bind function (winsock.h) associates a local address with a socket.
closesocket The closesocket function (winsock.h) closes an existing socket.
FD_SET The FD_SET macro (winsock.h) structure is used by Windows Sockets (Winsock) functions and service providers to place sockets into a set.
GetAcceptExSockaddrs The GetAcceptExSockaddrs function (winsock.h) parses the data obtained from a call to the AcceptEx function and passes the local and remote addresses to a sockaddr structure.
gethostbyaddr As of Windows Sockets 2, the gethostbyaddr function (winsock.h) is no longer recommended for use.
gethostbyname The gethostbyname function (winsock.h) retrieves host information corresponding to a host name from a host database.
gethostname The gethostname function (winsock.h) retrieves the standard host name for the local computer.

getpeername

The getpeername function (winsock.h) retrieves the address of the peer to which a socket is connected.

getprotobyname

The getprotobyname function (winsock.h) retrieves the protocol information corresponding to a protocol name.

getprotobynumber

The getprotobynumber function (winsock.h) retrieves protocol information corresponding to a protocol number.

getservbyname

The getservbyname function (winsock.h) retrieves service information corresponding to a service name and protocol.

getservbyport

The getservbyport function (winsock.h) retrieves service information corresponding to a port and protocol.

getsockname

The getsockname function (winsock.h) retrieves the local name for a socket.

getsockopt

The getsockopt function (winsock.h) retrieves a socket option.

htonl

The htonl function (winsock.h) converts a u_long from host to TCP/IP network byte order (which is big-endian).

htons

The htons function (winsock.h) converts a u_short from host to TCP/IP network byte order (which is big-endian).

inet_addr

The inet_addr function (winsock.h) converts a string containing an IPv4 dotted-decimal address into a proper address for the IN_ADDR structure.

inet_ntoa

The inet_ntoa function (winsock.h) converts an (Ipv4) Internet network address into an ASCII string in Internet standard dotted-decimal format.

ioctlsocket

The ioctlsocket function (winsock.h) controls the I/O mode of a socket.

ntohl

The ntohl function (winsock.h) converts a u_long from TCP/IP network order to host byte order, which is little-endian on Intel processors.

ntohs

The ntohs function (winsock.h) converts a u_short from TCP/IP network byte order to host byte order, which is little-endian on Intel processors.

recv

The recv function (winsock.h) receives data from a connected socket or a bound connectionless socket.

recvfrom

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

sendto

The sendto function (winsock.h) sends data to a specific destination.

setsockopt

The setsockopt function (winsock.h) sets a socket option.

shutdown

The shutdown function (winsock.h) disables sends or receives on a socket.

TransmitFile

The TransmitFile function (winsock.h) transmits file data over a connected socket handle.

WSAAsyncGetHostByAddr

The WSAAsyncGetHostByAddr function (winsock.h) asynchronously retrieves host information that corresponds to an address.

WSAAsyncGetHostByName

The WSAAsyncGetHostByName function (winsock.h) asynchronously retrieves host information that corresponds to a host name.

WSAAsyncGetProtoByName

The WSAAsyncGetProtoByName function (winsock.h) asynchronously retrieves protocol information that corresponds to a protocol name.

WSAAsyncGetProtoByNumber

The WSAAsyncGetProtoByNumber function (winsock.h) asynchronously retrieves protocol information that corresponds to a protocol number.

WSAAsyncGetServByName

The WSAAsyncGetServByName function (winsock.h) asynchronously retrieves service information that corresponds to a service name and port.

WSAAsyncGetServByPort

The WSAAsyncGetServByPort function (winsock.h) asynchronously retrieves service information that corresponds to a port and protocol.

WSAAsyncSelect

The WSAAsyncSelect function (winsock.h) requests Windows message-based notification of network events for a socket.

WSACancelAsyncRequest

The WSACancelAsyncRequest function (winsock.h) cancels an incomplete asynchronous operation.

WSACleanup

The WSACleanup function (winsock.h) terminates use of the WS2_32.dll.

WSAGetLastError

The WSAGetLastError function (winsock.h) returns the error status for the last Windows Sockets operation that failed.

WSARecvEx

The WSARecvEx function (winsock.h) receives data from a connected socket or a bound connectionless socket.

WSASetLastError

The WSASetLastError function (winsock.h) sets the error code that can be retrieved through the WSAGetLastError function.

WSAStartup

The WSAStartup function (winsock.h) initiates use of the Winsock DLL by a process.

Structures

fd_set

The fd_set structure (winsock.h) is used by Windows Sockets (Winsock) functions and service providers to place sockets into a set.

HOSTENT

The HOSTENT (winsock.h) structure is used by functions to store information about a given host, such as host name, IPv4 address, and so forth.

LINGER

The LINGER (winsock.h) structure maintains information about a specific socket that specifies how that socket should behave when data is queued to be sent.

PROTOENT

The PROTOENT (winsock.h) structure contains the name and protocol numbers that correspond to a given protocol name.

SERVENT

The SERVENT (winsock.h) structure is used to store or return the name and service number for a given service name.

SOCKADDR

The sockaddr structure varies depending on the protocol selected. (sockaddr)

SOCKADDR_IN

The SOCKADDR_IN (winsock.h) structure varies depending on the protocol selected.

TIMEVAL

The TIMEVAL (winsock.h) structure is used to specify a time interval. It is associated with the Berkeley Software Distribution (BSD) Time.h header file.

TRANSMIT_FILE_BUFFERS

The TRANSMIT_FILE_BUFFERS (winsock.h) structure specifies data to be transmitted before and after file data during a TransmitFile function file transfer operation.

WSADATA

The WSADATA (winsock.h) structure contains information about the Windows Sockets implementation.

__WSAFDIsSet function (winsock.h)

10/22/2022 • 2 minutes to read • Edit Online

The **__WSAFDIsSet** function specifies whether a socket is included in a set of socket descriptors.

Syntax

```
int __WSAFDIsSet(
  SOCKET unnamedParam1,
  fd_set *unnamedParam2
);
```

Parameters

`unnamedParam1`

TBD

`unnamedParam2`

Pointer to an fd_set structure containing the set of socket descriptors. The **__WSAFDIsSet** function determines whether the socket specified in the *fd* parameter is a member of that set.

Return value

None

Remarks

Windows Phone 8: This function is supported for Windows Phone Store apps on Windows Phone 8 and later.

Windows 8.1 and **Windows Server 2012 R2**: This function is supported for Windows Store apps on Windows 8.1, Windows Server 2012 R2, and later.

Requirements

Minimum supported client	Windows 8.1, Windows Vista [desktop apps | UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps | UWP apps]
Target Platform	Windows
Header	winsock.h (include Winsock2.h)
Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

WSAAsyncSelect

WSAEventSelect

fd_set

select

AcceptEx function (winsock.h)

10/22/2022 • 10 minutes to read • Edit Online

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

> **Note** This function is a Microsoft-specific extension to the Windows Sockets specification.

Syntax

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

Parameters

`[in] sListenSocket`

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

`[in] sAcceptSocket`

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

`[in] lpOutputBuffer`

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

`[in] dwReceiveDataLength`

The number of bytes in *lpOutputBuffer* that will be used for actual receive data at the beginning of the buffer. This size should not include the size of the local address of the server, nor the remote address of the client; they are appended to the output buffer. If *dwReceiveDataLength* is zero, accepting the connection will not result in a receive operation. Instead, **AcceptEx** completes as soon as a connection arrives, without waiting for any data.

`[in] dwLocalAddressLength`

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

`[in] dwRemoteAddressLength`

The number of bytes reserved for the remote address information. This value must be at least 16 bytes more than the maximum address length for the transport protocol in use. Cannot be zero.

`[out] lpdwBytesReceived`

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

`[in] lpOverlapped`

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

Return value

If no error occurs, the **AcceptEx** function completed successfully and a value of **TRUE** is returned.

If the function fails, **AcceptEx** returns **FALSE**. The WSAGetLastError function can then be called to return extended error information. If **WSAGetLastError** returns **ERROR_IO_PENDING**, then the operation was successfully initiated and is still in progress. If the error is WSAECONNRESET, an incoming connection was indicated, but was subsequently terminated by the remote peer prior to accepting the call.

Remarks

The **AcceptEx** function combines several socket functions into a single API/kernel transition. The **AcceptEx** function, when successful, performs three tasks:

- A new connection is accepted.
- Both the local and remote addresses for the connection are returned.
- The first block of data sent by the remote is received.

> **Note** The function pointer for the **AcceptEx** function must be obtained at run time by making a call to the WSAIoctl function with the **SIO_GET_EXTENSION_FUNCTION_POINTER** opcode specified. The input buffer passed to the **WSAIoctl** function must contain **WSAID_ACCEPTEX**, a globally unique identifier (GUID) whose value identifies the **AcceptEx** extension function. On success, the output returned by the **WSAIoctl** function contains a pointer to the **AcceptEx** function. The **WSAID_ACCEPTEX** GUID is defined in the *Mswsock.h* header file.

A program can make a connection to a socket more quickly using **AcceptEx** instead of the accept function.

A single output buffer receives the data, the local socket address (the server), and the remote socket address (the client).

Using a single buffer improves performance. When using **AcceptEx**, the GetAcceptExSockaddrs function must be called to parse the buffer into its three distinct parts (data, local socket address, and remote socket address). On Windows XP and later, once the **AcceptEx** function completes and the **SO_UPDATE_ACCEPT_CONTEXT** option is set on the accepted socket, the local address associated with the accepted socket can also be retrieved using the getsockname function. Likewise, the remote address associated with the accepted socket can be retrieved using the getpeername function.

The buffer size for the local and remote address must be 16 bytes more than the size of the sockaddr structure for the transport protocol in use because the addresses are written in an internal format. For example, the size of a **sockaddr_in** (the address structure for TCP/IP) is 16 bytes. Therefore, a buffer size of at least 32 bytes must

be specified for the local and remote addresses.

The **AcceptEx** function uses overlapped I/O, unlike the accept function. If your application uses **AcceptEx**, it can service a large number of clients with a relatively small number of threads. As with all overlapped Windows functions, either Windows events or completion ports can be used as a completion notification mechanism.

Another key difference between the **AcceptEx** function and the accept function is that **AcceptEx** requires the caller to already have two sockets:

- One that specifies the socket on which to listen.
- One that specifies the socket on which to accept the connection.

The *sAcceptSocket* parameter must be an open socket that is neither bound nor connected.

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

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

ReadFile
WriteFile
send
WSASend
recv
WSARecv
TransmitFile
closesocket
setsockopt(only for SO_UPDATE_ACCEPT_CONTEXT)

> **Note** If the TransmitFile function is called with both the TF_DISCONNECT and TF_REUSE_SOCKET flags, the specified socket has been returned to a state in which it is neither bound nor connected. The socket handle can then be passed to the **AcceptEx** function in the *sAcceptSocket* parameter, but the socket cannot be passed to the ConnectEx function.

When the **AcceptEx** function returns, the socket *sAcceptSocket* is in the default state for a connected socket. The socket *sAcceptSocket* does not inherit the properties of the socket associated with *sListenSocket* parameter until SO_UPDATE_ACCEPT_CONTEXT is set on the socket. Use the setsockopt function to set the SO_UPDATE_ACCEPT_CONTEXT option, specifying *sAcceptSocket* as the socket handle and *sListenSocket* as the option value.

For example:

```
//Need to #include <mswsock.h> for SO_UPDATE_ACCEPT_CONTEXT

int iResult = 0;

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

If a receive buffer is provided, the overlapped operation will not complete until a connection is accepted and data is read. Use the getsockopt function with the SO_CONNECT_TIME option to check whether a connection has been accepted. If it has been accepted, you can determine how long the connection has been established. The return value is the number of seconds that the socket has been connected. If the socket is not connected, the **getsockopt** returns 0xFFFFFFFF. Applications that check whether the overlapped operation has completed, in

combination with the SO_CONNECT_TIME option, can determine that a connection has been accepted but no data has been received. Scrutinizing a connection in this manner enables an application to determine whether connections that have been established for a while have received no data. It is recommended such connections be terminated by closing the accepted socket, which forces the **AcceptEx** function call to complete with an error.

For example:

```
INT seconds;
INT bytes = sizeof(seconds);
int iResult = 0;

iResult = getsockopt( sAcceptSocket, SOL_SOCKET, SO_CONNECT_TIME,
                      (char *)&seconds, (PINT)&bytes );

if ( iResult != NO_ERROR ) {
    printf( "getsockopt(SO_CONNECT_TIME) failed: %u\n", WSAGetLastError( ) );
    exit(1);
}
```

Note All I/O initiated by a given thread is canceled when that thread exits. For overlapped sockets, pending asynchronous operations can fail if the thread is closed before the operations complete. See ExitThread for more information.

Windows Phone 8: This function is supported for Windows Phone Store apps on Windows Phone 8 and later.

Windows 8.1 and **Windows Server 2012 R2**: This function is supported for Windows Store apps on Windows 8.1, Windows Server 2012 R2, and later.

Example Code

The following example uses the **AcceptEx** function using overlapped I/O and completion ports.

```
#ifndef UNICODE
#define UNICODE
#endif

#define WIN32_LEAN_AND_MEAN

#include <winsock2.h>
#include <ws2tcpip.h>
#include <mswsock.h>
#include <stdio.h>

// Need to link with Ws2_32.lib
#pragma comment(lib, "Ws2_32.lib")

int main()
{
    //-----------------------------------------
    // Declare and initialize variables
    WSADATA wsaData;
    int iResult = 0;
    BOOL bRetVal = FALSE;

    HANDLE hCompPort;
    HANDLE hCompPort2;

    LPFN_ACCEPTEX lpfnAcceptEx = NULL;
    GUID GuidAcceptEx = WSAID_ACCEPTEX;
    WSAOVERLAPPED olOverlap;
```

```
    SOCKET ListenSocket = INVALID_SOCKET;
    SOCKET AcceptSocket = INVALID_SOCKET;
    sockaddr_in service;
    char lpOutputBuf[1024];
    int outBufLen = 1024;
    DWORD dwBytes;

    hostent *thisHost;
    char *ip;
    u_short port;

    // Initialize Winsock
    iResult = WSAStartup(MAKEWORD(2, 2), &wsaData);
    if (iResult != NO_ERROR) {
        wprintf(L"Error at WSAStartup\n");
        return 1;
    }

    // Create a handle for the completion port
    hCompPort = CreateIoCompletionPort(INVALID_HANDLE_VALUE, NULL, (u_long) 0, 0);
    if (hCompPort == NULL) {
        wprintf(L"CreateIoCompletionPort failed with error: %u\n",
            GetLastError() );
        WSACleanup();
        return 1;
    }

    // Create a listening socket
    ListenSocket = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
    if (ListenSocket == INVALID_SOCKET) {
        wprintf(L"Create of ListenSocket socket failed with error: %u\n",
            WSAGetLastError() );
        WSACleanup();
        return 1;
    }

    // Associate the listening socket with the completion port
    CreateIoCompletionPort((HANDLE) ListenSocket, hCompPort, (u_long) 0, 0);

    //-----------------------------------------
    // Bind the listening socket to the local IP address
    // and port 27015
    port = 27015;
    thisHost = gethostbyname("");
    ip = inet_ntoa(*(struct in_addr *) *thisHost->h_addr_list);

    service.sin_family = AF_INET;
    service.sin_addr.s_addr = inet_addr(ip);
    service.sin_port = htons(port);

    if (bind(ListenSocket, (SOCKADDR *) & service, sizeof (service)) == SOCKET_ERROR) {
        wprintf(L"bind failed with error: %u\n", WSAGetLastError());
        closesocket(ListenSocket);
        WSACleanup();
        return 1;
    }

    //-----------------------------------------
    // Start listening on the listening socket
    iResult = listen(ListenSocket, 100);
    if (iResult == SOCKET_ERROR) {
        wprintf(L"listen failed with error: %u\n", WSAGetLastError());
        closesocket(ListenSocket);
        WSACleanup();
        return 1;
    }

    wprintf(L"Listening on address: %s:%d\n", ip, port);
```

```
    // Load the AcceptEx function into memory using WSAIoctl.
    // The WSAIoctl function is an extension of the ioctlsocket()
    // function that can use overlapped I/O. The function's 3rd
    // through 6th parameters are input and output buffers where
    // we pass the pointer to our AcceptEx function. This is used
    // so that we can call the AcceptEx function directly, rather
    // than refer to the Mswsock.lib library.
    iResult = WSAIoctl(ListenSocket, SIO_GET_EXTENSION_FUNCTION_POINTER,
             &GuidAcceptEx, sizeof (GuidAcceptEx),
             &lpfnAcceptEx, sizeof (lpfnAcceptEx),
             &dwBytes, NULL, NULL);
    if (iResult == SOCKET_ERROR) {
        wprintf(L"WSAIoctl failed with error: %u\n", WSAGetLastError());
        closesocket(ListenSocket);
        WSACleanup();
        return 1;
    }

    // Create an accepting socket
    AcceptSocket = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
    if (AcceptSocket == INVALID_SOCKET) {
        wprintf(L"Create accept socket failed with error: %u\n", WSAGetLastError());
        closesocket(ListenSocket);
        WSACleanup();
        return 1;
    }

    // Empty our overlapped structure and accept connections.
    memset(&olOverlap, 0, sizeof (olOverlap));

    bRetVal = lpfnAcceptEx(ListenSocket, AcceptSocket, lpOutputBuf,
                 outBufLen - ((sizeof (sockaddr_in) + 16) * 2),
                 sizeof (sockaddr_in) + 16, sizeof (sockaddr_in) + 16,
                 &dwBytes, &olOverlap);
    if (bRetVal == FALSE) {
        wprintf(L"AcceptEx failed with error: %u\n", WSAGetLastError());
        closesocket(AcceptSocket);
        closesocket(ListenSocket);
        WSACleanup();
        return 1;
    }

    // Associate the accept socket with the completion port
    hCompPort2 = CreateIoCompletionPort((HANDLE) AcceptSocket, hCompPort, (u_long) 0, 0);
    // hCompPort2 should be hCompPort if this succeeds
    if (hCompPort2 == NULL) {
        wprintf(L"CreateIoCompletionPort associate failed with error: %u\n",
            GetLastError() );
        closesocket(AcceptSocket);
        closesocket(ListenSocket);
        WSACleanup();
        return 1;
    }

    // Continue on to use send, recv, TransmitFile(), etc.,.
    //...

    return 0;
}
```

Notes for QoS

The TransmitFile function allows the setting of two flags, TF_DISCONNECT or TF_REUSE_SOCKET, that return the socket to a "disconnected, reusable" state after the file has been transmitted. These flags should not be used on a socket where quality of service has been requested, since the service provider may immediately delete any quality

of service associated with the socket before the file transfer has completed. The best approach for a QoS-enabled socket is to simply call the closesocket function when the file transfer has completed, rather than relying on these flags.

Notes for ATM

There are important issues associated with connection setup when using Asynchronous Transfer Mode (ATM) with Windows Sockets 2. Please see the Remarks section in the accept function documentation for important ATM connection setup information.

Requirements

Minimum supported client	Windows 8.1, Windows Vista [desktop apps | UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps | UWP apps]
Target Platform	Windows
Header	winsock.h (include Mswsock.h)
Library	Mswsock.lib
DLL	Mswsock.dll

See also

GetAcceptExSockaddrs

GetOverlappedResult

GetQueuedCompletionStatus

OVERLAPPED

TransmitFile

Winsock Functions

Winsock Reference

accept

closesocket

getsockopt

listen

sockaddr

bind function (winsock.h)

10/22/2022 • 9 minutes to read • Edit Online

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

Syntax

```
int bind(
  [in] SOCKET         s,
       const sockaddr *addr,
  [in] int            namelen
);
```

Parameters

`[in] s`

A descriptor identifying an unbound socket.

`addr`

A pointer to a sockaddr structure of the local address to assign to the bound socket .

`[in] namelen`

The length, in bytes, of the value pointed to by the *name* parameter.

Return value

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

ERROR CODE	MEANING
WSANOTINITIALISED	**Note** A successful WSAStartup call must occur before using this function.
WSAENETDOWN	The network subsystem has failed.
WSAEACCES	An attempt was made to access a socket in a way forbidden by its access permissions. This error is returned if nn attempt to bind a datagram socket to the broadcast address failed because the setsockopt option SO_BROADCAST is not enabled.

WSAEADDRINUSE	Only one usage of each socket address (protocol/network address/port) is normally permitted. This error is returned if a process on the computer is already bound to the same fully qualified address and the socket has not been marked to allow address reuse with SO_REUSEADDR. For example, the IP address and port specified in the *name* parameter are already bound to another socket being used by another application. For more information, see the SO_REUSEADDR socket option in the SOL_SOCKET Socket Options reference, Using SO_REUSEADDR and SO_EXCLUSIVEADDRUSE, and SO_EXCLUSIVEADDRUSE.
WSAEADDRNOTAVAIL	The requested address is not valid in its context. This error is returned if the specified address pointed to by the *name* parameter is not a valid local IP address on this computer.
WSAEFAULT	The system detected an invalid pointer address in attempting to use a pointer argument in a call. This error is returned if the *name* parameter is NULL, the *name* or *namelen* parameter is not a valid part of the user address space, the *namelen* parameter is too small, the *name* parameter contains an incorrect address format for the associated address family, or the first two bytes of the memory block specified by *name* do not match the address family associated with the socket descriptor *s*.
WSAEINPROGRESS	A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function.
WSAEINVAL	An invalid argument was supplied. This error is returned of the socket *s* is already bound to an address.
WSAENOBUFS	Typically, **WSAENOBUFS** is an indication that there aren't enough ephemeral ports to allocate for the bind.
WSAENOTSOCK	An operation was attempted on something that is not a socket. This error is returned if the descriptor in the *s* parameter is not a socket.

Remarks

The **bind** function is required on an unconnected socket before subsequent calls to the listen function. It is normally used to bind to either connection-oriented (stream) or connectionless (datagram) sockets. The **bind** function may also be used to bind to a raw socket (the socket was created by calling the socket function with the *type* parameter set to SOCK_RAW). The **bind** function may also be used on an unconnected socket before subsequent calls to the connect, ConnectEx, WSAConnect, WSAConnectByList, or WSAConnectByName functions before send operations.

When a socket is created with a call to the socket function, it exists in a namespace (address family), but it has no

name assigned to it. Use the **bind** function to establish the local association of the socket by assigning a local name to an unnamed socket.

A name consists of three parts when using the Internet address family:

- The address family.
- A host address.
- A port number that identifies the application.

In Windows Sockets 2, the *name* parameter is not strictly interpreted as a pointer to a sockaddr structure. It is cast this way for Windows Sockets 1.1 compatibility. Service providers are free to regard it as a pointer to a block of memory of size *namelen*. The first 2 bytes in this block (corresponding to the **sa_family** member of the **sockaddr** structure, the **sin_family** member of the **sockaddr_in** structure, or the **sin6_family** member of the **sockaddr_in6** structure) must contain the address family that was used to create the socket. Otherwise, an error WSAEFAULT occurs.

If an application does not care what local address is assigned, specify the constant value **INADDR_ANY** for an IPv4 local address or the constant value **in6addr_any** for an IPv6 local address in the **sa_data** member of the *name* parameter. This allows the underlying service provider to use any appropriate network address, potentially simplifying application programming in the presence of *multihomed* hosts (that is, hosts that have more than one network interface and address).

For TCP/IP, if the port is specified as zero, the service provider assigns a unique port to the application from the dynamic client port range. On Windows Vista and later, the dynamic client port range is a value between 49152 and 65535. This is a change from Windows Server 2003 and earlier where the dynamic client port range was a value between 1025 and 5000. The maximum value for the client dynamic port range can be changed by setting a value under the following registry key:

HKLM\SYSTEM\CurrentControlSet\Services\Tcpip\Parameters

The **MaxUserPort** registry value sets the value to use for the maximum value of the dynamic client port range. You must restart the computer for this setting to take effect.

On Windows Vista and later, the dynamic client port range can be viewed and changed using **netsh** commands. The dynamic client port range can be set differently for UDP and TCP and also for IPv4 and IPv6. For more information, see KB 929851.

The application can use getsockname after calling **bind** to learn the address and the port that has been assigned to the socket. If the Internet address is equal to **INADDR_ANY** or **in6addr_any**, **getsockname** cannot necessarily supply the address until the socket is connected, since several addresses can be valid if the host is multihomed. Binding to a specific port number other than port 0 is discouraged for client applications, since there is a danger of conflicting with another socket already using that port number on the local computer.

> **Note** When using **bind** with the SO_EXCLUSIVEADDRUSE or SO_REUSEADDR socket option, the socket option must be set prior to executing **bind** to have any affect. For more information, see SO_EXCLUSIVEADDRUSE and Using SO_REUSEADDR and SO_EXCLUSIVEADDRUSE.

For multicast operations, the preferred method is to call the **bind** function to associate a socket with a local IP address and then join the multicast group. Although this order of operations is not mandatory, it is strongly recommended. So a multicast application would first select an IPv4 or IPv6 address on the local computer, the wildcard IPv4 address (**INADDR_ANY**), or the wildcard IPv6 address (**in6addr_any**). The the multicast application would then call the **bind** function with this address in the in the **sa_data** member of the *name* parameter to associate the local IP address with the socket. If a wildcard address was specified, then Windows will select the local IP address to use. After the **bind** function completes, an application would then join the

multicast group of interest. For more information on how to join a multicast group, see the section on Multicast Programming. This socket can then be used to receive multicast packets from the multicast group using the recv, recvfrom, WSARecv, WSARecvEx, WSARecvFrom, or LPFN_WSARECVMSG (WSARecvMsg) functions.

The **bind** function is not normally required for send operations to a multicast group. The sendto,WSASendMsg, and WSASendTo functions implicitly bind the socket to the wildcard address if the socket is not already bound. The **bind** function is required before the use of the send or WSASend functions which do not perform an implicit bind and are allowed only on connected sockets, which means the socket must have already been bound for it to be connected. The **bind** function might be used before send operations using the **sendto**,**WSASendMsg**, or **WSASendTo** functions if an application wanted to select a specific local IP address on a local computer with multiple network interfaces and local IP addresses. Otherwise an implicit bind to the wildcard address using the **sendto**,**WSASendMsg** , or **WSASendTo** functions might result in a different local IP address being used for send operations.

> **Note** When issuing a blocking Winsock call such as **bind**, Winsock may need to wait for a network event before the call can complete. Winsock performs an alertable wait in this situation, which can be interrupted by an asynchronous procedure call (APC) scheduled on the same thread. Issuing another blocking Winsock call inside an APC that interrupted an ongoing blocking Winsock call on the same thread will lead to undefined behavior, and must never be attempted by Winsock clients.

Notes for IrDA Sockets

- The Af_irda.h header file must be explicitly included.
- Local names are not exposed in IrDA. IrDA client sockets therefore, must never call the **bind** function before the connect function. If the IrDA socket was previously bound to a service name using **bind**, the **connect** function will fail with SOCKET_ERROR.
- If the service name is of the form "LSAP-SELxxx," where xxx is a decimal integer in the range 1-127, the address indicates a specific LSAP-SEL xxx rather than a service name. Service names such as these allow server applications to accept incoming connections directed to a specific LSAP-SEL, without first performing an ISA service name query to get the associated LSAP-SEL. One example of this service name type is a non-Windows device that does not support IAS.

Windows Phone 8: This function is supported for Windows Phone Store apps on Windows Phone 8 and later.

Windows 8.1 and **Windows Server 2012 R2**: This function is supported for Windows Store apps on Windows 8.1, Windows Server 2012 R2, and later.

Examples

The following example demonstrates the use of the **bind** function. For another example that uses the **bind** function, see Getting Started With Winsock.

```
#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 main()
{

    // Declare some variables
    WSADATA wsaData;

    int iResult = 0;            // used to return function results

    // the listening socket to be created
    SOCKET ListenSocket = INVALID_SOCKET;

    // The socket address to be passed to bind
    sockaddr_in service;

    //----------------------
    // Initialize Winsock
    iResult = WSAStartup(MAKEWORD(2, 2), &wsaData);
    if (iResult != NO_ERROR) {
        wprintf(L"Error at WSAStartup()\n");
        return 1;
    }
    //----------------------
    // Create a SOCKET for listening for
    // incoming connection requests
    ListenSocket = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
    if (ListenSocket == INVALID_SOCKET) {
        wprintf(L"socket function failed with error: %u\n", WSAGetLastError());
        WSACleanup();
        return 1;
    }
    //----------------------
    // The sockaddr_in structure specifies the address family,
    // IP address, and port for the socket that is being bound.
    service.sin_family = AF_INET;
    service.sin_addr.s_addr = inet_addr("127.0.0.1");
    service.sin_port = htons(27015);

    //----------------------
    // Bind the socket.
    iResult = bind(ListenSocket, (SOCKADDR *) &service, sizeof (service));
    if (iResult == SOCKET_ERROR) {
        wprintf(L"bind failed with error %u\n", WSAGetLastError());
        closesocket(ListenSocket);
        WSACleanup();
        return 1;
    }
    else
        wprintf(L"bind returned success\n");

    WSACleanup();
    return 0;
}
```

Requirements

Minimum supported client	Windows 8.1, Windows Vista [desktop apps | UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps | UWP apps]
Target Platform	Windows
Header	winsock.h (include Winsock2.h)
Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

Multicast Programming

SOL_SOCKET Socket Options

SO_EXCLUSIVEADDRUSE

TCP/IP Raw Sockets

Using SO_REUSEADDR and SO_EXCLUSIVEADDRUSE

WSACancelBlockingCall

Winsock Functions

Winsock Reference

connect

getsockname

listen

setsockopt

sockaddr

socket

closesocket function (winsock.h)

10/22/2022 • 9 minutes to read • Edit Online

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

Syntax

```
int closesocket(
  [in] SOCKET s
);
```

Parameters

`[in] s`

A descriptor identifying the socket to close.

Return value

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

ERROR CODE	MEANING
WSANOTINITIALISED	A successful WSAStartup call must occur before using this function.
WSAENETDOWN	The network subsystem has failed.
WSAENOTSOCK	The descriptor is not a socket.
WSAEINPROGRESS	A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function.
WSAEINTR	The (blocking) Windows Socket 1.1 call was canceled through WSACancelBlockingCall.
WSAEWOULDBLOCK	The socket is marked as nonblocking, but the **l_onoff** member of the linger structure is set to nonzero and the **l_linger** member of the **linger** structure is set to a nonzero timeout value.

Remarks

The **closesocket** function closes a socket. Use it to release the socket descriptor passed in the *s* parameter. Note that the socket descriptor passed in the *s* parameter may immediately be reused by the system as soon as **closesocket** function is issued. As a result, it is not reliable to expect further references to the socket descriptor

passed in the *s* parameter to fail with the error WSAENOTSOCK. A Winsock client must never issue **closesocket** on *s* concurrently with another Winsock function call.

Any pending overlapped send and receive operations (WSASend/ WSASendTo/ WSARecv/ WSARecvFrom with an overlapped socket) issued by any thread in this process are also canceled. Any event, completion routine, or completion port action specified for these overlapped operations is performed. The pending overlapped operations fail with the error status WSA_OPERATION_ABORTED.

An application should not assume that any outstanding I/O operations on a socket will all be guaranteed to completed when **closesocket** returns. The **closesocket** function will initiate cancellation on the outstanding I/O operations, but that does not mean that an application will receive I/O completion for these I/O operations by the time the **closesocket** function returns. Thus, an application should not cleanup any resources (WSAOVERLAPPED structures, for example) referenced by the outstanding I/O requests until the I/O requests are indeed completed.

An application should always have a matching call to **closesocket** for each successful call to socket to return any socket resources to the system.

The linger structure maintains information about a specific socket that specifies how that socket should behave when data is queued to be sent and the **closesocket** function is called on the socket.

The **l_onoff** member of the **linger** structure determines whether a socket should remain open for a specified amount of time after a **closesocket** function call to enable queued data to be sent. This member can be modified in two ways:

- Call the setsockopt function with the *optname* parameter set to **SO_DONTLINGER**. The *optval* parameter determines how the **l_onoff** member is modified.
- Call the setsockopt function with the *optname* parameter set to **SO_LINGER**. The *optval* parameter specifies how both the **l_onoff** and **l_linger** members are modified.

The **l_linger** member of the **linger** structure determines the amount of time, in seconds, a socket should remain open. This member is only applicable if the **l_onoff** member of the **linger** structure is nonzero.

The default parameters for a socket are the **l_onoff** member of the **linger** structure is zero, indicating that the socket should not remain open. The default value for the **l_linger** member of the **linger** structure is zero, but this value is ignored when the **l_onoff** member is set to zero.

To enable a socket to remain open, an application should set the **l_onoff** member to a nonzero value and set the **l_linger** member to the desired timeout in seconds. To disable a socket from remaining open, an application only needs to set the **l_onoff** member of the **linger** structure to zero.

If an application calls the setsockopt function with the *optname* parameter set to **SO_DONTLINGER** to set the **l_onoff** member to a nonzero value, the value for the **l_linger** member is not specified. In this case, the timeout used is implementation dependent. If a previous timeout has been established for a socket (by previously calling the **setsockopt** function with the *optname* parameter set to **SO_LINGER**), this timeout value should be reinstated by the service provider.

The semantics of the **closesocket** function are affected by the socket options that set members of **linger** structure.

L_ONOFF	L_LINGER	TYPE OF CLOSE	WAIT FOR CLOSE?
zero	Do not care	Graceful close	No
nonzero	zero	Hard	No

nonzero	nonzero	Graceful if all data is sent within timeout value specified in the **l_linger** member. Hard if all data could not be sent within timeout value specified in the **l_linger** member.	Yes

If the **l_onoff** member of the LINGER structure is zero on a stream socket, the **closesocket** call will return immediately and does not receive WSAEWOULDBLOCK whether the socket is blocking or nonblocking. However, any data queued for transmission will be sent, if possible, before the underlying socket is closed. This is also called a graceful disconnect or close. In this case, the Windows Sockets provider cannot release the socket and other resources for an arbitrary period, thus affecting applications that expect to use all available sockets. This is the default behavior for a socket.

If the **l_onoff** member of the linger structure is nonzero and **l_linger** member is zero, **closesocket** is not blocked even if queued data has not yet been sent or acknowledged. This is called a hard or abortive close, because the socket's virtual circuit is reset immediately, and any unsent data is lost. On Windows, any **recv** call on the remote side of the circuit will fail with WSAECONNRESET.

If the **l_onoff** member of the linger structure is set to nonzero and **l_linger** member is set to a nonzero timeout on a blocking socket, the **closesocket** call blocks until the remaining data has been sent or until the timeout expires. This is called a graceful disconnect or close if all of the data is sent within timeout value specified in the **l_linger** member. If the timeout expires before all data has been sent, the Windows Sockets implementation terminates the connection before **closesocket** returns and this is called a hard or abortive close.

Setting the **l_onoff** member of the linger structure to nonzero and the **l_linger** member with a nonzero timeout interval on a nonblocking socket is not recommended. In this case, the call to **closesocket** will fail with an error of WSAEWOULDBLOCK if the close operation cannot be completed immediately. If **closesocket** fails with WSAEWOULDBLOCK the socket handle is still valid, and a disconnect is not initiated. The application must call **closesocket** again to close the socket.

If the **l_onoff** member of the linger structure is nonzero and the **l_linger** member is a nonzero timeout interval on a blocking socket, the result of the **closesocket** function can't be used to determine whether all data has been sent to the peer. If the data is sent before the timeout specified in the **l_linger** member expires or if the connection was aborted, the **closesocket** function won't return an error code (the return value from the **closesocket** function is zero).

The **closesocket** call will only block until all data has been delivered to the peer or the timeout expires. If the connection is reset because the timeout expires, then the socket will not go into TIME_WAIT state. If all data is sent within the timeout period, then the socket can go into TIME_WAIT state.

If the **l_onoff** member of the linger structure is nonzero and the **l_linger** member is a zero timeout interval on a blocking socket, then a call to **closesocket** will reset the connection. The socket will not go to the TIME_WAIT state.

The getsockopt function can be called with the *optname* parameter set to **SO_LINGER** to retrieve the current value of the **linger** structure associated with a socket.

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

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

- If the **l_onoff** member of the LINGER structure is zero (the default for a socket), **closesocket** returns immediately and the connection is gracefully closed in the background.
- If the **l_onoff** member of the linger structure is set to nonzero and the **l_linger** member is set to zero (no timeout) **closesocket** returns immediately and the connection is reset or terminated.
- If the **l_onoff** member of the linger structure is set to nonzero and the **l_linger** member is set to a nonzero timeout:– For a blocking socket, **closesocket** blocks until all data is sent or the timeout expires.

  – For a nonblocking socket, **closesocket** returns immediately indicating failure.

For additional information please see Graceful Shutdown, Linger Options, and Socket Closure for more information.

> **Note** When issuing a blocking Winsock call such as **closesocket**, Winsock may need to wait for a network event before the call can complete. Winsock performs an alertable wait in this situation, which can be interrupted by an asynchronous procedure call (APC) scheduled on the same thread. Issuing another blocking Winsock call inside an APC that interrupted an ongoing blocking Winsock call on the same thread will lead to undefined behavior, and must never be attempted by Winsock clients.

Notes for IrDA Sockets

Keep the following in mind:

- The Af_irda.h header file must be explicitly included.
- The standard linger options are supported.
- Although IrDA does not provide a graceful close, IrDA will defer closing until receive queues are purged. Thus, an application can send data and immediately call the socket function, and be confident that the receiver will copy the data before receiving an FD_CLOSE message.

Notes for ATM

The following are important issues associated with connection teardown when using Asynchronous Transfer Mode (ATM) and Windows Sockets 2:

- Using the **closesocket** or shutdown functions with SD_SEND or SD_BOTH results in a RELEASE signal being sent out on the control channel. Due to ATM's use of separate signal and data channels, it is possible that a RELEASE signal could reach the remote end before the last of the data reaches its destination, resulting in a loss of that data. One possible solutions is programming a sufficient delay between the last data sent and the **closesocket** or shutdown function calls for an ATM socket.
- Half close is not supported by ATM.
- Both abortive and graceful disconnects result in a RELEASE signal being sent out with the same cause field. In either case, received data at the remote end of the socket is still delivered to the application. See Graceful Shutdown, Linger Options, and Socket Closure for more information.

Windows Phone 8: This function is supported for Windows Phone Store apps on Windows Phone 8 and later.

Windows 8.1 and **Windows Server 2012 R2**: This function is supported for Windows Store apps on Windows 8.1, Windows Server 2012 R2, and later.

Requirements

Minimum supported client	Windows 8.1, Windows Vista [desktop apps | UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps | UWP apps]
Target Platform	Windows
Header	winsock.h (include Winsock2.h)
Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

Graceful Shutdown, Linger Options, and Socket Closure

WSAAsyncSelect

WSADuplicateSocket

WSAOVERLAPPED

Winsock Functions

Winsock Reference

accept

getsockopt

ioctlsocket

linger

setsockopt

socket

fd_set structure (winsock.h)

10/22/2022 • 2 minutes to read • Edit Online

The **fd_set** structure is used by various Windows Sockets functions and service providers, such as the select function, to place sockets into a "set" for various purposes, such as testing a given socket for readability using the *readfds* parameter of the **select** function.

Syntax

```
typedef struct fd_set {
  u_int  fd_count;
  SOCKET fd_array[FD_SETSIZE];
} fd_set, FD_SET, *PFD_SET, *LPFD_SET;
```

Members

`fd_count`

The number of sockets in the set.

`fd_array[FD_SETSIZE]`

An array of sockets that are in the set.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	winsock.h (include Winsock2.h, Winsock.h)

See also

WSAAsyncSelect

WSAEventSelect

select

FD_SET macro (winsock.h)

10/22/2022 • 2 minutes to read • Edit Online

The **fd_set** structure is used by various Windows Sockets functions and service providers, such as the select function, to place sockets into a "set" for various purposes, such as testing a given socket for readability using the *readfds* parameter of the **select** function.

Syntax

```
void FD_SET(
   fd,
   set
);
```

Parameters

`fd`

`set`

Return value

None

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winsock.h (include Winsock2.h, Winsock.h)

See also

WSAAsyncSelect

WSAEventSelect

select

GetAcceptExSockaddrs function (winsock.h)

10/22/2022 • 2 minutes to read • Edit Online

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

> **Note** This function is a Microsoft-specific extension to the Windows Sockets specification.

Syntax

```
void GetAcceptExSockaddrs(
  [in]  PVOID    lpOutputBuffer,
  [in]  DWORD    dwReceiveDataLength,
  [in]  DWORD    dwLocalAddressLength,
  [in]  DWORD    dwRemoteAddressLength,
  [out] sockaddr **LocalSockaddr,
  [out] LPINT    LocalSockaddrLength,
  [out] sockaddr **RemoteSockaddr,
  [out] LPINT    RemoteSockaddrLength
);
```

Parameters

`[in] lpOutputBuffer`

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

`[in] dwReceiveDataLength`

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

`[in] dwLocalAddressLength`

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

`[in] dwRemoteAddressLength`

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

`[out] LocalSockaddr`

A pointer to the sockaddr structure that receives the local address of the connection (the same information that would be returned by the getsockname function). This parameter must be specified.

`[out] LocalSockaddrLength`

The size, in bytes, of the local address. This parameter must be specified.

`[out] RemoteSockaddr`

A pointer to the sockaddr structure that receives the remote address of the connection (the same information that would be returned by the getpeername function). This parameter must be specified.

`[out] RemoteSockaddrLength`

The size, in bytes, of the local address. This parameter must be specified.

Return value

None

Remarks

The **GetAcceptExSockaddrs** function is used exclusively with the AcceptEx function to parse the first data that the socket receives into local and remote addresses. The **AcceptEx** function returns local and remote address information in an internal format. Application developers need to use the **GetAcceptExSockaddrs** function if there is a need for the sockaddr structures containing the local or remote addresses.

> **Note** The function pointer for the **GetAcceptExSockaddrs** function must be obtained at run time by making a call to the WSAIoctl function with the **SIO_GET_EXTENSION_FUNCTION_POINTER** opcode specified. The input buffer passed to the **WSAIoctl** function must contain **WSAID_GETACCEPTEXSOCKADDRS**, a globally unique identifier (GUID) whose value identifies the **GetAcceptExSockaddrs** extension function. On success, the output returned by the **WSAIoctl** function contains a pointer to the **GetAcceptExSockaddrs** function. The **WSAID_GETACCEPTEXSOCKADDRS** GUID is defined in the *Mswsock.h* header file.

Windows Phone 8: This function is supported for Windows Phone Store apps on Windows Phone 8 and later.

Windows 8.1 and **Windows Server 2012 R2**: This function is supported for Windows Store apps on Windows 8.1, Windows Server 2012 R2, and later.

Requirements

Minimum supported client	Windows 8.1, Windows Vista [desktop apps | UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps | UWP apps]
Target Platform	Windows
Header	winsock.h (include Mswsock.h)
Library	Mswsock.lib
DLL	Mswsock.dll

See also

AcceptEx

Winsock Functions

Winsock Reference

getpeername

getsockname

sockaddr

gethostbyaddr function (winsock.h)

10/22/2022 • 4 minutes to read • Edit Online

[**gethostbyaddr** is no longer recommended for use as of Windows Sockets 2. Instead, use getnameinfo.]

The **gethostbyaddr** function retrieves the host information corresponding to a network address.

Syntax

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

Parameters

`addr`

TBD

`len`

TBD

`type`

TBD

Return value

If no error occurs, **gethostbyaddr** returns a pointer to the hostent structure. Otherwise, it returns a null pointer, and a specific error code can be retrieved by calling WSAGetLastError.

ERROR CODE	MEANING
WSANOTINITIALISED	A successful WSAStartup call must occur before using this function.
WSAEINVAL	An invalid argument was supplied. This error is returned if **AF_INET6** was specified in the *type* parameter and the *len* parameter was not set equal to the size of an IPv6 address.
WSAENETDOWN	The network subsystem has failed.
WSAHOST_NOT_FOUND	Authoritative answer host not found.
WSATRY_AGAIN	Nonauthoritative host not found, or server failed.

WSANO_RECOVERY	A nonrecoverable error occurred.
WSANO_DATA	Valid name, no data record of requested type.
WSAEINPROGRESS	A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function.
WSAEAFNOSUPPORT	The *type* specified is not supported by the Windows Sockets implementation.
WSAEFAULT	The *addr* parameter is not a valid part of the user address space, or the *len* parameter is too small.
WSAEINTR	A blocking Windows Socket 1.1 call was canceled through WSACancelBlockingCall.

Remarks

The **gethostbyaddr** function returns a pointer to the hostent structure that contains the name and address corresponding to the given network address.

The memory for the hostent structure returned by the **gethostbyaddr** function is allocated internally by the Winsock DLL from thread local storage. Only a single **hostent** structure is allocated and used, no matter how many times the **gethostbyaddr** or gethostbyname functions are called on the thread. The returned **hostent** structure must be copied to an application buffer if additional calls are to be made to the **gethostbyaddr** or **gethostbyname** functions on the same thread. Otherwise, the return value will be overwritten by subsequent **gethostbyaddr** or **gethostbyname** calls on the same thread. The internal memory allocated for the returned **hostent** structure is released by the Winsock DLL when the thread exits.

An application should not try to release the memory used by the returned hostent structure. The application must never attempt to modify this structure or to free any of its components. Furthermore, only one copy of this structure is allocated per thread, so the application should copy any information it needs before issuing any other function calls to **gethostbyaddr** or gethostbyname.

Although **gethostbyaddr** no longer recommended for use as of Windows Sockets 2 and the getnameinfo function should be used, **gethostbyaddr** is capable of returning a NetBIOS name; **getnameinfo** is not. Developers requiring NetBIOS name resolution may need to use **gethostbyaddr** until their applications are completely independent of NetBIOS names.

> **Note** The capability to perform reverse lookups using the **gethostbyaddr** function is convenient, but such lookups are considered inherently unreliable, and should be used only as a hint.

Example Code

The following example demonstrates the use of the **gethostbyaddr** function.

```
#include <winsock2.h>
#include <ws2tcpip.h>
#include <stdio.h>
```

```
int main(int argc, char **argv)
{

    //-----------------------------------------
    // Declare and initialize variables
    WSADATA wsaData;
    int iResult;

    DWORD dwError;
    int i = 0;
    int bIpv6 = 0;

    struct hostent *remoteHost;
    char *host_addr;
    struct in_addr addr = { 0 };
    IN6_ADDR addr6;

    char **pAlias;

    // Validate the parameters
    if (argc < 2) {
        printf("usage: %s 4 ipv4address\n", argv[0]);
        printf(" or\n");
        printf("usage: %s 6 ipv6address\n", argv[0]);
        printf("  to return the hostname\n");
        printf("       %s 4 127.0.0.1\n", argv[0]);
        printf("       %s 6 0::1\n", argv[0]);
        return 1;
    }
    // Validate parameters
    if (atoi(argv[1]) == 4)
        bIpv6 = 0;
    else if (atoi(argv[1]) == 6)
        bIpv6 = 1;
    else {
        printf("usage: %s 4 ipv4address\n", argv[0]);
        printf(" or\n");
        printf("usage: %s 6 ipv6address\n", argv[0]);
        printf("  to return the hostname\n");
        printf("       %s 4 127.0.0.1\n", argv[0]);
        printf("       %s 6 0::1\n", argv[0]);
        return 1;
    }

    // Initialize Winsock
    iResult = WSAStartup(MAKEWORD(2, 2), &wsaData);
    if (iResult != 0) {
        printf("WSAStartup failed: %d\n", iResult);
        return 1;
    }

    host_addr = argv[2];

    printf("Calling gethostbyaddr with %s\n", host_addr);
    if (bIpv6 == 1) {
        {
            iResult = inet_pton(AF_INET6, host_addr, &addr6);
            if (iResult == 0) {
                printf("The IPv6 address entered must be a legal address\n");
                return 1;
            } else
                remoteHost = gethostbyaddr((char *) &addr6, 16, AF_INET6);
        }
    } else {
        addr.s_addr = inet_addr(host_addr);
        if (addr.s_addr == INADDR_NONE) {
            printf("The IPv4 address entered must be a legal address\n");
            return 1;
        } else
```

```
            remoteHost = gethostbyaddr((char *) &addr, 4, AF_INET);
    }

    if (remoteHost == NULL) {
        dwError = WSAGetLastError();
        if (dwError != 0) {
            if (dwError == WSAHOST_NOT_FOUND) {
                printf("Host not found\n");
                return 1;
            } else if (dwError == WSANO_DATA) {
                printf("No data record found\n");
                return 1;
            } else {
                printf("Function failed with error: %ld\n", dwError);
                return 1;
            }
        }
    } else {
        printf("Function returned:\n");
        printf("\tOfficial name: %s\n", remoteHost->h_name);
        for (pAlias = remoteHost->h_aliases; *pAlias != 0; pAlias++) {
            printf("\tAlternate name #%d: %s\n", ++i, *pAlias);
        }
        printf("\tAddress type: ");
        switch (remoteHost->h_addrtype) {
        case AF_INET:
            printf("AF_INET\n");
            break;
        case AF_INET6:
            printf("AF_INET6\n");
            break;
        case AF_NETBIOS:
            printf("AF_NETBIOS\n");
            break;
        default:
            printf(" %d\n", remoteHost->h_addrtype);
            break;
        }
        printf("\tAddress length: %d\n", remoteHost->h_length);

        if (remoteHost->h_addrtype == AF_INET) {
            while (remoteHost->h_addr_list[i] != 0) {
                addr.s_addr = *(u_long *) remoteHost->h_addr_list[i++];
                printf("\tIPv4 Address #%d: %s\n", i, inet_ntoa(addr));
            }
        } else if (remoteHost->h_addrtype == AF_INET6)
            printf("\tRemotehost is an IPv6 address\n");
    }

    return 0;
}
```

Windows Phone 8: This function is supported for Windows Phone Store apps on Windows Phone 8 and later.

Windows 8.1 and **Windows Server 2012 R2**: This function is supported for Windows Store apps on Windows 8.1, Windows Server 2012 R2, and later.

Requirements

Minimum supported client	Windows 8.1, Windows Vista [desktop apps | UWP apps]

Minimum supported server	Windows Server 2003 [desktop apps | UWP apps]
Target Platform	Windows
Header	winsock.h (include Winsock2.h, Winsock.h)
Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

GetAddrInfoEx

GetAddrInfoW

WSAAsyncGetHostByAddr

Winsock Functions

Winsock Reference

addrinfo

addrinfoW

getaddrinfo

gethostbyname

hostent

gethostbyname function (winsock.h)

10/22/2022 • 5 minutes to read • Edit Online

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

> **Note** The **gethostbyname** function has been deprecated by the introduction of the getaddrinfo function. Developers creating Windows Sockets 2 applications are urged to use the **getaddrinfo** function instead of **gethostbyname**.

Syntax

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

Parameters

`name`

TBD

Return value

If no error occurs, **gethostbyname** returns a pointer to the hostent structure described above. Otherwise, it returns a **null** pointer and a specific error number can be retrieved by calling WSAGetLastError.

ERROR CODE	MEANING
WSANOTINITIALISED	A successful WSAStartup call must occur before using this function.
WSAENETDOWN	The network subsystem has failed.
WSAHOST_NOT_FOUND	Authoritative answer host not found.
WSATRY_AGAIN	Nonauthoritative host not found, or server failure.
WSANO_RECOVERY	A nonrecoverable error occurred.

WSANO_DATA	The requested name is valid, but no data of the requested type was found. This error is also returned if the *name* parameter contains a string representation of an IPv6 address or an illegal IPv4 address. This error should not be interpreted to mean that the *name* parameter contains a name string that has been validated for a particular protocol (an IP hostname, for example). Since Winsock supports multiple name service providers, a name may potentially be valid for one provider and not accepted by another provider.
WSAEINPROGRESS	A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function.
WSAEFAULT	The *name* parameter is not a valid part of the user address space.
WSAEINTR	A blocking Windows Socket 1.1 call was canceled through WSACancelBlockingCall.

Remarks

The **gethostbyname** function returns a pointer to a hostent structure—a structure allocated by Windows Sockets. The **hostent** structure contains the results of a successful search for the host specified in the *name* parameter.

If the host specified in the *name* parameter has both IPv4 and IPv6 addresses, only the IPv4 addresses will be returned. The **gethostbyname** function can only return IPv4 addresses for the *name* parameter. The getaddrinfo function and associated addrinfo structure should be used if IPv6 addresses for the host are required or if both IPv4 and IPv6 addresses for the host are required.

If the *name* parameter points to an empty string or *name* is **NULL**, the returned string is the same as the string returned by a successful gethostname function call (the standard host name for the local computer).

If the *name* parameter contains a string representation of a legal IPv4 address, then the binary IPv4 address that represents the string is returned in the hostent structure. The **h_name** member of the **hostent** structure contains the string representation of the IPv4 address and the **h_addr_list** contains the binary IPv4 address. If the *name* parameter contains a string representation of an IPv6 address or an illegal IPv4 address, then the **gethostbyname** function will fail and return WSANO_DATA.

The memory for the hostent structure returned by the **gethostbyname** function is allocated internally by the Winsock DLL from thread local storage. Only a single **hostent** structure is allocated and used, no matter how many times the gethostbyaddr or **gethostbyname** functions are called on the thread. The returned **hostent** structure must be copied to an application buffer if additional calls are to be made to the **gethostbyname** or **gethostbyaddr** functions on the same thread. Otherwise, the return value will be overwritten by subsequent **gethostbyname** or **gethostbyaddr** calls on the same thread. The internal memory allocated for the returned **hostent** structure is released by the Winsock DLL when the thread exits.

An application should not try to release the memory used by the returned hostent structure. The application must never attempt to modify this structure or to free any of its components. Furthermore, only one copy of this structure is allocated per thread, so the application should copy any information it needs before issuing any other function calls to **gethostbyname** or gethostbyaddr .

The **gethostbyname** function cannot take an IP address string as a parameter passed to it in the *name* and

resolve it to a host name. Such a request is treated exactly as a string representation of an IPv4 address or an unknown host name were passed. An application can use the inet_addr to convert an IPv4 address string to a binary IPv4 address, then use another function, gethostbyaddr, to resolve the IPv4 address to a host name.

> **Note** The **gethostbyname** function does not check the size of the *name* parameter before passing the buffer. With an improperly sized *name* parameter, heap corruption can occur.

Example Code

The following examples demonstrates the use of the **gethostbyname** function.

```
#include <winsock2.h>
#include <ws2tcpip.h>
#include <stdio.h>
#include <windows.h>
#pragma comment(lib, "ws2_32.lib")

int main(int argc, char **argv)
{

    //-----------------------------------------
    // Declare and initialize variables
    WSADATA wsaData;
    int iResult;

    DWORD dwError;
    int i = 0;

    struct hostent *remoteHost;
    char *host_name;
    struct in_addr addr;

    char **pAlias;

    // Validate the parameters
    if (argc != 2) {
        printf("usage: %s hostname\n", argv[0]);
        printf("  to return the IP addresses for the host\n");
        printf("       %s www.contoso.com\n", argv[0]);
        printf(" or\n");
        printf("       %s IPv4string\n", argv[0]);
        printf("  to return an IPv4 binary address for an IPv4string\n");
        printf("       %s 127.0.0.1\n", argv[0]);
        return 1;
    }
    // Initialize Winsock
    iResult = WSAStartup(MAKEWORD(2, 2), &wsaData);
    if (iResult != 0) {
        printf("WSAStartup failed: %d\n", iResult);
        return 1;
    }

    host_name = argv[1];

    printf("Calling gethostbyname with %s\n", host_name);
    remoteHost = gethostbyname(host_name);

    if (remoteHost == NULL) {
        dwError = WSAGetLastError();
        if (dwError != 0) {
            if (dwError == WSAHOST_NOT_FOUND) {
                printf("Host not found\n");
                return 1;
            } else if (dwError == WSANO_DATA) {
```

```
                printf("No data record found\n");
                return 1;
            } else {
                printf("Function failed with error: %ld\n", dwError);
                return 1;
            }
        }
    } else {
        printf("Function returned:\n");
        printf("\tOfficial name: %s\n", remoteHost->h_name);
        for (pAlias = remoteHost->h_aliases; *pAlias != 0; pAlias++) {
            printf("\tAlternate name #%d: %s\n", ++i, *pAlias);
        }
        printf("\tAddress type: ");
        switch (remoteHost->h_addrtype) {
        case AF_INET:
            printf("AF_INET\n");
            break;
        case AF_NETBIOS:
            printf("AF_NETBIOS\n");
            break;
        default:
            printf(" %d\n", remoteHost->h_addrtype);
            break;
        }
        printf("\tAddress length: %d\n", remoteHost->h_length);

        i = 0;
        if (remoteHost->h_addrtype == AF_INET)
        {
            while (remoteHost->h_addr_list[i] != 0) {
                addr.s_addr = *(u_long *) remoteHost->h_addr_list[i++];
                printf("\tIP Address #%d: %s\n", i, inet_ntoa(addr));
            }
        }
        else if (remoteHost->h_addrtype == AF_NETBIOS)
        {
            printf("NETBIOS address was returned\n");
        }
    }

    return 0;
}
```

Windows Phone 8: This function is supported for Windows Phone Store apps on Windows Phone 8 and later.

Windows 8.1 and **Windows Server 2012 R2**: This function is supported for Windows Store apps on Windows 8.1, Windows Server 2012 R2, and later.

Requirements

Minimum supported client	Windows 8.1, Windows Vista [desktop apps | UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps | UWP apps]
Target Platform	Windows
Header	winsock.h (include Winsock2.h, Winsock.h)

Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

GetAddrInfoEx

GetAddrInfoW

WSAAsyncGetHostByName

Winsock Functions

Winsock Reference

addrinfo

addrinfoW

getaddrinfo

gethostbyaddr

gethostname

hostent

inet_addr

gethostname function (winsock.h)

10/22/2022 • 2 minutes to read • Edit Online

The **gethostname** function retrieves the standard host name for the local computer.

Syntax

```
int gethostname(
  [out] char *name,
  [in]  int  namelen
);
```

Parameters

`[out] name`

A pointer to a buffer that receives the local host name.

`[in] namelen`

The length, in bytes, of the buffer pointed to by the *name* parameter.

Return value

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

ERROR CODE	MEANING
WSAEFAULT	The *name* parameter is a **NULL** pointer or is not a valid part of the user address space. This error is also returned if the buffer size specified by *namelen* parameter is too small to hold the complete host name.
WSANOTINITIALISED	A successful WSAStartup call must occur before using this function.
WSAENETDOWN	The network subsystem has failed.
WSAEINPROGRESS	A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function.

Remarks

The **gethostname** function returns the name of the local host into the buffer specified by the *name* parameter. The host name is returned as a **null**-terminated string. The form of the host name is dependent on the Windows Sockets provider—it can be a simple host name, or it can be a fully qualified domain name. However, it is guaranteed that the name returned will be successfully parsed by gethostbyname and WSAAsyncGetHostByName.

The maximum length of the name returned in the buffer pointed to by the *name* parameter is dependent on the namespace provider.

If the **gethostname** function is used on a cluster resource on Windows Server 2008, Windows Server 2003, or Windows 2000 Server and the *CLUSTER_NETWORK_NAME* environment variable is defined, then the value in this environment variable overrides the actual hostname and is returned. On a cluster resource, the *CLUSTER_NETWORK_NAME* environment variable contains the name of the cluster.

The **gethostname** function queries namespace providers to determine the local host name using the SVCID_HOSTNAME GUID defined in the *Svgguid.h* header file. If no namespace provider responds, then the **gethostname** function returns the NetBIOS name of the local computer.

The maximum length, in bytes, of the string returned in the buffer pointed to by the *name* parameter is dependent on the namespace provider, but this string must be 256 bytes or less. So if a buffer of 256 bytes is passed in the *name* parameter and the *namelen* parameter is set to 256, the buffer size will always be adequate.

> **Note** If no local host name has been configured, **gethostname** must succeed and return a token host name that gethostbyname or WSAAsyncGetHostByName can resolve.

Windows Phone 8: This function is supported for Windows Phone Store apps on Windows Phone 8 and later.

Windows 8.1 and **Windows Server 2012 R2**: This function is supported for Windows Store apps on Windows 8.1, Windows Server 2012 R2, and later.

Requirements

Minimum supported client	Windows 8.1, Windows Vista [desktop apps | UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps | UWP apps]
Target Platform	Windows
Header	winsock.h (include Winsock2.h)
Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

GetAddrInfoW

WSAAsyncGetHostByName

Winsock Functions

Winsock Reference

gethostbyname

getpeername function (winsock.h)

10/22/2022 • 2 minutes to read • Edit Online

The **getpeername** function retrieves the address of the peer to which a socket is connected.

Syntax

```
int getpeername(
  [in]      SOCKET   s,
  [out]     sockaddr *name,
  [in, out] int      *namelen
);
```

Parameters

`[in] s`

A descriptor identifying a connected socket.

`[out] name`

The SOCKADDR structure that receives the address of the peer.

`[in, out] namelen`

A pointer to the size, in bytes, of the *name* parameter.

Return value

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

ERROR CODE	MEANING
WSANOTINITIALISED	A successful WSAStartup call must occur before using this function.
WSAENETDOWN	The network subsystem has failed.
WSAEFAULT	The *name* or the *namelen* parameter is not in a valid part of the user address space, or the *namelen* parameter is too small.
WSAEINPROGRESS	A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function.
WSAENOTCONN	The socket is not connected.

WSAENOTSOCK	The descriptor is not a socket.

Remarks

The **getpeername** function retrieves the address of the peer connected to the socket *s* and stores the address in the SOCKADDR structure identified by the *name* parameter. This function works with any address family and it simply returns the address to which the socket is connected. The **getpeername** function can be used only on a connected socket.

For datagram sockets, only the address of a peer specified in a previous connect call will be returned. Any address specified by a previous sendto call will not be returned by **getpeername**.

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

Windows Phone 8: This function is supported for Windows Phone Store apps on Windows Phone 8 and later.

Windows 8.1 and **Windows Server 2012 R2**: This function is supported for Windows Store apps on Windows 8.1, Windows Server 2012 R2, and later.

Requirements

Minimum supported client	Windows 8.1, Windows Vista [desktop apps | UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps | UWP apps]
Target Platform	Windows
Header	winsock.h (include Winsock2.h)
Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

Winsock Functions

Winsock Reference

bind

connect

getsockname

sendto

socket

getprotobyname function (winsock.h)

10/22/2022 • 2 minutes to read • Edit Online

The **getprotobyname** function retrieves the protocol information corresponding to a protocol name.

Syntax

```
protoent * getprotobyname(
  [in] const char *name
);
```

Parameters

`[in] name`

Pointer to a null-terminated protocol name.

Return value

If no error occurs, **getprotobyname** returns a pointer to the protoent. Otherwise, it returns a null pointer and a specific error number can be retrieved by calling WSAGetLastError.

ERROR CODE	MEANING
WSANOTINITIALISED	A successful WSAStartup call must occur before using this function.
WSAENETDOWN	The network subsystem has failed.
WSAHOST_NOT_FOUND	Authoritative answer protocol not found.
WSATRY_AGAIN	A nonauthoritative protocol not found, or server failure.
WSANO_RECOVERY	Nonrecoverable errors, the protocols database is not accessible.
WSANO_DATA	Valid name, no data record of requested type.
WSAEINPROGRESS	A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function.
WSAEFAULT	The *name* parameter is not a valid part of the user address space.

WSAEINTR	A blocking Windows Socket 1.1 call was canceled through WSACancelBlockingCall.

Remarks

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

Windows Phone 8: This function is supported for Windows Phone Store apps on Windows Phone 8 and later.

Windows 8.1 and **Windows Server 2012 R2**: This function is supported for Windows Store apps on Windows 8.1, Windows Server 2012 R2, and later.

Requirements

Minimum supported client	Windows 8.1, Windows Vista [desktop apps | UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps | UWP apps]
Target Platform	Windows
Header	winsock.h (include Winsock2.h)
Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

WSAAsyncGetProtoByName

Winsock Functions

Winsock Reference

getprotobynumber

getprotobynumber function (winsock.h)

10/22/2022 • 2 minutes to read • Edit Online

The **getprotobynumber** function retrieves protocol information corresponding to a protocol number.

Syntax

```
protoent * getprotobynumber(
  int proto
);
```

Parameters

`proto`

TBD

Return value

If no error occurs, **getprotobynumber** returns a pointer to the protoent structure. Otherwise, it returns a null pointer and a specific error number can be retrieved by calling WSAGetLastError.

ERROR CODE	MEANING
WSANOTINITIALISED	A successful WSAStartup call must occur before using this function.
WSAENETDOWN	The network subsystem has failed.
WSAHOST_NOT_FOUND	Authoritative answer protocol not found.
WSATRY_AGAIN	A nonauthoritative Protocol not found, or server failure.
WSANO_RECOVERY	Nonrecoverable errors, the protocols database is not accessible.
WSANO_DATA	Valid name, no data record of requested type.
WSAEINPROGRESS	A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function.
WSAEINTR	A blocking Windows Socket 1.1 call was canceled through WSACancelBlockingCall.

Remarks

This **getprotobynumber** function returns a pointer to the protoent structure as previously described in getprotobyname. The contents of the structure correspond to the given protocol number.

The pointer that is returned points to the structure allocated by Windows Sockets. The application must never attempt to modify this structure or to free any of its components. Furthermore, only one copy of this structure is allocated per thread, so the application should copy any information that it needs before issuing any other Windows Sockets function calls.

Windows Phone 8: This function is supported for Windows Phone Store apps on Windows Phone 8 and later.

Windows 8.1 and **Windows Server 2012 R2**: This function is supported for Windows Store apps on Windows 8.1, Windows Server 2012 R2, and later.

Requirements

Minimum supported client	Windows 8.1, Windows Vista [desktop apps | UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps | UWP apps]
Target Platform	Windows
Header	winsock.h (include Winsock2.h)
Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

WSAAsyncGetProtoByNumber

Winsock Functions

Winsock Reference

getprotobyname

getservbyname function (winsock.h)

10/22/2022 • 2 minutes to read • Edit Online

The **getservbyname** function retrieves service information corresponding to a service name and protocol.

Syntax

```
servent * getservbyname(
  [in] const char *name,
  [in] const char *proto
);
```

Parameters

`[in] name`

A pointer to a **null**-terminated service name.

`[in] proto`

A pointer to a **null**-terminated protocol name. If this pointer is **NULL**, the **getservbyname** function returns the first service entry where *name* matches the **s_name** member of the servent structure or the **s_aliases** member of the **servent** structure. Otherwise, **getservbyname** matches both the *name* and the *proto*.

Return value

If no error occurs, **getservbyname** returns a pointer to the servent structure. Otherwise, it returns a **null** pointer and a specific error number can be retrieved by calling WSAGetLastError.

ERROR CODE	MEANING
WSANOTINITIALISED	A successful WSAStartup call must occur before using this function.
WSAENETDOWN	The network subsystem has failed.
WSAHOST_NOT_FOUND	Authoritative Answer Service not found.
WSATRY_AGAIN	A nonauthoritative Service not found, or server failure.
WSANO_RECOVERY	Nonrecoverable errors, the services database is not accessible.
WSANO_DATA	Valid name, no data record of requested type.

WSAEINPROGRESS	A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function.
WSAEINTR	A blocking Windows Socket 1.1 call was canceled through WSACancelBlockingCall.

Remarks

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

The pointer that is returned points to the **servent** structure allocated by the Windows Sockets library. The application must never attempt to modify this structure or to free any of its components. Furthermore, only one copy of this structure is allocated per thread, so the application should copy any information it needs before issuing any other Windows Sockets function calls.

Windows Phone 8: This function is supported for Windows Phone Store apps on Windows Phone 8 and later.

Windows 8.1 and **Windows Server 2012 R2**: This function is supported for Windows Store apps on Windows 8.1, Windows Server 2012 R2, and later.

Requirements

Minimum supported client	Windows 8.1, Windows Vista [desktop apps | UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps | UWP apps]
Target Platform	Windows
Header	winsock.h (include Winsock2.h)
Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

WSAAsyncGetServByName

Winsock Functions

Winsock Reference

getservbyport

getservbyport function (winsock.h)

10/22/2022 • 2 minutes to read • Edit Online

The **getservbyport** function retrieves service information corresponding to a port and protocol.

Syntax

```
servent * getservbyport(
  [in] int        port,
  [in] const char *proto
);
```

Parameters

`[in] port`

Port for a service, in network byte order.

`[in] proto`

Optional pointer to a protocol name. If this is null, **getservbyport** returns the first service entry for which the *port* matches the **s_port** of the servent structure. Otherwise, **getservbyport** matches both the *port* and the *proto* parameters.

Return value

If no error occurs, **getservbyport** returns a pointer to the servent structure. Otherwise, it returns a null pointer and a specific error number can be retrieved by calling WSAGetLastError.

ERROR CODE	MEANING
WSANOTINITIALISED	A successful WSAStartup call must occur before using this function.
WSAENETDOWN	The network subsystem has failed.
WSAHOST_NOT_FOUND	Authoritative Answer Service not found.
WSATRY_AGAIN	A nonauthoritative Service not found, or server failure.
WSANO_RECOVERY	Nonrecoverable errors, the services database is not accessible.
WSANO_DATA	Valid name, no data record of requested type.

WSAEINPROGRESS	A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function.
WSAEFAULT	The *proto* parameter is not a valid part of the user address space.
WSAEINTR	A blocking Windows Socket 1.1 call was canceled through WSACancelBlockingCall.

Remarks

The **getservbyport** function returns a pointer to a servent structure as it does in the getservbyname function.

The **servent** structure is allocated by Windows Sockets. The application must never attempt to modify this structure or to free any of its components. Furthermore, only one copy of this structure is allocated per thread, so the application should copy any information it needs before issuing any other Windows Sockets function calls.

Windows Phone 8: This function is supported for Windows Phone Store apps on Windows Phone 8 and later.

Windows 8.1 and **Windows Server 2012 R2**: This function is supported for Windows Store apps on Windows 8.1, Windows Server 2012 R2, and later.

Requirements

Minimum supported client	Windows 8.1, Windows Vista [desktop apps | UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps | UWP apps]
Target Platform	Windows
Header	winsock.h (include Winsock2.h)
Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

WSAAsyncGetServByPort

Winsock Functions

Winsock Reference

getservbyname

getsockname function (winsock.h)

10/22/2022 • 2 minutes to read • Edit Online

The **getsockname** function retrieves the local name for a socket.

Syntax

```
int getsockname(
  [in]      SOCKET   s,
  [out]     sockaddr *name,
  [in, out] int      *namelen
);
```

Parameters

`[in] s`

Descriptor identifying a socket.

`[out] name`

Pointer to a SOCKADDR structure that receives the address (name) of the socket.

`[in, out] namelen`

Size of the *name* buffer, in bytes.

Return value

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

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

WSAEINVAL	The socket has not been bound to an address with bind, or ADDR_ANY is specified in **bind** but connection has not yet occurred.

Remarks

The **getsockname** function retrieves the current name for the specified socket descriptor in *name*. It is used on the bound or connected socket specified by the *s* parameter. The local association is returned. This call is especially useful when a connect call has been made without doing a bind first; the **getsockname** function provides the only way to determine the local association that has been set by the system.

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

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

Windows Phone 8: This function is supported for Windows Phone Store apps on Windows Phone 8 and later.

Windows 8.1 and **Windows Server 2012 R2**: This function is supported for Windows Store apps on Windows 8.1, Windows Server 2012 R2, and later.

Requirements

Minimum supported client	Windows 8.1, Windows Vista [desktop apps | UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps | UWP apps]
Target Platform	Windows
Header	winsock.h (include Winsock2.h)
Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

SOCKADDR

Winsock Functions

Winsock Reference

bind

getpeername

socket

getsockopt function (winsock.h)

10/22/2022 • 17 minutes to read • Edit Online

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

Syntax

```
int getsockopt(
  [in]      SOCKET s,
  [in]      int    level,
  [in]      int    optname,
  [out]     char   *optval,
  [in, out] int    *optlen
);
```

Parameters

`[in] s`

A descriptor identifying a socket.

`[in] level`

The level at which the option is defined. Example: SOL_SOCKET.

`[in] optname`

The socket option for which the value is to be retrieved. Example: SO_ACCEPTCONN. The *optname* value must be a socket option defined within the specified *level*, or behavior is undefined.

`[out] optval`

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

`[in, out] optlen`

A pointer to the size, in bytes, of the *optval* buffer.

Return value

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

ERROR CODE	MEANING
WSANOTINITIALISED	A successful WSAStartup call must occur before using this function.
WSAENETDOWN	**Note** The network subsystem has failed.

WSAEFAULT	One of the *optval* or the *optlen* parameters is not a valid part of the user address space, or the *optlen* parameter is too small.
WSAEINPROGRESS	A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function.
WSAEINVAL	The *level* parameter is unknown or invalid.
WSAENOPROTOOPT	The option is unknown or unsupported by the indicated protocol family.
WSAENOTSOCK	The descriptor is not a socket.

Remarks

The **getsockopt** function retrieves the current value for a socket option associated with a socket of any type, in any state, and stores the result in *optval*. Options can exist at multiple protocol levels, but they are always present at the uppermost socket level. Options affect socket operations, such as the packet routing and OOB data transfer.

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

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

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

The following options are supported for **getsockopt**. The Type column identifies the type of data addressed by *optval*.

For more information on socket options, see Socket Options.

The following table of value for the *optname* parameter are valid when the *level* parameter is set to **SOL_SOCKET**.

VALUE	TYPE	MEANING
SO_ACCEPTCONN	BOOL	The socket is listening.
SO_BROADCAST	BOOL	The socket is configured for the transmission and receipt of broadcast messages.
SO_BSP_STATE	CSADDR_INFO	Returns the local address, local port, remote address, remote port, socket type, and protocol used by a socket.
SO_CONDITIONAL_ACCEPT	BOOL	Returns current socket state, either from a previous call to setsockopt or the system default.

SO_CONNECT_TIME	DWORD	Returns the number of seconds a socket has been connected. This socket option is valid for connection oriented protocols only.
SO_DEBUG	BOOL	Debugging is enabled.
SO_DONTLINGER	BOOL	If **TRUE**, the SO_LINGER option is disabled.
SO_DONTROUTE	BOOL	Routing is disabled. Setting this succeeds but is ignored on AF_INET sockets; fails on AF_INET6 sockets with WSAENOPROTOOPT. This option is not supported on ATM sockets.
SO_ERROR	int	Retrieves error status and clear.
SO_EXCLUSIVEADDRUSE	BOOL	Prevents any other socket from binding to the same address and port. This option must be set before calling the bind function.
SO_GROUP_ID	GROUP	Reserved.
SO_GROUP_PRIORITY	int	Reserved.
SO_KEEPALIVE	BOOL	Keep-alives are being sent. Not supported on ATM sockets.
SO_LINGER	LINGER structure	Returns the current linger options.
SO_MAX_MSG_SIZE	unsigned int	The maximum size of a message for message-oriented socket types (for example, SOCK_DGRAM). Has no meaning for stream oriented sockets.
SO_OOBINLINE	BOOL	OOB data is being received in the normal data stream. (See section Windows Sockets 1.1 Blocking Routines and EINPROGRESS for a discussion of this topic.)
SO_PORT_SCALABILITY	BOOL	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.
SO_PROTOCOL_INFO	WSAPROTOCOL_INFO	A description of the protocol information for the protocol that is bound to this socket.

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.
SO_REUSEADDR	BOOL	The socket can be bound to an address which is already in use. Not applicable for ATM sockets.
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.
SO_TYPE	int	The type of the socket (for example, SOCK_STREAM).
PVD_CONFIG	Service Provider Dependent	An opaque data structure object from the service provider associated with socket *s*. This object stores the current configuration information of the service provider. The exact format of this data structure is service provider specific.

level = **IPPROTO_TCP**

See **TCP_NODELAY** in IPPROTO_TCP socket options. Also see that topic for more complete and detailed information about socket options for *level* = **IPPROTO_TCP**.

The following table of value for the *optname* parameter are valid when the *level* parameter is set to **NSPROTO_IPX**.

> **Note** Windows NT supports all IPX options. Windows Me, Windows 98, and Windows 95 support only the following options:
> IPX_PTYPE
> IPX_FILTERPTYPE
> IPX_DSTYPE
> IPX_RECVHDR
> IPX_MAXSIZE
> IPX_ADDRESS

VALUE	TYPE	MEANING
IPX_PTYPE	int	Retrieves the IPX packet type.
IPX_FILTERPTYPE	int	Retrieves the receive filter packet type
IPX_DSTYPE	int	Obtains the value of the data stream field in the SPX header on every packet sent.

IPX_EXTENDED_ADDRESS	BOOL	Finds out whether extended addressing is enabled.
IPX_RECVHDR	BOOL	Finds out whether the protocol header is sent up on all receive headers.
IPX_MAXSIZE	int	Obtains the maximum data size that can be sent.
IPX_ADDRESS	IPX_ADDRESS_DATA structure	Obtains information about a specific adapter to which IPX is bound. Adapter numbering is base zero. The **adapternum** member is filled in upon return.
IPX_GETNETINFO	IPX_NETNUM_DATA structure	Obtains information about a specific IPX network number. If not available in the cache, uses RIP to obtain information.
IPX_GETNETINFO_NORIP	IPX_NETNUM_DATA structure	Obtains information about a specific IPX network number. If not available in the cache, will not use RIP to obtain information, and returns error.
IPX_SPXGETCONNECTIONSTATUS	IPX_SPXCONNSTATUS_DATA structure	Retrieves information about a connected SPX socket.
IPX_ADDRESS_NOTIFY	IPX_ADDRESS_DATA structure	Retrieves status notification when changes occur on an adapter to which IPX is bound.
IPX_MAX_ADAPTER_NUM	int	Retrieves maximum number of adapters present, numbered as base zero.
IPX_RERIPNETNUMBER	IPX_NETNUM_DATA structure	Similar to IPX_GETNETINFO, but forces IPX to use RIP for resolution, even if the network information is in the local cache.
IPX_IMMEDIATESPXACK	BOOL	Directs SPX connections not to delay before sending an ACK. Applications without back-and-forth traffic should set this to **TRUE** to increase performance.
TCP_MAXSEG	int	Receives TCP maximum-segment size. Supported in Windows 10 and newer versions.

The following table lists value for the *optname* that represent BSD socket options that are not supported by the **getsockopt** function.

VALUE	TYPE	MEANING

SO_RCVLOWAT	int	Receives low watermark.
SO_RCVTIMEO	int	Receives time-out.
SO_SNDLOWAT	int	Sends low watermark.
SO_SNDTIMEO	int	Sends time-out.
TCP_MAXSEG	int	Receives TCP maximum-segment size. Not supported in versions before Windows 10.

> **Note** When using the recv function, if no data arrives during the period specified in SO_RCVTIMEO, the **recv** function completes. In Windows versions prior to Windows 2000, any data received subsequently fails with WSAETIMEDOUT. In Windows 2000 and later, if no data arrives within the period specified in SO_RCVTIMEO, the **recv** function returns WSAETIMEDOUT, and if data is received, **recv** returns SUCCESS.

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

More detailed information on some of the socket options for the *optname* parameter supported by the **getsockopt** function are listed below.

SO_CONNECT_TIME
This option returns the number of seconds a socket has been connected. This option is valid for connection oriented protocols only.
The SO_CONNECT_TIME option can be used with the **getsockopt** function to check whether a connection has been established. This option can also be used while a ConnectEx function call is in progress. If a connection is established, the SO_CONNECT_TIME option can determine how long the connection has been established. If the socket is not connected, the **getsockopt** returns SOCKET_ERROR. Checking a connection like this is necessary to see if connections that have been established for a while, without sending any data. It is recommended that applications terminate these connections.

SO_DEBUG

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

SO_ERROR
The SO_ERROR option returns and resets the per socket–based error code, which is different from the per thread based–error code that is handled using the WSAGetLastError and WSASetLastError function calls. A successful call using the socket does not reset the socket based error code returned by the SO_ERROR option.
SO_EXCLUSIVEADDRUSE
Prevents any other socket from binding to the same address and port. This option must be set before calling the bind function. See the SO_EXCLUSIVEADDRUSE reference for more information.
SO_GROUP_ID

> **Note** This option is reserved. This option is also exclusive to **getsockopt**; the value should be **NULL**.

SO_GROUP_PRIORITY

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

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

SO_KEEPALIVE

An application can request that a TCP/IP service provider enable the use of keep-alive packets on TCP connections by turning on the SO_KEEPALIVE socket option. This option queries the current value of the keep-alive option on a socket. A Windows Sockets provider need not support the use of keep-alive: if it does, the precise semantics are implementation-specific but should conform to section 4.2.3.6 on the *Requirements for Internet Hosts—Communication Layers* specified in RFC 1122 available at the IETF website. If a connection is dropped as the result of keep-alives the error code WSAENETRESET is returned to any calls in progress on the socket, and any subsequent calls will fail with WSAENOTCONN. SO_KEEPALIVE is not supported on ATM sockets, and requests to enable the use of keep-alive packets on an ATM socket results in an error being returned by the socket.

SO_LINGER

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

SO_MAX_MSG_SIZE

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

SO_PROTOCOL_INFO

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

SO_SNDBUF

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

SO_REUSEADDR

By default, a socket cannot be bound (see bind) to a local address that is already in use. On occasion, however, it can be necessary to reuse an address in this way. Because every connection is uniquely identified by the combination of local and remote addresses, there is no problem with having two sockets bound to the same local address as long as the remote addresses are different. To inform the Windows Sockets provider that a **bind** on a socket should not be disallowed because the desired address is already in use by another socket, the application should set the SO_REUSEADDR socket option for the socket before issuing the **bind**. Note that the option is interpreted only at the time of the **bind**: it is therefore unnecessary (but harmless) to set the option on a socket that is not to be bound to an existing address, and setting or resetting the option after the **bind** has no effect on this or any other socket. SO_REUSEADDR is not applicable for ATM sockets, and although requests to reuse and address do not result in an error, they have no effect on when an ATM socket is in use.

PVD_CONFIG

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

TCP_NODELAY

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

> **Note** When issuing a blocking Winsock call such as **getsockopt**, Winsock may need to wait for a network event before the call can complete. Winsock performs an alertable wait in this situation, which can be interrupted by an asynchronous procedure call (APC) scheduled on the same thread. Issuing another blocking Winsock call inside an APC that interrupted an ongoing blocking Winsock call on the same thread will lead to undefined behavior, and must never be attempted by Winsock clients.

Example Code

The following code sample demonstrates the use of the **getsockopt** function.

```
#include <stdio.h>
#include "winsock2.h"
#include <windows.h>

void main() {

  //----------------------------------------
  // Declare variables
  WSADATA wsaData;
  SOCKET ListenSocket;
  sockaddr_in service;

  //----------------------------------------
  // Initialize Winsock
  int iResult = WSAStartup( MAKEWORD(2,2), &wsaData );
  if( iResult != NO_ERROR )
    printf("Error at WSAStartup\n");

  //----------------------------------------
  // Create a listening socket
  ListenSocket = socket( AF_INET, SOCK_STREAM, IPPROTO_TCP );
  if (ListenSocket == INVALID_SOCKET) {
    printf("Error at socket()\n");
    WSACleanup();
    return;
  }

  //----------------------------------------
  // Bind the socket to the local IP address
  // and port 27015
  hostent* thisHost;
  char* ip;
  u_short port;
  port = 27015;
  thisHost = gethostbyname("");
  ip = inet_ntoa (*(struct in_addr *)*thisHost->h_addr_list);

  service.sin_family = AF_INET;
  service.sin_addr.s_addr = inet_addr(ip);
  service.sin_port = htons(port);
```

```
  if ( bind( ListenSocket,(SOCKADDR*) &service, sizeof(service) )  == SOCKET_ERROR ) {
    printf("bind failed\n");
    closesocket(ListenSocket);
    return;
  }

  //----------------------------------------
  // Initialize variables and call getsockopt.
  // The SO_ACCEPTCONN parameter is a socket option
  // that tells the function to check whether the
  // socket has been put in listening mode or not.
  // The various socket options return different
  // information about the socket. This call should
  // return 0 to the optVal parameter, since the socket
  // is not in listening mode.
  int optVal;
  int optLen = sizeof(int);

  if (getsockopt(ListenSocket,
    SOL_SOCKET,
    SO_ACCEPTCONN,
    (char*)&optVal,
    &optLen) != SOCKET_ERROR)
    printf("SockOpt Value: %ld\n", optVal);

  //----------------------------------------
  // Put the listening socket in listening mode.
  if (listen( ListenSocket, 100 ) == SOCKET_ERROR) {
    printf("error listening\n");
  }

  //----------------------------------------
  // Call getsockopt again to verify that
  // the socket is in listening mode.
  if (getsockopt(ListenSocket,
    SOL_SOCKET,
    SO_ACCEPTCONN,
    (char*)&optVal,
    &optLen) != SOCKET_ERROR)
    printf("SockOpt Value: %ld\n", optVal);

  WSACleanup();
  return;
}
```

Notes for IrDA Sockets

- The Af_irda.h header file must be explicitly included.
- Windows returns WSAENETDOWN to indicate the underlying transceiver driver failed to initialize with the IrDA protocol stack.
- IrDA supports several special socket options:

VALUE	TYPE	MEANING
IRLMP_ENUMDEVICES	*DEVICELIST	Describes devices in range.
IRLMP_IAS_QUERY	*IAS_QUERY	Retrieve IAS attributes.

Before an IrDA socket connection can be initiated, a device address must be obtained by performing a **getsockopt**(,,IRLMP_ENUMDEVICES,,) function call, which returns a list of all available IrDA devices. A device

address returned from the function call is copied into a SOCKADDR_IRDA structure, which in turn is used by a subsequent call to the connect function call.

Discovery can be performed in two ways:

1. First, performing a getsockopt function call with the IRLMP_ENUMDEVICES option causes a single discovery to be run on each idle adapter. The list of discovered devices and cached devices (on active adapters) is returned immediately.

   The following code demonstrates this approach.

```
#include <winsock2.h>
#include <ws2tcpip.h>
#include <af_irda.h>
#include <stdio.h>
#include <windows.h>

// link with Ws2_32.lib

int __cdecl main()
{

    //-----------------------------------------
    // Declare and initialize variables
    WSADATA wsaData;

    int iResult;
    int i;
    DWORD dwError;

    SOCKET Sock = INVALID_SOCKET;

#define DEVICE_LIST_LEN    10


    SOCKADDR_IRDA DestSockAddr = { AF_IRDA, 0, 0, 0, 0, "SampleIrDAService" };

    unsigned char DevListBuff[sizeof (DEVICELIST) -
                              sizeof (IRDA_DEVICE_INFO) +
                              (sizeof (IRDA_DEVICE_INFO) * DEVICE_LIST_LEN)];

    int DevListLen = sizeof (DevListBuff);
    PDEVICELIST pDevList;

    pDevList = (PDEVICELIST) & DevListBuff;

    // Initialize Winsock
    iResult = WSAStartup(MAKEWORD(2, 2), &wsaData);
    if (iResult != 0) {
        printf("WSAStartup failed: %d\n", iResult);
        return 1;
    }

    Sock = socket(AF_IRDA, SOCK_STREAM, 0);
    if (Sock == INVALID_SOCKET) {
        dwError = WSAGetLastError();
        printf
            ("socket failed trying to create an AF_IRDA socket with error %d\n",
             dwError);

        if (dwError == WSAEAFNOSUPPORT) {
            printf("Check that the local computer has an infrared device\n");
            printf
                ("and a device driver is installed for the infrared device\n");
        }
        WSACleanup();
        return 1;
```

```
    }
    // Sock is not in connected state
    iResult = getsockopt(Sock, SOL_IRLMP, IRLMP_ENUMDEVICES,
                         (char *) pDevList, &DevListLen);
    if (iResult == SOCKET_ERROR) {
        printf("getsockopt failed with error %d\n", WSAGetLastError());
        WSACleanup();
        return 1;
    }

    if (pDevList->numDevice == 0) {
        // no devices discovered or cached
        // not a bad idea to run a couple of times
        printf("No IRDA devices were discovered or cached\n");
    } else {
        // one per discovered device
        for (i = 0; i < (int) pDevList->numDevice; i++) {
            // typedef struct _IRDA_DEVICE_INFO
            // {
            //     u_char    irdaDeviceID[4];
            //     char      irdaDeviceName[22];
            //     u_char    irdaDeviceHints1;
            //     u_char    irdaDeviceHints2;
            //     u_char    irdaCharSet;
            // } _IRDA_DEVICE_INFO;

            // pDevList->Device[i]. see _IRDA_DEVICE_INFO for fields
            // display the device names and let the user select one
        }
    }

    // assume the user selected the first device [0]
    memcpy(&DestSockAddr.irdaDeviceID[0], &pDevList->Device[0].irdaDeviceID[0],
           4);

    iResult = connect(Sock, (const struct sockaddr *) &DestSockAddr,
                      sizeof (SOCKADDR_IRDA));
    if (iResult == SOCKET_ERROR) {
        printf("connect failed with error %d\n", WSAGetLastError());
    } else
        printf("connect to first IRDA device was successful\n");

    WSACleanup();
    return 0;
}
```

2. The second approach to performing discovery of IrDA device addresses is to perform a lazy discovery; in this approach, the application is not notified until the discovered devices list changes from the last discovery run by the stack.

The **DEVICELIST** structure shown in the Type column in the previous table is an extendible array of device descriptions. IrDA fills in as many device descriptions as can fit in the specified buffer. The device description consists of a device identifier necessary to form a sockaddr_irda structure, and a displayable string describing the device.

The **IAS_QUERY** structure shown in the Type column in the previous table is used to retrieve a single attribute of a single class from a peer device's IAS database. The application specifies the device and class to query and the attribute and attribute type. Note that the device would have been obtained previously by a call to **getsockopt**(IRLMP_ENUMDEVICES). It is expected that the application allocates a buffer, of the necessary size, for the returned parameters.

Many level socket options are not meaningful to IrDA; only SO_LINGER and SO_DONTLINGER are specifically supported.

Windows Phone 8: This function is supported for Windows Phone Store apps on Windows Phone 8 and later.

Windows 8.1 and **Windows Server 2012 R2**: This function is supported for Windows Store apps on Windows 8.1, Windows Server 2012 R2, and later.

Requirements

Minimum supported client	Windows 8.1, Windows Vista [desktop apps | UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps | UWP apps]
Target Platform	Windows
Header	winsock.h (include Winsock2.h)
Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

IPPROTO_IP Socket Options

IPPROTO_IPV6 Socket Options

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

Socket Options

WSAAsyncSelect

WSAConnect

WSAGetLastError

WSAIoctl

WSASetLastError

Winsock Functions

ioctlsocket

recv

setsockopt

socket

HOSTENT structure (winsock.h)

10/22/2022 • 3 minutes to read • Edit Online

The **hostent** structure is used by functions to store information about a given host, such as host name, IPv4 address, and so forth. An application should never attempt to modify this structure or to free any of its components. Furthermore, only one copy of the **hostent** structure is allocated per thread, and an application should therefore copy any information that it needs before issuing any other Windows Sockets API calls.

Syntax

```
typedef struct hostent {
  char  *h_name;
  char  **h_aliases;
  short h_addrtype;
  short h_length;
  char  **h_addr_list;
} HOSTENT, *PHOSTENT, *LPHOSTENT;
```

Members

`h_name`

The official name of the host (PC). If using the DNS or similar resolution system, it is the Fully Qualified Domain Name (FQDN) that caused the server to return a reply. If using a local hosts file, it is the first entry after the IPv4 address.

`h_aliases`

A **NULL**-terminated array of alternate names.

`h_addrtype`

The type of address being returned.

`h_length`

The length, in bytes, of each address.

`h_addr_list`

A **NULL**-terminated list of addresses for the host. Addresses are returned in network byte order. The macro **h_addr** is defined to be `h_addr_list[0]` for compatibility with older software.

Remarks

The gethostbyaddr and gethostbyname functions returns a pointer to a **hostent** structure—a structure allocated by Windows Sockets. The **hostent** structure contains the results of a successful search for the host specified in the *name* parameter.

The memory for the **hostent** structure returned by the gethostbyaddr and gethostbyname functions is allocated internally by the Winsock DLL from thread local storage. Only a single **hostent** structure is allocated and used, no matter how many times the **gethostbyaddr** or **gethostbyname** functions are called on the thread. The returned **hostent** structure must be copied to an application buffer if additional calls are to be made to the

gethostbyaddr or **gethostbyname** functions on the same thread. Otherwise, the return value will be overwritten by subsequent **gethostbyaddr** or **gethostbyname** calls on the same thread. The internal memory allocated for the returned **hostent** structure is released by the Winsock DLL when the thread exits.

An application should not try to release the memory used by the returned **hostent** structure. The application must never attempt to modify this structure or to free any of its components. Furthermore, only one copy of this structure is allocated per thread, so the application should copy any information it needs before issuing any other function calls to gethostbyaddr or gethostbyname.

Examples

The following examples demonstrates the use of the **hostent** structure with the gethostbyname function.

```
#define WIN32_LEAN_AND_MEAN

#include <winsock2.h>
#include <ws2tcpip.h>
#include <stdio.h>

// Need to link with Ws2_32.lib
#pragma comment(lib, "ws2_32.lib")

int main(int argc, char **argv)
{

    //-----------------------------------------
    // Declare and initialize variables
    WSADATA wsaData;
    int iResult;

    DWORD dwError;
    int i = 0;

    struct hostent *remoteHost;
    char *host_name;
    struct in_addr addr;

    char **pAlias;

    // Validate the parameters
    if (argc != 2) {
        printf("usage: %s ipv4address\n", argv[0]);
        printf(" or\n");
        printf("       %s hostname\n", argv[0]);
        printf("  to return the host\n");
        printf("       %s 127.0.0.1\n", argv[0]);
        printf("  to return the IP addresses for a host\n");
        printf("       %s www.contoso.com\n", argv[0]);
        return 1;
    }
    // Initialize Winsock
    iResult = WSAStartup(MAKEWORD(2, 2), &wsaData);
    if (iResult != 0) {
        printf("WSAStartup failed: %d\n", iResult);
        return 1;
    }

    host_name = argv[1];

// If the user input is an alpha name for the host, use gethostbyname()
// If not, get host by addr (assume IPv4)
    if (isalpha(host_name[0])) {        /* host address is a name */
        printf("Calling gethostbyname with %s\n", host_name);
        remoteHost = gethostbyname(host_name);
    } else {
        printf("Calling gethostbyaddr with %s\n", host_name);
        addr.s_addr = inet_addr(host_name);
```

```
        if (addr.s_addr == INADDR_NONE) {
            printf("The IPv4 address entered must be a legal address\n");
            return 1;
        } else
            remoteHost = gethostbyaddr((char *) &addr, 4, AF_INET);
    }

    if (remoteHost == NULL) {
        dwError = WSAGetLastError();
        if (dwError != 0) {
            if (dwError == WSAHOST_NOT_FOUND) {
                printf("Host not found\n");
                return 1;
            } else if (dwError == WSANO_DATA) {
                printf("No data record found\n");
                return 1;
            } else {
                printf("Function failed with error: %ld\n", dwError);
                return 1;
            }
        }
    } else {
        printf("Function returned:\n");
        printf("\tOfficial name: %s\n", remoteHost->h_name);
        for (pAlias = remoteHost->h_aliases; *pAlias != 0; pAlias++) {
            printf("\tAlternate name #%d: %s\n", ++i, *pAlias);
        }
        printf("\tAddress type: ");
        switch (remoteHost->h_addrtype) {
        case AF_INET:
            printf("AF_INET\n");
            break;
        case AF_INET6:
            printf("AF_INET6\n");
            break;
        case AF_NETBIOS:
            printf("AF_NETBIOS\n");
            break;
        default:
            printf(" %d\n", remoteHost->h_addrtype);
            break;
        }
        printf("\tAddress length: %d\n", remoteHost->h_length);

        if (remoteHost->h_addrtype == AF_INET) {
            while (remoteHost->h_addr_list[i] != 0) {
                addr.s_addr = *(u_long *) remoteHost->h_addr_list[i++];
                printf("\tIPv4 Address #%d: %s\n", i, inet_ntoa(addr));
            }
        } else if (remoteHost->h_addrtype == AF_INET6)
            printf("\tRemotehost is an IPv6 address\n");
    }

    return 0;
}
```

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]

Header	winsock.h (include Winsock2.h)

See also

GetAddrInfoEx

GetAddrInfoW

addrinfo

addrinfoW

getaddrinfo

gethostbyaddr

gethostbyname

htonl function (winsock.h)

10/22/2022 • 2 minutes to read • Edit Online

The **htonl** function converts a **u_long** from host to TCP/IP network byte order (which is big-endian).

Syntax

```
u_long htonl(
  [in] u_long hostlong
);
```

Parameters

`[in] hostlong`

A 32-bit number in host byte order.

Return value

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

Remarks

The **htonl** function takes a 32-bit number in host byte order and returns a 32-bit number in the network byte order used in TCP/IP networks (the AF_INET or AF_INET6 address family).

The **htonl** function can be used to convert an IPv4 address in host byte order to the IPv4 address in network byte order. This function does not do any checking to determine if the *hostlong* parameter is a valid IPv4 address.

The **htonl** function does not require that the Winsock DLL has previously been loaded with a successful call to the WSAStartup function.

Windows 8.1 and **Windows Server 2012 R2**: This function is supported for Windows Store apps on Windows 8.1, Windows Server 2012 R2, and later.

Requirements

Minimum supported client	Windows 8.1, Windows Vista [desktop apps | UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps | UWP apps]
Target Platform	Windows
Header	winsock.h (include Winsock2.h)
Library	Ws2_32.lib

DLL	Ws2_32.dll

See also

InetNtop

InetPton

WSAHtonl

WSAHtons

WSANtohl

WSANtohs

htond

htonf

htonll

htons

inet_addr

inet_ntoa

ntohd

ntohf

ntohll

ntohs

htons function (winsock.h)

10/22/2022 • 2 minutes to read • Edit Online

The **htons** function converts a **u_short** from host to TCP/IP network byte order (which is big-endian).

Syntax

```
u_short htons(
  [in] u_short hostshort
);
```

Parameters

`[in] hostshort`

A 16-bit number in host byte order.

Return value

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

Remarks

The **htons** function takes a 16-bit number in host byte order and returns a 16-bit number in network byte order used in TCP/IP networks (the AF_INET or AF_INET6 address family).

The **htons** function can be used to convert an IP port number in host byte order to the IP port number in network byte order.

The **htons** function does not require that the Winsock DLL has previously been loaded with a successful call to the WSAStartup function.

Windows 8.1 and **Windows Server 2012 R2**: This function is supported for Windows Store apps on Windows 8.1, Windows Server 2012 R2, and later.

Requirements

Minimum supported client	Windows 8.1, Windows Vista [desktop apps | UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps | UWP apps]
Target Platform	Windows
Header	winsock.h (include Winsock2.h)
Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

InetNtop

InetPton

WSAHtonl

WSAHtons

WSANtohl

WSANtohs

htond

htonf

htonl

htonll

inet_addr

inet_ntoa

ntohd

ntohf

ntohl

ntohll

ntohs

inet_addr function (winsock.h)

10/22/2022 • 4 minutes to read • Edit Online

The **inet_addr** function converts a string containing an IPv4 dotted-decimal address into a proper address for the IN_ADDR structure.

Syntax

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

Parameters

`cp`

TBD

Return value

If no error occurs, the **inet_addr** function returns an unsigned long value containing a suitable binary representation of the Internet address given.

If the string in the *cp* parameter does not contain a legitimate Internet address, for example if a portion of an "a.b.c.d" address exceeds 255, then **inet_addr** returns the value **INADDR_NONE**.

On Windows Server 2003and later if the string in the *cp* parameter is an empty string, then **inet_addr** returns the value **INADDR_NONE**. If **NULL** is passed in the *cp* parameter, then **inet_addr** returns the value **INADDR_NONE**.

On Windows XPand earlier if the string in the *cp* parameter is an empty string, then **inet_addr** returns the value **INADDR_ANY**. If **NULL** is passed in the *cp* parameter, then **inet_addr** returns the value **INADDR_NONE**.

Remarks

The **inet_addr** function interprets the character string specified by the *cp* parameter. This string represents a numeric Internet address expressed in the Internet standard "."' notation. The value returned is a number suitable for use as an Internet address. All Internet addresses are returned in IP's network order (bytes ordered from left to right). If you pass in " " (a space) to the **inet_addr** function, **inet_addr** returns zero.

On Windows Vista and later, the RtlIpv4StringToAddress function can be used to convert a string representation of an IPv4 address to a binary IPv4 address represented as an IN_ADDR structure. On Windows Vista and later, the RtlIpv6StringToAddress function can be used to convert a string representation of an IPv6 address to a binary IPv6 address represented as an **IN6_ADDR** structure.

Internet Addresses

Values specified using the "."' notation take one of the following forms:

a.b.c.d a.b.c a.b a

When four parts are specified, each is interpreted as a byte of data and assigned, from left to right, to the 4 bytes of an Internet address. When an Internet address is viewed as a 32-bit integer quantity on the Intel architecture,

the bytes referred to above appear as "d.c.b.a''. That is, the bytes on an Intel processor are ordered from right to left.

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

INTERNET ADDRESS VALUE	MEANING
"4.3.2.16"	Decimal
"004.003.002.020"	Octal
"0x4.0x3.0x2.0x10"	Hexadecimal
"4.003.002.0x10"	Mix

The **inet_addr** function supports the decimal, octal, hexadecimal, and mixed notations for the string passed in the *cp* parameter.

> **Note** The following notations are only used by Berkeley software, and nowhere else on the Internet. For compatibility with Berkeley software, the **inet_addr** function also supports the additional notations specified below.

When a three-part address is specified, the last part is interpreted as a 16-bit quantity and placed in the right-most 2 bytes of the network address. This makes the three-part address format convenient for specifying Class B network addresses as "128.net.host''

When a two-part address is specified, the last part is interpreted as a 24-bit quantity and placed in the right-most 3 bytes of the network address. This makes the two-part address format convenient for specifying Class A network addresses as "net.host''.

When only one part is given, the value is stored directly in the network address without any byte rearrangement.

Windows Phone 8: This function is supported for Windows Phone Store apps on Windows Phone 8 and later.

Windows 8.1 and **Windows Server 2012 R2**: This function is supported for Windows Store apps on Windows 8.1, Windows Server 2012 R2, and later.

Examples

The following code example shows how to use the **inet_addr** function.

```
#define WIN32_LEAN_AND_MEAN

#include <winsock2.h>
#include <stdio.h>
#include <windows.h>


// need link with Ws2_32.lib
#pragma comment(lib, "Ws2_32.lib")

int __cdecl main(int argc, char **argv)
{

    //-----------------------------------------
    // Declare and initialize variables
    WSADATA wsaData;
    int iResult;

    unsigned long ulAddr = INADDR_NONE;

    // Validate the parameters
    if (argc != 2) {
        printf("usage: %s <IPv4 address>\n", argv[0]);
        printf("  inetaddr converts a string containing an\n");
        printf("  IPv4 address in one of the supported formats\n");
        printf("  to a unsigned long representing an IN_ADDR\n");
        printf("      %s 192.168.16.34\n", argv[0]);
        return 1;
    }
    // Initialize Winsock
    iResult = WSAStartup(MAKEWORD(2, 2), &wsaData);
    if (iResult != 0) {
        printf("WSAStartup failed: %d\n", iResult);
        return 1;
    }

//--------------------------------
// Call inet_addr(). If the call succeeds,
// the result variable will hold a IN_ADDR
    ulAddr = inet_addr(argv[1]);
    if ( ulAddr == INADDR_NONE ) {
        printf("inet_addr failed and returned INADDR_NONE\n");
        WSACleanup();
        return 1;
    }

    if (ulAddr == INADDR_ANY) {
        printf("inet_addr failed and returned INADDR_ANY\n");
        WSACleanup();
        return 1;
    }

    printf("inet_addr returned success\n");

    // Here we could implement code to retrieve each address and
    // print out the hex bytes
    // for(i=0, ptr= (Char*) &ulAddr; i < 4; i++, ptr++) {

    WSACleanup();
    return 0;
}
```

Requirements

Minimum supported client	Windows 8.1, Windows Vista [desktop apps | UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps | UWP apps]
Target Platform	Windows
Header	winsock.h (include Winsock2.h, Winsock.h)
Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

IN6_ADDR

IN_ADDR

InetNtop

RtlIpv4AddressToString

RtlIpv4AddressToStringEx

RtlIpv4StringToAddress

RtlIpv4StringToAddressEx

RtlIpv6AddressToString

RtlIpv6AddressToStringEx

RtlIpv6StringToAddress

RtlIpv6StringToAddressEx

Winsock Functions

Winsock Reference

inet_ntoa

inet_ntoa function (winsock.h)

10/22/2022 • 2 minutes to read • Edit Online

The **inet_ntoa** function converts an (Ipv4) Internet network address into an ASCII string in Internet standard dotted-decimal format.

Syntax

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

Parameters

`in`

TBD

Return value

If no error occurs, **inet_ntoa** returns a character pointer to a static buffer containing the text address in standard "."' notation. Otherwise, it returns **NULL**.

Remarks

The **inet_ntoa** function takes an Internet address structure specified by the *in* parameter and returns a **NULL**-terminated ASCII string that represents the address in "." (dot) notation as in "192.168.16.0", an example of an IPv4 address in dotted-decimal notation. The string returned by **inet_ntoa** resides in memory that is allocated by Windows Sockets. The application should not make any assumptions about the way in which the memory is allocated. The string returned is guaranteed to be valid only until the next Windows Sockets function call is made within the same thread. Therefore, the data should be copied before another Windows Sockets call is made.

The WSAAddressToString function can be used to convert a sockaddr structure containing an IPv4 address to a string representation of an IPv4 address in Internet standard dotted-decimal notation. The advantage of the **WSAAddressToString** function is that it supports both IPv4 and IPv6 addresses. Another advantage of the **WSAAddressToString** function is that there are both ASCII and Unicode versions of this function.

On Windows Vista and later, the RtlIpv4AddressToString function can be used to convert an IPv4 address represented as an IN_ADDR structure to a string representation of an IPv4 address in Internet standard dotted-decimal notation. On Windows Vista and later, the RtlIpv6AddressToString function can be used to convert an IPv6 address represented as an **IN6_ADDR** structure to a string representation of an IPv6 address.

Windows Phone 8: This function is supported for Windows Phone Store apps on Windows Phone 8 and later.

Windows 8.1 and **Windows Server 2012 R2**: This function is supported for Windows Store apps on Windows 8.1, Windows Server 2012 R2, and later.

Requirements

Minimum supported client	Windows 8.1, Windows Vista [desktop apps | UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps | UWP apps]
Target Platform	Windows
Header	winsock.h (include Winsock2.h, Winsock.h)
Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

IN6_ADDR

InetNtop

RtlIpv4AddressToString

RtlIpv4AddressToStringEx

RtlIpv4StringToAddress

RtlIpv4StringToAddressEx

RtlIpv6AddressToString

RtlIpv6AddressToStringEx

RtlIpv6StringToAddress

RtlIpv6StringToAddressEx

SOCKADDR

WSAAddressToString

Winsock Functions

Winsock Reference

in_addr

inet_addr

ioctlsocket function (winsock.h)

10/22/2022 • 2 minutes to read • Edit Online

The **ioctlsocket** function controls the I/O mode of a socket.

Syntax

```
int ioctlsocket(
  [in]      SOCKET s,
  [in]      long   cmd,
  [in, out] u_long *argp
);
```

Parameters

`[in] s`

A descriptor identifying a socket.

`[in] cmd`

A command to perform on the socket *s*.

`[in, out] argp`

A pointer to a parameter for *cmd*.

Return value

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

ERROR CODE	MEANING
WSANOTINITIALISED	A successful WSAStartup call must occur before using this function.
WSAENETDOWN	The network subsystem has failed.
WSAEINPROGRESS	A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function.
WSAENOTSOCK	The descriptor *s* is not a socket.
WSAEFAULT	The *argp* parameter is not a valid part of the user address space.

Remarks

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

The WSAIoctl function is used to set or retrieve operating parameters associated with the socket, the transport protocol, or the communications subsystem.

The **WSAIoctl** function is more powerful than the **ioctlsocket** function and supports a large number of possible values for the operating parameters to set or retrieve.

Example Code

The following example demonstrates the use of the **ioctlsocket** function.

```
#include <winsock2.h>
#include <stdio.h>

#pragma comment(lib, "Ws2_32.lib")

void main()
{
//-------------------------
// Initialize Winsock
WSADATA wsaData;
int iResult;
u_long iMode = 0;

iResult = WSAStartup(MAKEWORD(2,2), &wsaData);
if (iResult != NO_ERROR)
  printf("Error at WSAStartup()\n");

//-------------------------
// Create a SOCKET object.
SOCKET m_socket;
m_socket = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
if (m_socket == INVALID_SOCKET) {
  printf("Error at socket(): %ld\n", WSAGetLastError());
  WSACleanup();
  return;
}

//-------------------------
// Set the socket I/O mode: In this case FIONBIO
// enables or disables the blocking mode for the
// socket based on the numerical value of iMode.
// If iMode = 0, blocking is enabled;
// If iMode != 0, non-blocking mode is enabled.

iResult = ioctlsocket(m_socket, FIONBIO, &iMode);
if (iResult != NO_ERROR)
  printf("ioctlsocket failed with error: %ld\n", iResult);


}
```

Compatibility

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

Windows Phone 8: This function is supported for Windows Phone Store apps on Windows Phone 8 and later.

Windows 8.1 and **Windows Server 2012 R2**: This function is supported for Windows Store apps on

Windows 8.1, Windows Server 2012 R2, and later.

Requirements

Minimum supported client	Windows 8.1, Windows Vista [desktop apps | UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps | UWP apps]
Target Platform	Windows
Header	winsock.h (include Winsock2.h)
Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

WSAAsyncSelect

WSAEventSelect

WSAIoctl

Winsock Functions

Winsock Reference

getsockopt

setsockopt

socket

LINGER structure (winsock.h)

10/22/2022 • 3 minutes to read • Edit Online

The **linger** structure maintains information about a specific socket that specifies how that socket should behave when data is queued to be sent and the closesocket function is called on the socket.

Syntax

```
typedef struct linger {
  u_short l_onoff;
  u_short l_linger;
} LINGER, *PLINGER, *LPLINGER;
```

Members

`l_onoff`

Type: **u_short**

Specifies whether a socket should remain open for a specified amount of time after a closesocket function call to enable queued data to be sent. This member can have one of the following values.

VALUE	MEANING
0	The socket will not remain open. This is the value set if the setsockopt function is called with the *optname* parameter set to **SO_DONTLINGER** and the *optval* parameter is zero. This value is also set if the setsockopt function is called with the *optname* parameter set to **SO_LINGER** and the **linger** structure passed in the *optval* parameter has the **l_onoff** member set to 0.
nonzero	The socket will remain open for a specified amount of time. This value is set if the setsockopt function is called with the *optname* parameter set to **SO_DONTLINGER** and the *optval* parameter is nonzero. This value is also set if the setsockopt function is called with the *optname* parameter set to **SO_LINGER** and the **linger** structure passed in the *optval* parameter has the **l_onoff** member set to a nonzero value.

`l_linger`

Type: **u_short**

The linger time in seconds. This member specifies how long to remain open after a closesocket function call to enable queued data to be sent. This member is only applicable if the **l_onoff** member of the **linger** structure is set to a nonzero value.

This value is set if the setsockopt function is called with the *optname* parameter set to **SO_LINGER**. The *optval* parameter passed to the **setsockopt** function must contain a **linger** structure that is copied to the internal **linger** structure maintained for the socket.

Remarks

The **l_onoff** member of the **linger** structure determines whether a socket should remain open for a specified amount of time after a closesocket function call to enable queued data to be sent. Somewhat confusing is that this member can be modified in two ways:

- Call the setsockopt function with the *optname* parameter set to **SO_DONTLINGER**. The *optval* parameter determines how the **l_onoff** member is modified.
- Call the setsockopt function with the *optname* parameter set to **SO_LINGER**. The *optval* parameter specifies how both the **l_onoff** and **l_linger** members are modified.

The **l_linger** member of the **linger** structure determines the amount of time, in seconds, a socket should remain open. This member is only applicable if the **l_onoff** member of the **linger** structure is nonzero.

To enable a socket to remain open, an application should set the **l_onoff** member to a nonzero value and set the **l_linger** member to the desired time-out in seconds. To disable a socket from remaining open, an application only needs to set the **l_onoff** member of the **linger** structure to zero.

If an application calls the setsockopt function with the *optname* parameter set to **SO_DONTLINGER** to set the **l_onoff** member to a nonzero value, the value for the **l_linger** member is not specified. In this case, the time-out used is implementation dependent. If a previous time-out has been established for a socket (by enabling SO_LINGER), this time-out value should be reinstated by the service provider.

Note that enabling a nonzero timeout on a nonblocking socket is not recommended.

The getsockopt function can be called with the *optname* parameter set to **SO_LINGER** to retrieve the current value of the **linger** structure associated with a socket.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	winsock.h (include Winsock2.h)

See also

Graceful Shutdown, Linger Options, and Socket Closure

closesocket

getsockopt

setsockopt

ntohl function (winsock.h)

10/22/2022 • 2 minutes to read • Edit Online

The **ntohl** function converts a **u_long** from TCP/IP network order to host byte order (which is little-endian on Intel processors).

Syntax

```
u_long ntohl(
  [in] u_long netlong
);
```

Parameters

`[in] netlong`

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

Return value

The **ntohl** function returns the value supplied in the *netlong* parameter with the byte order reversed. If *netlong* is already in host byte order, then this function will reverse it. It is up to the application to determine if the byte order must be reversed.

Remarks

The **ntohl** function takes a 32-bit number in TCP/IP network byte order (the AF_INET or AF_INET6 address family) and returns a 32-bit number in host byte order.

The **ntohl** function can be used to convert an IPv4 address in network byte order to the IPv4 address in host byte order. This function does not do any checking to determine if the *netlong* parameter is a valid IPv4 address.

The **ntohl** function does not require that the Winsock DLL has previously been loaded with a successful call to the WSAStartup function.

Windows 8.1 and **Windows Server 2012 R2**: This function is supported for Windows Store apps on Windows 8.1, Windows Server 2012 R2, and later.

Requirements

Minimum supported client	Windows 8.1, Windows Vista [desktop apps | UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps | UWP apps]
Target Platform	Windows
Header	winsock.h (include Winsock2.h)

Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

InetNtop

InetPton

WSAHtonl

WSAHtons

WSANtohl

WSANtohs

htond

htonf

htonl

htonll

htons

inet_addr

inet_ntoa

ntohd

ntohf

ntohs

ntohs function (winsock.h)

10/22/2022 • 2 minutes to read • Edit Online

The **ntohs** function converts a **u_short** from TCP/IP network byte order to host byte order (which is little-endian on Intel processors).

Syntax

```
u_short ntohs(
  [in] u_short netshort
);
```

Parameters

`[in] netshort`

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

Return value

The **ntohs** function returns the value in host byte order. If the *netshort* parameter is already in host byte order, then this function will reverse it. It is up to the application to determine if the byte order must be reversed.

Remarks

The **ntohs** function takes a 16-bit number in TCP/IP network byte order (the AF_INET or AF_INET6 address family) and returns a 16-bit number in host byte order.

The **ntohs** function can be used to convert an IP port number in network byte order to the IP port number in host byte order.

The **ntohs** function does not require that the Winsock DLL has previously been loaded with a successful call to the WSAStartup function.

Windows 8.1 and **Windows Server 2012 R2**: This function is supported for Windows Store apps on Windows 8.1, Windows Server 2012 R2, and later.

Requirements

Minimum supported client	Windows 8.1, Windows Vista [desktop apps | UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps | UWP apps]
Target Platform	Windows
Header	winsock.h (include Winsock2.h)
Library	Ws2_32.lib

DLL	Ws2_32.dll

See also

InetNtop

InetPton

WSAHtonl

WSAHtons

WSANtohl

WSANtohs

htond

htonf

htonl

htonll

htons

inet_addr

inet_ntoa

ntohd

ntohf

ntohl

ntohll

PROTOENT structure (winsock.h)

10/22/2022 • 2 minutes to read • Edit Online

The **protoent** structure contains the name and protocol numbers that correspond to a given protocol name. Applications must never attempt to modify this structure or to free any of its components. Furthermore, only one copy of this structure is allocated per thread, and therefore, the application should copy any information it needs before issuing any other Windows Sockets function calls.

Syntax

```
typedef struct protoent {
  char  *p_name;
  char  **p_aliases;
  short p_proto;
} PROTOENT, *PPROTOENT, *LPPROTOENT;
```

Members

`p_name`

Official name of the protocol.

`p_aliases`

Null-terminated array of alternate names.

`p_proto`

Protocol number, in host byte order.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	winsock.h (include Winsock2.h)

recv function (winsock.h)

10/22/2022 • 7 minutes to read • Edit Online

The **recv** function receives data from a connected socket or a bound connectionless socket.

Syntax

```
int recv(
  [in]  SOCKET s,
  [out] char   *buf,
  [in]  int    len,
  [in]  int    flags
);
```

Parameters

`[in] s`

The descriptor that identifies a connected socket.

`[out] buf`

A pointer to the buffer to receive the incoming data.

`[in] len`

The length, in bytes, of the buffer pointed to by the *buf* parameter.

`[in] flags`

A set of flags that influences the behavior of this function. See remarks below. See the Remarks section for details on the possible value for this parameter.

Return value

If no error occurs, **recv** returns the number of bytes received and the buffer pointed to by the *buf* parameter will contain this data received. If the connection has been gracefully closed, the return value is zero.

Otherwise, a value of SOCKET_ERROR is returned, and a specific error code can be retrieved by calling WSAGetLastError.

ERROR CODE	MEANING
WSANOTINITIALISED	A successful WSAStartup call must occur before using this function.
WSAENETDOWN	The network subsystem has failed.
WSAEFAULT	The *buf* parameter is not completely contained in a valid part of the user address space.

WSAENOTCONN	The socket is not connected.
WSAEINTR	The (blocking) call was canceled through WSACancelBlockingCall.
WSAEINPROGRESS	A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function.
WSAENETRESET	For a connection-oriented socket, this error indicates that the connection has been broken due to *keep-alive* activity that detected a failure while the operation was in progress. For a datagram socket, this error indicates that the time to live has expired.
WSAENOTSOCK	The descriptor is not a socket.
WSAEOPNOTSUPP	MSG_OOB was specified, but the socket is not stream-style such as type SOCK_STREAM, OOB data is not supported in the communication domain associated with this socket, or the socket is unidirectional and supports only send operations.
WSAESHUTDOWN	The socket has been shut down; it is not possible to receive on a socket after shutdown has been invoked with **how** set to SD_RECEIVE or SD_BOTH.
WSAEWOULDBLOCK	The socket is marked as nonblocking and the receive operation would block.
WSAEMSGSIZE	The message was too large to fit into the specified buffer and was truncated.
WSAEINVAL	The socket has not been bound with bind, or an unknown flag was specified, or MSG_OOB was specified for a socket with SO_OOBINLINE enabled or (for byte stream sockets only) *len* was zero or negative.
WSAECONNABORTED	The virtual circuit was terminated due to a time-out or other failure. The application should close the socket as it is no longer usable.
WSAETIMEDOUT	The connection has been dropped because of a network failure or because the peer system failed to respond.
WSAECONNRESET	The virtual circuit was reset by the remote side executing a hard or abortive close. The application should close the socket as it is no longer usable. On a UDP-datagram socket, this error would indicate that a previous send operation resulted in an ICMP "Port Unreachable" message.

Remarks

The **recv** function is used to read incoming data on connection-oriented sockets, or connectionless sockets.

When using a connection-oriented protocol, the sockets must be connected before calling **recv**. When using a connectionless protocol, the sockets must be bound before calling **recv**.

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

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

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

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

If the datagram or message is larger than the buffer specified, the buffer is filled with the first part of the datagram, and **recv** generates the error WSAEMSGSIZE. For unreliable protocols (for example, UDP) the excess data is lost; for reliable protocols, the data is retained by the service provider until it is successfully read by calling **recv** with a large enough buffer.

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

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

The *flags* parameter can be used to influence the behavior of the function invocation beyond the options specified for the associated socket. The semantics of this function are determined by the socket options and the *flags* parameter. The possible value of *flags* parameter is constructed by using the bitwise OR operator with any of the following values.

VALUE	MEANING
MSG_PEEK	Peeks at the incoming data. The data is copied into the buffer, but is not removed from the input queue.
MSG_OOB	Processes Out Of Band (OOB) data.

MSG_WAITALL	The receive request will complete only when one of the following events occurs: • The buffer supplied by the caller is completely full. • The connection has been closed. • The request has been canceled or an error occurred. Note that if the underlying transport does not support MSG_WAITALL, or if the socket is in a non-blocking mode, then this call will fail with **WSAEOPNOTSUPP**. Also, if MSG_WAITALL is specified along with MSG_OOB, MSG_PEEK, or MSG_PARTIAL, then this call will fail with **WSAEOPNOTSUPP**. This flag is not supported on datagram sockets or message-oriented sockets.

> **Note** When issuing a blocking Winsock call such as **recv**, Winsock may need to wait for a network event before the call can complete. Winsock performs an alertable wait in this situation, which can be interrupted by an asynchronous procedure call (APC) scheduled on the same thread. Issuing another blocking Winsock call inside an APC that interrupted an ongoing blocking Winsock call on the same thread will lead to undefined behavior, and must never be attempted by Winsock clients.

Example Code

The following code example shows the use of the **recv** function.

```
#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")

#define DEFAULT_BUFLEN 512
#define DEFAULT_PORT "27015"

int __cdecl main() {

    //-----------------------
    // Declare and initialize variables.
    WSADATA wsaData;
    int iResult;

    SOCKET ConnectSocket = INVALID_SOCKET;
    struct sockaddr_in clientService;

    char *sendbuf = "this is a test";
    char recvbuf[DEFAULT_BUFLEN];
    int recvbuflen = DEFAULT_BUFLEN;

    //-----------------------
    // Initialize Winsock
    iResult = WSAStartup(MAKEWORD(2,2), &wsaData);
    if (iResult != NO_ERROR) {
      printf("WSAStartup failed: %d\n", iResult);
      return 1;
    }

    //-----------------------
    // Create a SOCKET for connecting to server
    ConnectSocket = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
```

```
    if (ConnectSocket == INVALID_SOCKET) {
        printf("Error at socket(): %ld\n", WSAGetLastError() );
        WSACleanup();
        return 1;
    }

    //----------------------
    // The sockaddr_in structure specifies the address family,
    // IP address, and port of the server to be connected to.
    clientService.sin_family = AF_INET;
    clientService.sin_addr.s_addr = inet_addr( "127.0.0.1" );
    clientService.sin_port = htons( 27015 );

    //----------------------
    // Connect to server.
    iResult = connect( ConnectSocket, (SOCKADDR*) &clientService, sizeof(clientService) );
    if ( iResult == SOCKET_ERROR) {
        closesocket (ConnectSocket);
        printf("Unable to connect to server: %ld\n", WSAGetLastError());
        WSACleanup();
        return 1;
    }

    // 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 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;
    }

    // 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: %d\n", WSAGetLastError());

    } while( iResult > 0 );

    // cleanup
    closesocket(ConnectSocket);
    WSACleanup();

    return 0;
}
```

Example Code

For more information, and another example of the **recv** function, see Getting Started With Winsock.

Windows Phone 8: This function is supported for Windows Phone Store apps on Windows Phone 8 and later.

Windows 8.1 and **Windows Server 2012 R2**: This function is supported for Windows Store apps on Windows 8.1, Windows Server 2012 R2, and later.

Requirements

Minimum supported client	Windows 8.1, Windows Vista [desktop apps | UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps | UWP apps]
Target Platform	Windows
Header	winsock.h (include Winsock2.h)
Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

WSAAsyncSelect

WSARecv

WSARecvEx

Winsock Functions

Winsock Reference

recvfrom

select

send

socket

recvfrom function (winsock.h)

10/22/2022 • 7 minutes to read • Edit Online

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

Syntax

```
int recvfrom(
  [in]                SOCKET   s,
  [out]               char     *buf,
  [in]                int      len,
  [in]                int      flags,
  [out]               sockaddr *from,
  [in, out, optional] int      *fromlen
);
```

Parameters

`[in] s`

A descriptor identifying a bound socket.

`[out] buf`

A buffer for the incoming data.

`[in] len`

The length, in bytes, of the buffer pointed to by the *buf* parameter.

`[in] flags`

A set of options that modify the behavior of the function call beyond the options specified for the associated socket. See the Remarks below for more details.

`[out] from`

An optional pointer to a buffer in a sockaddr structure that will hold the source address upon return.

`[in, out, optional] fromlen`

An optional pointer to the size, in bytes, of the buffer pointed to by the *from* parameter.

Return value

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

ERROR CODE	MEANING
WSANOTINITIALISED	A successful WSAStartup call must occur before using this function.

WSAENETDOWN	The network subsystem has failed.
WSAEFAULT	The buffer pointed to by the *buf* or *from* parameters are not in the user address space, or the *fromlen* parameter is too small to accommodate the source address of the peer address.
WSAEINTR	The (blocking) call was canceled through WSACancelBlockingCall.
WSAEINPROGRESS	A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function.
WSAEINVAL	The socket has not been bound with bind, or an unknown flag was specified, or MSG_OOB was specified for a socket with SO_OOBINLINE enabled, or (for byte stream-style sockets only) *len* was zero or negative.
WSAEISCONN	The socket is connected. This function is not permitted with a connected socket, whether the socket is connection oriented or connectionless.
WSAENETRESET	For a datagram socket, this error indicates that the time to live has expired.
WSAENOTSOCK	The descriptor in the *s* parameter is not a socket.
WSAEOPNOTSUPP	MSG_OOB was specified, but the socket is not stream-style such as type SOCK_STREAM, OOB data is not supported in the communication domain associated with this socket, or the socket is unidirectional and supports only send operations.
WSAESHUTDOWN	The socket has been shut down; it is not possible to recvfrom on a socket after shutdown has been invoked with *how* set to SD_RECEIVE or SD_BOTH.
WSAEWOULDBLOCK	The socket is marked as nonblocking and the recvfrom operation would block.
WSAEMSGSIZE	The message was too large to fit into the buffer pointed to by the *buf* parameter and was truncated.
WSAETIMEDOUT	The connection has been dropped, because of a network failure or because the system on the other end went down without notice.
WSAECONNRESET	The virtual circuit was reset by the remote side executing a hard or abortive close. The application should close the socket; it is no longer usable. On a UDP-datagram socket this error indicates a previous send operation resulted in an ICMP *Port Unreachable* message.

Remarks

The **recvfrom** function reads incoming data on both connected and unconnected sockets and captures the address from which the data was sent. This function is typically used with connectionless sockets. The local address of the socket must be known. For server applications, this is usually done explicitly through bind. Explicit binding is discouraged for client applications. For client applications using this function, the socket can become bound implicitly to a local address through sendto, WSASendTo, or WSAJoinLeaf.

For stream-oriented sockets such as those of type SOCK_STREAM, a call to **recvfrom** returns as much information as is currently available—up to the size of the buffer specified. If the socket has been configured for inline reception of OOB data (socket option SO_OOBINLINE) and OOB data is yet unread, only OOB data will be returned. The application can use the ioctlsocket or WSAIoctl**SIOCATMARK** command to determine whether any more OOB data remains to be read. The *from* and *fromlen* parameters are ignored for connection-oriented sockets.

For message-oriented sockets, data is extracted from the first enqueued message, up to the size of the buffer specified. If the datagram or message is larger than the buffer specified, the buffer is filled with the first part of the datagram, and **recvfrom** generates the error WSAEMSGSIZE. For unreliable protocols (for example, UDP) the excess data is lost. For UDP if the packet received contains no data (empty), the return value from the **recvfrom** function function is zero.

If the *from* parameter is nonzero and the socket is not connection oriented, (type SOCK_DGRAM for example), the network address of the peer that sent the data is copied to the corresponding sockaddr structure. The value pointed to by *fromlen* is initialized to the size of this structure and is modified, on return, to indicate the actual size of the address stored in the **sockaddr** structure.

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

If the socket is connection oriented and the remote side has shut down the connection gracefully, the call to **recvfrom** will complete immediately with zero bytes received. If the connection has been reset **recvfrom** will fail with the error WSAECONNRESET.

The *flags* parameter can be used to influence the behavior of the function invocation beyond the options specified for the associated socket. The semantics of this function are determined by the socket options and the *flags* parameter. The latter is constructed by using the bitwise OR operator with any of the following values.

VALUE	MEANING
MSG_PEEK	Peeks at the incoming data. The data is copied into the buffer but is not removed from the input queue.
MSG_OOB	Processes Out Of Band (OOB) data.

> **Note** When issuing a blocking Winsock call such as **recvfrom**, Winsock may need to wait for a network event before the call can complete. Winsock performs an alertable wait in this situation, which can be interrupted by an asynchronous procedure call (APC) scheduled on the same thread. Issuing another blocking Winsock call inside an APC that interrupted an ongoing blocking Winsock call on the same thread will lead to undefined behavior, and must never be attempted by Winsock clients.

Example Code

The following example demonstrates the use of the **recvfrom** function.

```
#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 main()
{

    int iResult = 0;

    WSADATA wsaData;

    SOCKET RecvSocket;
    struct sockaddr_in RecvAddr;

    unsigned short Port = 27015;

    char RecvBuf[1024];
    int BufLen = 1024;

    struct sockaddr_in SenderAddr;
    int SenderAddrSize = sizeof (SenderAddr);

    //-----------------------------------------------
    // Initialize Winsock
    iResult = WSAStartup(MAKEWORD(2, 2), &wsaData);
    if (iResult != NO_ERROR) {
        wprintf(L"WSAStartup failed with error %d\n", iResult);
        return 1;
    }
    //-----------------------------------------------
    // Create a receiver socket to receive datagrams
    RecvSocket = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);
    if (RecvSocket == INVALID_SOCKET) {
        wprintf(L"socket failed with error %d\n", WSAGetLastError());
        return 1;
    }
    //-----------------------------------------------
    // Bind the socket to any address and the specified port.
    RecvAddr.sin_family = AF_INET;
    RecvAddr.sin_port = htons(Port);
    RecvAddr.sin_addr.s_addr = htonl(INADDR_ANY);

    iResult = bind(RecvSocket, (SOCKADDR *) & RecvAddr, sizeof (RecvAddr));
    if (iResult != 0) {
        wprintf(L"bind failed with error %d\n", WSAGetLastError());
        return 1;
    }
    //-----------------------------------------------
    // Call the recvfrom function to receive datagrams
    // on the bound socket.
    wprintf(L"Receiving datagrams...\n");
    iResult = recvfrom(RecvSocket,
                       RecvBuf, BufLen, 0, (SOCKADDR *) & SenderAddr, &SenderAddrSize);
    if (iResult == SOCKET_ERROR) {
        wprintf(L"recvfrom failed with error %d\n", WSAGetLastError());
    }
```

```
    //-----------------------------------------------
    // Close the socket when finished receiving datagrams
    wprintf(L"Finished receiving. Closing socket.\n");
    iResult = closesocket(RecvSocket);
    if (iResult == SOCKET_ERROR) {
        wprintf(L"closesocket failed with error %d\n", WSAGetLastError());
        return 1;
    }

    //-----------------------------------------------
    // Clean up and exit.
    wprintf(L"Exiting.\n");
    WSACleanup();
    return 0;
}
```

Windows Phone 8: This function is supported for Windows Phone Store apps on Windows Phone 8 and later.

Windows 8.1 and **Windows Server 2012 R2**: This function is supported for Windows Store apps on Windows 8.1, Windows Server 2012 R2, and later.

Requirements

Minimum supported client	Windows 8.1, Windows Vista [desktop apps | UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps | UWP apps]
Target Platform	Windows
Header	winsock.h (include Winsock2.h)
Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

WSAAsyncSelect

WSAEventSelect

Winsock Functions

Winsock Reference

recv

send

sockaddr

socket

sendto function (winsock.h)

10/22/2022 • 8 minutes to read • Edit Online

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

Syntax

```
int sendto(
  [in] SOCKET         s,
  [in] const char     *buf,
  [in] int            len,
  [in] int            flags,
  [in] const sockaddr *to,
  [in] int            tolen
);
```

Parameters

`[in] s`

A descriptor identifying a (possibly connected) socket.

`[in] buf`

A pointer to a buffer containing the data to be transmitted.

`[in] len`

The length, in bytes, of the data pointed to by the *buf* parameter.

`[in] flags`

A set of flags that specify the way in which the call is made.

`[in] to`

An optional pointer to a sockaddr structure that contains the address of the target socket.

`[in] tolen`

The size, in bytes, of the address pointed to by the *to* parameter.

Return value

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

ERROR CODE	MEANING
WSANOTINITIALISED	A successful WSAStartup call must occur before using this function.

WSAENETDOWN	The network subsystem has failed.
WSAEACCES	The requested address is a broadcast address, but the appropriate flag was not set. Call setsockopt with the SO_BROADCAST parameter to allow the use of the broadcast address.
WSAEINVAL	An unknown flag was specified, or MSG_OOB was specified for a socket with SO_OOBINLINE enabled.
WSAEINTR	A blocking Windows Sockets 1.1 call was canceled through WSACancelBlockingCall.
WSAEINPROGRESS	A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function.
WSAEFAULT	The *buf* or *to* parameters are not part of the user address space, or the *tolen* parameter is too small.
WSAENETRESET	The connection has been broken due to keep-alive activity detecting a failure while the operation was in progress.
WSAENOBUFS	No buffer space is available.
WSAENOTCONN	The socket is not connected (connection-oriented sockets only).
WSAENOTSOCK	The descriptor is not a socket.
WSAEOPNOTSUPP	MSG_OOB was specified, but the socket is not stream-style such as type SOCK_STREAM, OOB data is not supported in the communication domain associated with this socket, or the socket is unidirectional and supports only receive operations.
WSAESHUTDOWN	The socket has been shut down; it is not possible to sendto on a socket after shutdown has been invoked with *how* set to SD_SEND or SD_BOTH.
WSAEWOULDBLOCK	The socket is marked as nonblocking and the requested operation would block.
WSAEMSGSIZE	The socket is message oriented, and the message is larger than the maximum supported by the underlying transport.
WSAEHOSTUNREACH	The remote host cannot be reached from this host at this time.

WSAECONNABORTED	The virtual circuit was terminated due to a time-out or other failure. The application should close the socket as it is no longer usable.
WSAECONNRESET	The virtual circuit was reset by the remote side executing a hard or abortive close. For UDP sockets, the remote host was unable to deliver a previously sent UDP datagram and responded with a "Port Unreachable" ICMP packet. The application should close the socket as it is no longer usable.
WSAEADDRNOTAVAIL	The remote address is not a valid address, for example, ADDR_ANY.
WSAEAFNOSUPPORT	Addresses in the specified family cannot be used with this socket.
WSAEDESTADDRREQ	A destination address is required.
WSAENETUNREACH	The network cannot be reached from this host at this time.
WSAEHOSTUNREACH	A socket operation was attempted to an unreachable host.
WSAETIMEDOUT	The connection has been dropped, because of a network failure or because the system on the other end went down without notice.

Remarks

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

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

> **Note** If a socket is opened, a setsockopt call is made, and then a **sendto** call is made, Windows Sockets performs an implicit **bind** function call.

If the socket is unbound, unique values are assigned to the local association by the system, and the socket is then marked as bound. If the socket is connected, the getsockname function can be used to determine the local IP address and port associated with the socket.

If the socket is not connected, the getsockname function can be used to determine the local port number associated with the socket but the IP address returned is set to the wildcard address for the given protocol (for example, INADDR_ANY or "0.0.0.0" for IPv4 and IN6ADDR_ANY_INIT or "::" for IPv6).

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

The **sendto** function is normally used on a connectionless socket to send a datagram to a specific peer socket identified by the *to* parameter. Even if the connectionless socket has been previously connected to a specific address, the *to* parameter overrides the destination address for that particular datagram only. On a connection-oriented socket, the *to* and *tolen* parameters are ignored, making **sendto** equivalent to send.

> **Note** When issuing a blocking Winsock call such as **sendto**, Winsock may need to wait for a network event before the call can complete. Winsock performs an alertable wait in this situation, which can be interrupted by an asynchronous procedure call (APC) scheduled on the same thread. Issuing another blocking Winsock call inside an APC that interrupted an ongoing blocking Winsock call on the same thread will lead to undefined behavior, and must never be attempted by Winsock clients.

Example Code

The following example demonstrates the use of the **sendto** function.

```
#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 main()
{

    int iResult;
    WSADATA wsaData;

    SOCKET SendSocket = INVALID_SOCKET;
    sockaddr_in RecvAddr;

    unsigned short Port = 27015;

    char SendBuf[1024];
    int BufLen = 1024;

    //----------------------
    // Initialize Winsock
    iResult = WSAStartup(MAKEWORD(2, 2), &wsaData);
    if (iResult != NO_ERROR) {
        wprintf(L"WSAStartup failed with error: %d\n", iResult);
        return 1;
    }

    //-----------------------------------------------
    // Create a socket for sending data
    SendSocket = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);
    if (SendSocket == INVALID_SOCKET) {
        wprintf(L"socket failed with error: %ld\n", WSAGetLastError());
        WSACleanup();
        return 1;
    }
    //-----------------------------------------------
    // Set up the RecvAddr structure with the IP address of
    // the receiver (in this example case "192.168.1.1")
    // and the specified port number.
```

```
    RecvAddr.sin_family = AF_INET;
    RecvAddr.sin_port = htons(Port);
    RecvAddr.sin_addr.s_addr = inet_addr("192.168.1.1");

    //---------------------------------------------
    // Send a datagram to the receiver
    wprintf(L"Sending a datagram to the receiver...\n");
    iResult = sendto(SendSocket,
                     SendBuf, BufLen, 0, (SOCKADDR *) & RecvAddr, sizeof (RecvAddr));
    if (iResult == SOCKET_ERROR) {
        wprintf(L"sendto failed with error: %d\n", WSAGetLastError());
        closesocket(SendSocket);
        WSACleanup();
        return 1;
    }
    //---------------------------------------------
    // When the application is finished sending, close the socket.
    wprintf(L"Finished sending. Closing socket.\n");
    iResult = closesocket(SendSocket);
    if (iResult == SOCKET_ERROR) {
        wprintf(L"closesocket failed with error: %d\n", WSAGetLastError());
        WSACleanup();
        return 1;
    }
    //---------------------------------------------
    // Clean up and quit.
    wprintf(L"Exiting.\n");
    WSACleanup();
    return 0;
}
```

For Sockets Using IP (Version 4)

To send a broadcast (on a SOCK_DGRAM only), the address pointed to by the *to* parameter can be constructed to contain the special IPv4 address INADDR_BROADCAST (defined in Winsock2.h), together with the intended port number. If the address pointed to by the *to* parameter contains the INADDR_BROADCAST address and intended port, then the broadcast will be sent out on all interfaces to that port.

If the broadcast should be sent out only on a specific interface, then the address pointed to by the *to* parameter should contain the subnet broadcast address for the interface and the intended port. For example, an IPv4 network address of 192.168.1.0 with a subnet mask of 255.255.255.0 would use a subnet broadcast address of 192.168.1.255.

It is generally inadvisable for a broadcast datagram to exceed the size at which fragmentation can occur, which implies that the data portion of the datagram (excluding headers) should not exceed 512 bytes.

If no buffer space is available within the transport system to hold the data to be transmitted, **sendto** will block unless the socket has been placed in a nonblocking mode. On nonblocking, stream oriented sockets, the number of bytes written can be between 1 and the requested length, depending on buffer availability on both the client and server systems. The select, WSAAsyncSelect or WSAEventSelect function can be used to determine when it is possible to send more data.

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

The *flags* parameter can be used to influence the behavior of the function invocation beyond the options specified for the associated socket. The semantics of this function are determined by the socket options and the *flags* parameter. The latter is constructed by using the bitwise OR operator with any of the following values.

VALUE	MEANING

MSG_DONTROUTE	Specifies that the data should not be subject to routing. A Windows Sockets service provider can choose to ignore this flag.
MSG_OOB	Sends OOB data (stream-style socket such as SOCK_STREAM only).

Windows Phone 8: This function is supported for Windows Phone Store apps on Windows Phone 8 and later.

Windows 8.1 and **Windows Server 2012 R2**: This function is supported for Windows Store apps on Windows 8.1, Windows Server 2012 R2, and later.

Requirements

Minimum supported client	Windows 8.1, Windows Vista [desktop apps | UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps | UWP apps]
Target Platform	Windows
Header	winsock.h (include Winsock2.h)
Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

WSAAsyncSelect

WSAEventSelect

Winsock Functions

Winsock Reference

recv

recvfrom

select

send

socket

SERVENT structure (winsock.h)

10/22/2022 • 2 minutes to read • Edit Online

The **servent** structure is used to store or return the name and service number for a given service name.

Syntax

```
typedef struct servent {
  char  *s_name;
  char  **s_aliases;
#if ...
  char  *s_proto;
#if ...
  short s_port;
#else
  short s_port;
#endif
#else
  char  *s_proto;
#endif
} SERVENT, *PSERVENT, *LPSERVENT;
```

Members

`s_name`

The official name of the service.

`s_aliases`

A **NULL**-terminated array of alternate names.

`s_proto`

The name of the protocol to use when contacting the service.

`s_port`

The port number at which the service can be contacted. Port numbers are returned in network byte order.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	winsock.h (include Winsock2.h)

See also

getservbyname

setsockopt function (winsock.h)

10/22/2022 • 10 minutes to read • Edit Online

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

Syntax

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

Parameters

`[in] s`

A descriptor that identifies a socket.

`[in] level`

The level at which the option is defined (for example, SOL_SOCKET).

`[in] optname`

The socket option for which the value is to be set (for example, SO_BROADCAST). The *optname* parameter must be a socket option defined within the specified *level*, or behavior is undefined.

`[in] optval`

A pointer to the buffer in which the value for the requested option is specified.

`[in] optlen`

The size, in bytes, of the buffer pointed to by the *optval* parameter.

Return value

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

ERROR CODE	MEANING
WSANOTINITIALISED	A successful WSAStartup call must occur before using this function.
WSAENETDOWN	The network subsystem has failed.

WSAEFAULT	The buffer pointed to by the *optval* parameter is not in a valid part of the process address space or the *optlen* parameter is too small.
WSAEINPROGRESS	A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function.
WSAEINVAL	The *level* parameter is not valid, or the information in the buffer pointed to by the *optval* parameter is not valid.
WSAENETRESET	The connection has timed out when SO_KEEPALIVE is set.
WSAENOPROTOOPT	The option is unknown or unsupported for the specified provider or socket (see SO_GROUP_PRIORITY limitations).
WSAENOTCONN	The connection has been reset when SO_KEEPALIVE is set.
WSAENOTSOCK	The descriptor is not a socket.

Remarks

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

> **Note** If the **setsockopt** function is called before the bind function, TCP/IP options will not be checked by using TCP/IP until the **bind** occurs. In this case, the **setsockopt** function call will always succeed, but the **bind** function call can fail because of an early **setsockopt** call failing.

> **Note** If a socket is opened, a **setsockopt** call is made, and then a sendto call is made, Windows Sockets performs an implicit bind function call.

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

The following tables list some of the common options supported by the **setsockopt** function. The Type column identifies the type of data addressed by *optval* parameter. The Description column provides some basic information about the socket option. For more complete lists of socket options and more detailed information (default values, for example), see the detailed topics under Socket Options.

level = SOL_SOCKET

VALUE	TYPE	DESCRIPTION
SO_BROADCAST	BOOL	Configures a socket for sending broadcast data.
SO_CONDITIONAL_ACCEPT	BOOL	Enables incoming connections are to be accepted or rejected by the application, not by the protocol stack.
SO_DEBUG	BOOL	Enables debug output. Microsoft providers currently do not output any debug information.
SO_DONTLINGER	BOOL	Does not block close waiting for unsent data to be sent. Setting this option is equivalent to setting SO_LINGER with **l_onoff** set to zero.
SO_DONTROUTE	BOOL	Sets whether outgoing data should be sent on interface the socket is bound to and not a routed on some other interface. This option is not supported on ATM sockets (results in an error).
SO_GROUP_PRIORITY	int	Reserved.
SO_KEEPALIVE	BOOL	Enables sending keep-alive packets for a socket connection. Not supported on ATM sockets (results in an error).
SO_LINGER	LINGER	Lingers on close if unsent data is present.
SO_OOBINLINE	BOOL	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. For a discussion of this topic, see Protocol Independent Out-Of-band Data.
SO_RCVBUF	int	Specifies the total per-socket buffer space reserved for receives.
SO_REUSEADDR	BOOL	Allows the socket to be bound to an address that is already in use. For more information, see bind. Not applicable on ATM sockets.
SO_EXCLUSIVEADDRUSE	BOOL	Enables a socket to be bound for exclusive access. Does not require administrative privilege.
SO_RCVTIMEO	DWORD	Sets the timeout, in milliseconds, for blocking receive calls.
SO_SNDBUF	int	Specifies the total per-socket buffer space reserved for sends.

SO_SNDTIMEO	DWORD	The timeout, in milliseconds, for blocking send calls.
SO_UPDATE_ACCEPT_CONTEXT	int	Updates the accepting socket with the context of the listening socket.
PVD_CONFIG	Service Provider Dependent	This object stores the configuration information for the service provider associated with socket *s*. The exact format of this data structure is service provider specific.

For more complete and detailed information about socket options for *level* = **SOL_SOCKET**, see SOL_SOCKET Socket Options.

level = **IPPROTO_TCP**

See **TCP_NODELAY** in IPPROTO_TCP socket options. Also see that topic for more complete and detailed information about socket options for *level* = **IPPROTO_TCP**.

level = **NSPROTO_IPX**

VALUE	TYPE	DESCRIPTION
IPX_PTYPE	int	Sets the IPX packet type.
IPX_FILTERPTYPE	int	Sets the receive filter packet type
IPX_STOPFILTERPTYPE	int	Stops filtering the filter type set with IPX_FILTERTYPE
IPX_DSTYPE	int	Sets the value of the data stream field in the SPX header on every packet sent.
IPX_EXTENDED_ADDRESS	BOOL	Sets whether extended addressing is enabled.
IPX_RECVHDR	BOOL	Sets whether the protocol header is sent up on all receive headers.
IPX_RECEIVE_BROADCAST	BOOL	Indicates broadcast packets are likely on the socket. Set to **TRUE** by default. Applications that do not use broadcasts should set this to **FALSE** for better system performance.
IPX_IMMEDIATESPXACK	BOOL	Directs SPX connections not to delay before sending an ACK. Applications without back-and-forth traffic should set this to **TRUE** to increase performance.

For more complete and detailed information about socket options for *level* = **NSPROTO_IPX**, see NSPROTO_IPX Socket Options.

BSD options not supported for **setsockopt** are shown in the following table.

VALUE	TYPE	DESCRIPTION
SO_ACCEPTCONN	BOOL	Returns whether a socket is in listening mode. This option is only Valid for connection-oriented protocols. This socket option is not supported for the setting.
SO_RCVLOWAT	int	A socket option from BSD UNIX included for backward compatibility. This option sets the minimum number of bytes to process for socket input operations.
SO_SNDLOWAT	int	A socket option from BSD UNIX included for backward compatibility. This option sets the minimum number of bytes to process for socket output operations.
SO_TYPE	int	Returns the socket type for the given socket (SOCK_STREAM or SOCK_DGRAM, for example This socket option is not supported for the setting the socket type.

Note When issuing a blocking Winsock call such as **setsockopt**, Winsock may need to wait for a network event before the call can complete. Winsock performs an alertable wait in this situation, which can be interrupted by an asynchronous procedure call (APC) scheduled on the same thread. Issuing another blocking Winsock call inside an APC that interrupted an ongoing blocking Winsock call on the same thread will lead to undefined behavior, and must never be attempted by Winsock clients.

Example Code

The following example demonstrates the **setsockopt** function.

```
#ifndef UNICODE
#define UNICODE
#endif

#ifndef WIN32_LEAN_AND_MEAN
#define WIN32_LEAN_AND_MEAN
#endif

#include <winsock2.h>
#include <Ws2tcpip.h>
#include <stdio.h>

// Link with ws2_32.lib
#pragma comment(lib, "Ws2_32.lib")

int main()
{

    //-----------------------------------------
    // Declare variables
    WSADATA wsaData;
```

```
    SOCKET ListenSocket;
    sockaddr_in service;

    int iResult = 0;

    BOOL bOptVal = FALSE;
    int bOptLen = sizeof (BOOL);

    int iOptVal = 0;
    int iOptLen = sizeof (int);

    //-----------------------------------------
    // Initialize Winsock
    iResult = WSAStartup(MAKEWORD(2, 2), &wsaData);
    if (iResult != NO_ERROR) {
        wprintf(L"Error at WSAStartup()\n");
        return 1;
    }
    //-----------------------------------------
    // Create a listening socket
    ListenSocket = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
    if (ListenSocket == INVALID_SOCKET) {
        wprintf(L"socket function failed with error: %u\n", WSAGetLastError());
        WSACleanup();
        return 1;
    }
    //-----------------------------------------
    // Bind the socket to the local IP address
    // and port 27015
    hostent *thisHost;
    char *ip;
    u_short port;
    port = 27015;
    thisHost = gethostbyname("");
    ip = inet_ntoa(*(struct in_addr *) *thisHost->h_addr_list);

    service.sin_family = AF_INET;
    service.sin_addr.s_addr = inet_addr(ip);
    service.sin_port = htons(port);

    iResult = bind(ListenSocket, (SOCKADDR *) & service, sizeof (service));
    if (iResult == SOCKET_ERROR) {
        wprintf(L"bind failed with error %u\n", WSAGetLastError());
        closesocket(ListenSocket);
        WSACleanup();
        return 1;
    }
    //-----------------------------------------
    // Initialize variables and call setsockopt.
    // The SO_KEEPALIVE parameter is a socket option
    // that makes the socket send keepalive messages
    // on the session. The SO_KEEPALIVE socket option
    // requires a boolean value to be passed to the
    // setsockopt function. If TRUE, the socket is
    // configured to send keepalive messages, if FALSE
    // the socket configured to NOT send keepalive messages.
    // This section of code tests the setsockopt function
    // by checking the status of SO_KEEPALIVE on the socket
    // using the getsockopt function.

    bOptVal = TRUE;

    iResult = getsockopt(ListenSocket, SOL_SOCKET, SO_KEEPALIVE, (char *) &iOptVal, &iOptLen);
    if (iResult == SOCKET_ERROR) {
        wprintf(L"getsockopt for SO_KEEPALIVE failed with error: %u\n", WSAGetLastError());
    } else
        wprintf(L"SO_KEEPALIVE Value: %ld\n", iOptVal);

    iResult = setsockopt(ListenSocket, SOL_SOCKET, SO_KEEPALIVE, (char *) &bOptVal, bOptLen);
```

```
    if (iResult == SOCKET_ERROR) {
        wprintf(L"setsockopt for SO_KEEPALIVE failed with error: %u\n", WSAGetLastError());
    } else
        wprintf(L"Set SO_KEEPALIVE: ON\n");

    iResult = getsockopt(ListenSocket, SOL_SOCKET, SO_KEEPALIVE, (char *) &iOptVal, &iOptLen);
    if (iResult == SOCKET_ERROR) {
        wprintf(L"getsockopt for SO_KEEPALIVE failed with error: %u\n", WSAGetLastError());
    } else
        wprintf(L"SO_KEEPALIVE Value: %ld\n", iOptVal);

    closesocket(ListenSocket);
    WSACleanup();
    return 0;
}
```

Notes for IrDA Sockets

When developing applications using Windows sockets for IrDA, note the following:

- The Af_irda.h header file must be explicitly included.
- IrDA provides the following socket option:

VALUE	TYPE	MEANING
IRLMP_IAS_SET	*IAS_SET	Sets IAS attributes

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

IrDA provides an IAS database that stores IrDA-based information. Limited access to the IAS database is available through the Windows Sockets 2 interface, but such access is not normally used by applications, and exists primarily to support connections to non-Windows devices that are not compliant with the Windows Sockets 2 IrDA conventions.

The following structure, **IAS_SET**, is used with the IRLMP_IAS_SET setsockopt option to manage the local IAS database:

```
// #include <Af_irda.h> for this struct

typedef struct _IAS_SET {
    u_char      irdaClassName[IAS_MAX_CLASSNAME];
    char      irdaAttribName[IAS_MAX_ATTRIBNAME];
    u_long    irdaAttribType;
    union
    {
              LONG irdaAttribInt;
              struct
              {
                   u_long   Len;
                   u_char    OctetSeq[IAS_MAX_OCTET_STRING];
              } irdaAttribOctetSeq;
              struct
              {
                   u_long    Len;
                   u_long    CharSet;
                   u_char    UsrStr[IAS_MAX_USER_STRING];
              } irdaAttribUsrStr;
    } irdaAttribute;
} IAS_SET, *PIAS_SET, FAR *LPIASSET;
```

The following structure, **IAS_QUERY**, is used with the IRLMP_IAS_QUERY setsockopt option to query a peer's IAS database:

```
// #include <Af_irda.h> for this struct

typedef struct _WINDOWS_IAS_QUERY {
        u_char   irdaDeviceID[4];
        char     irdaClassName[IAS_MAX_CLASSNAME];
        char     irdaAttribName[IAS_MAX_ATTRIBNAME];
        u_long   irdaAttribType;
        union
        {
                  LONG    irdaAttribInt;
                  struct
                  {
                          u_long  Len;
                          u_char  OctetSeq[IAS_MAX_OCTET_STRING];
                  } irdaAttribOctetSeq;
                  struct
                  {
                          u_long  Len;
                          u_long  CharSet;
                          u_char  UsrStr[IAS_MAX_USER_STRING];
                  } irdaAttribUsrStr;
        } irdaAttribute;
} IAS_QUERY, *PIAS_QUERY, FAR *LPIASQUERY;
```

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

Windows Phone 8: This function is supported for Windows Phone Store apps on Windows Phone 8 and later.

Windows 8.1 and **Windows Server 2012 R2**: This function is supported for Windows Store apps on Windows 8.1, Windows Server 2012 R2, and later.

Requirements

Minimum supported client	Windows 8.1, Windows Vista [desktop apps | UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps | UWP apps]
Target Platform	Windows
Header	winsock.h (include Winsock2.h)
Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

IPPROTO_IP Socket Options

IPPROTO_IPV6 Socket Options

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

Socket Options

WSAAsyncSelect

WSAEventSelect

WSAIoctl

Winsock Functions

bind

getsockopt

ioctlsocket

socket

shutdown function (winsock.h)

10/22/2022 • 5 minutes to read • Edit Online

The **shutdown** function disables sends or receives on a socket.

Syntax

```
int shutdown(
  [in] SOCKET s,
  [in] int    how
);
```

Parameters

`[in] s`

A descriptor identifying a socket.

`[in] how`

A flag that describes what types of operation will no longer be allowed. Possible values for this flag are listed in the *Winsock2.h* header file.

VALUE	MEANING
SD_RECEIVE 0	Shutdown receive operations.
SD_SEND 1	Shutdown send operations.
SD_BOTH 2	Shutdown both send and receive operations.

Return value

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

ERROR CODE	MEANING
WSAECONNABORTED	The virtual circuit was terminated due to a time-out or other failure. The application should close the socket as it is no longer usable. This error applies only to a connection-oriented socket.

WSAECONNRESET	The virtual circuit was reset by the remote side executing a hard or abortive close. The application should close the socket as it is no longer usable. This error applies only to a connection-oriented socket.
WSAEINPROGRESS	A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function.
WSAEINVAL	The *how* parameter is not valid, or is not consistent with the socket type. For example, SD_SEND is used with a UNI_RECV socket type.
WSAENETDOWN	The network subsystem has failed.
WSAENOTCONN	The socket is not connected. This error applies only to a connection-oriented socket.
WSAENOTSOCK	**Note** The descriptor is not a socket.
WSANOTINITIALISED	A successful WSAStartup call must occur before using this function.

Remarks

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

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

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

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

The **shutdown** function does not close the socket. Any resources attached to the socket will not be freed until closesocket is invoked.

To assure that all data is sent and received on a connected socket before it is closed, an application should use **shutdown** to close connection before calling closesocket. One method to wait for notification that the remote end has sent all its data and initiated a graceful disconnect uses the WSAEventSelect function as follows :

1. Call WSAEventSelect to register for FD_CLOSE notification.
2. Call **shutdown** with *how*=SD_SEND.
3. When FD_CLOSE received, call the recv or WSARecv until the function completes with success and indicates that zero bytes were received. If SOCKET_ERROR is returned, then the graceful disconnect is not possible.
4. Call closesocket.

Another method to wait for notification that the remote end has sent all its data and initiated a graceful disconnect uses overlapped receive calls follows :

1. Call **shutdown** with *how*=SD_SEND.
2. Call recv or WSARecv until the function completes with success and indicates zero bytes were received. If SOCKET_ERROR is returned, then the graceful disconnect is not possible.
3. Call closesocket.

> **Note** The **shutdown** function does not block regardless of the SO_LINGER setting on the socket.

For more information, see the section on Graceful Shutdown, Linger Options, and Socket Closure.

Once the **shutdown** function is called to disable send, receive, or both, there is no method to re-enable send or receive for the existing socket connection.

An application should not rely on being able to reuse a socket after it has been shut down. In particular, a Windows Sockets provider is not required to support the use of connect on a socket that has been shut down.

If an application wants to reuse a socket, then the DisconnectEx function should be called with the *dwFlags* parameter set to **TF_REUSE_SOCKET** to close a connection on a socket and prepare the socket handle to be reused. When the **DisconnectEx** request completes, the socket handle can be passed to the AcceptEx or ConnectEx function.

If an application wants to reuse a socket, the TransmitFile or TransmitPackets functions can be called with the *dwFlags* parameter set with **TF_DISCONNECT** and **TF_REUSE_SOCKET** to disconnect after all the data has been queued for transmission and prepare the socket handle to be reused. When the **TransmitFile** request completes, the socket handle can be passed to the function call previously used to establish the connection, such as AcceptEx or ConnectEx. When the **TransmitPackets** function completes, the socket handle can be passed to the **AcceptEx** function.

> **Note** The socket level disconnect is subject to the behavior of the underlying transport. For example, a TCP socket may be subject to the TCP TIME_WAIT state, causing the DisconnectEx, TransmitFile, or TransmitPackets call to be delayed.

> **Note** When issuing a blocking Winsock call such as **shutdown**, Winsock may need to wait for a network event before the call can complete. Winsock performs an alertable wait in this situation, which can be interrupted by an asynchronous procedure call (APC) scheduled on the same thread. Issuing another blocking Winsock call inside an APC that interrupted an ongoing blocking Winsock call on the same thread will lead to undefined behavior, and must never be attempted by Winsock clients.

Notes for ATM

There are important issues associated with connection teardown when using Asynchronous Transfer Mode (ATM) and Windows Sockets 2. For more information about these important considerations, see the section titled Notes for ATM in the Remarks section of the closesocket function reference.

Windows Phone 8: This function is supported for Windows Phone Store apps on Windows Phone 8 and later.

Windows 8.1 and **Windows Server 2012 R2**: This function is supported for Windows Store apps on Windows 8.1, Windows Server 2012 R2, and later.

Requirements

Minimum supported client	Windows 8.1, Windows Vista [desktop apps | UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps | UWP apps]
Target Platform	Windows
Header	winsock.h (include Winsock2.h, Webhost.h)
Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

AcceptEx

ConnectEx

DisconnectEx

TransmitFile

TransmitPackets

WSAEventSelect

Winsock Functions

Winsock Reference

connect

socket

SOCKADDR structure (winsock.h)

10/22/2022 • 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.

Syntax

```
typedef struct sockaddr {
  u_short sa_family;
  char    sa_data[14];
} SOCKADDR, *PSOCKADDR, *LPSOCKADDR;
```

Members

`sa_family`

`sa_data[14]`

Requirements

Header	winsock.h (include Ws2ipdef.h)

See also

SOCKADDR_STORAGE

SOCKADDR_IN structure (winsock.h)

10/22/2022 • 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.

Syntax

```
typedef struct sockaddr_in {
  short          sin_family;
  u_short        sin_port;
  struct in_addr sin_addr;
  char           sin_zero[8];
} SOCKADDR_IN, *PSOCKADDR_IN, *LPSOCKADDR_IN;
```

Members

`sin_family`

`sin_port`

`sin_addr`

`sin_zero[8]`

Requirements

Header	winsock.h (include Ws2ipdef.h)

See also

SOCKADDR_STORAGE

TIMEVAL structure (winsock.h)

10/22/2022 • 2 minutes to read • Edit Online

The **timeval** structure is used to specify a time interval. It is associated with the Berkeley Software Distribution (BSD) *Time.h* header file.

Syntax

```
typedef struct timeval {
  long tv_sec;
  long tv_usec;
} TIMEVAL, *PTIMEVAL, *LPTIMEVAL;
```

Members

`tv_sec`

Time interval, in seconds.

`tv_usec`

Time interval, in microseconds. This value is used in combination with the **tv_sec** member to represent time interval values that are not a multiple of seconds.

Remarks

The **timeval** structure is used in Windows Sockets by the select function to specify the maximum time the function can take to complete. The time interval is a combination of the values in **tv_sec** and **tv_usec** members.

Several functions are added on Windows Vista and later that use the **timeval** structure. These functions include GetAddrInfoEx, SetAddrInfoEx, WSAConnectByList, and WSAConnectByName.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	winsock.h (include Winsock2.h)

See also

GetAddrInfoEx

SetAddrInfoEx

WSAConnectByList

WSAConnectByName

linger

select

TRANSMIT_FILE_BUFFERS structure (winsock.h)

10/22/2022 • 2 minutes to read • Edit Online

The **TRANSMIT_FILE_BUFFERS** structure specifies data to be transmitted before and after file data during a TransmitFile function file transfer operation.

Syntax

```
typedef struct _TRANSMIT_FILE_BUFFERS {
  PVOID Head;
  DWORD HeadLength;
  PVOID Tail;
  DWORD TailLength;
} TRANSMIT_FILE_BUFFERS, *PTRANSMIT_FILE_BUFFERS, *LPTRANSMIT_FILE_BUFFERS;
```

Members

`Head`

Pointer to a buffer that contains data to be transmitted before the file data is transmitted.

`HeadLength`

Size of the buffer pointed to by **Head**, in bytes, to be transmitted.

`Tail`

Pointer to a buffer that contains data to be transmitted after the file data is transmitted.

`TailLength`

Size of the buffer pointed to **Tail**, in bytes, to be transmitted.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	winsock.h (include Winsock.h)

See also

TransmitFile

TransmitFile function (winsock.h)

10/22/2022 • 11 minutes to read • Edit Online

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

> **Note** This function is a Microsoft-specific extension to the Windows Sockets specification.

Syntax

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

Parameters

`hSocket`

A handle to a connected socket. The **TransmitFile** function will transmit the file data over this socket. The socket specified by the *hSocket* parameter must be a connection-oriented socket of type **SOCK_STREAM**, **SOCK_SEQPACKET**, or **SOCK_RDM**.

`hFile`

A handle to the open file that the **TransmitFile** function transmits. Since the operating system reads the file data sequentially, you can improve caching performance by opening the handle with FILE_FLAG_SEQUENTIAL_SCAN.

The *hFile* parameter is optional. If the *hFile* parameter is **NULL**, only data in the header and/or the tail buffer is transmitted. Any additional action, such as socket disconnect or reuse, is performed as specified by the *dwFlags* parameter.

`nNumberOfBytesToWrite`

The number of bytes in the file to transmit. The **TransmitFile** function completes when it has sent the specified number of bytes, or when an error occurs, whichever occurs first.

Set this parameter to zero in order to transmit the entire file.

`nNumberOfBytesPerSend`

The size, in bytes, of each block of data sent in each send operation. This parameter is used by Windows' sockets layer to determine the block size for send operations. To select the default send size, set this parameter to zero.

The *nNumberOfBytesPerSend* parameter is useful for protocols that have limitations on the size of individual

send requests.

`lpOverlapped`

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

You can use the *lpOverlapped* parameter to specify a 64-bit offset within the file at which to start the file data transfer by setting the **Offset** and **OffsetHigh** member of the OVERLAPPED structure. If *lpOverlapped* is a **NULL** pointer, the transmission of data always starts at the current byte offset in the file.

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

`lpTransmitBuffers`

A pointer to a TRANSMIT_FILE_BUFFERS data structure that contains pointers to data to send before and after the file data is sent. This parameter should be set to a **NULL** pointer if you want to transmit only the file data.

`dwReserved`

A set of flags used to modify the behavior of the **TransmitFile** function call. The *dwFlags* parameter can contain a combination of the following options defined in the *Mswsock.h* header file:

FLAG	MEANING
TF_DISCONNECT	Start a transport-level disconnect after all the file data has been queued for transmission.
TF_REUSE_SOCKET	Prepare the socket handle to be reused. This flag is valid only if **TF_DISCONNECT** is also specified. When the **TransmitFile** request completes, the socket handle can be passed to the function call previously used to establish the connection, such as AcceptEx or ConnectEx. Such reuse is mutually exclusive; for example, if the **AcceptEx** function was called for the socket, reuse is allowed only for subsequent calls to the **AcceptEx** function, and not allowed for a subsequent call to **ConnectEx**. **Note** The socket level file transmit is subject to the behavior of the underlying transport. For example, a TCP socket may be subject to the TCP TIME_WAIT state, causing the **TransmitFile** call to be delayed.
TF_USE_DEFAULT_WORKER	Directs the Windows Sockets service provider to use the system's default thread to process long **TransmitFile** requests. The system default thread can be adjusted using the following registry parameter as a **REG_DWORD**: **HKEY_LOCAL_MACHINE\CurrentControlSet\Services\AFD\Parameters\TransmitWorker**

TF_USE_SYSTEM_THREAD	Directs the Windows Sockets service provider to use system threads to process long **TransmitFile** requests.
TF_USE_KERNEL_APC	Directs the driver to use kernel asynchronous procedure calls (APCs) instead of worker threads to process long **TransmitFile** requests. Long **TransmitFile** requests are defined as requests that require more than a single read from the file or a cache; the request therefore depends on the size of the file and the specified length of the send packet. Use of TF_USE_KERNEL_APC can deliver significant performance benefits. It is possible (though unlikely), however, that the thread in which context **TransmitFile** is initiated is being used for heavy computations; this situation may prevent APCs from launching. Note that the Winsock kernel mode driver uses normal kernel APCs, which launch whenever a thread is in a wait state, which differs from user-mode APCs, which launch whenever a thread is in an alertable wait state initiated in user mode).
TF_WRITE_BEHIND	Complete the **TransmitFile** request immediately, without pending. If this flag is specified and **TransmitFile** succeeds, then the data has been accepted by the system but not necessarily acknowledged by the remote end. Do not use this setting with the TF_DISCONNECT and TF_REUSE_SOCKET flags. **Note** If the file being sent is not in the file system cache, the request pends.

Return value

If the **TransmitFile** function succeeds, the return value is **TRUE**. Otherwise, the return value is **FALSE**. To get extended error information, call WSAGetLastError. An error code of WSA_IO_PENDING or ERROR_IO_PENDING indicates that the overlapped operation has been successfully initiated and that completion will be indicated at a later time. Any other error code indicates that the overlapped operation was not successfully initiated and no completion indication will occur. Applications should handle either ERROR_IO_PENDING or WSA_IO_PENDING in this case.

RETURN CODE	DESCRIPTION
WSAECONNABORTED	An established connection was aborted by the software in your host machine. This error is returned if the virtual circuit was terminated due to a time-out or other failure.
WSAECONNRESET	An existing connection was forcibly closed by the remote host. This error is returned for a stream socket when the virtual circuit was reset by the remote side. The application should close the socket as it is no longer usable.

WSAEFAULT	The system detected an invalid pointer address in attempting to use a pointer argument in a call. This error is returned if the *lpTransmitBuffers* or *lpOverlapped* parameter is not totally contained in a valid part of the user address space.
WSAEINVAL	An invalid argument was supplied. This error is returned if the *hSocket* parameter specified a socket of type **SOCK_DGRAM** or **SOCK_RAW**. This error is returned if the *dwFlags* parameter has the **TF_REUSE_SOCKET** flag set, but the **TF_DISCONNECT** flag was not set. This error is also returned if the offset specified in the OVERLAPPED structure pointed to by the *lpOverlapped* is not within the file. This error is also returned if the *nNumberOfBytesToWrite* parameter is set to a value greater than 2,147,483,646, the maximum value for a 32-bit integer minus 1.
WSAENETDOWN	A socket operation encountered a dead network.This error is returned if the network subsystem has failed.
WSAENETRESET	The connection has been broken due to keep-alive activity detecting a failure while the operation was in progress.
WSAENOBUFS	An operation on a socket could not be performed because the system lacked sufficient buffer space or because a queue was full. This error is also returned if the Windows Sockets provider reports a buffer deadlock.
WSAENOTCONN	A request to send or receive data was disallowed because the socket is not connected.
WSAENOTSOCK	An operation was attempted on something that is not a socket. This error is returned if the *hSocket* parameter is not a socket.
WSAESHUTDOWN	A request to send or receive data was disallowed because the socket had already been shut down in that direction with a previous shutdown call. This error is returned if the socket has been shut down for sending. It is not possible to call TransmitFile on a socket after the shutdown function has been called on the socket with the *how* parameter set to **SD_SEND** or **SD_BOTH**.
WSANOTINITIALISED	Either the application has not called the WSAStartup function, or **WSAStartup** failed. A successful **WSAStartup** call must occur before using the TransmitFile function.
WSA_IO_PENDING	An overlapped I/O operation is in progress. This value is returned if an overlapped I/O operation was successfully initiated and indicates that 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 the overlapped operation has been canceled due to the closure of the socket, the execution of the "SIO_FLUSH" command in WSAIoctl, or the thread that initiated the overlapped request exited before the operation completed. **Note** All I/O initiated by a given thread is canceled when that thread exits. For overlapped sockets, pending asynchronous operations can fail if the thread is closed before the asynchronous operations complete. For more information, see ExitThread.

Remarks

The **TransmitFile** function uses the operating system's cache manager to retrieve the file data, and provide high-performance file data transfer over sockets.

The **TransmitFile** function only supports connection-oriented sockets of type **SOCK_STREAM**, **SOCK_SEQPACKET**, and **SOCK_RDM**. Sockets of type **SOCK_DGRAM** and **SOCK_RAW** are not supported. The TransmitPackets function can be used with sockets of type **SOCK_DGRAM**.

The maximum number of bytes that can be transmitted using a single call to the **TransmitFile** function is 2,147,483,646, the maximum value for a 32-bit integer minus 1. The maximum number of bytes to send in a single call includes any data sent before or after the file data pointed to by the *lpTransmitBuffers* parameter plus the value specified in the *nNumberOfBytesToWrite* parameter for the length of file data to send. If an application needs to transmit a file larger than 2,147,483,646 bytes, then multiple calls to the **TransmitFile** function can be used with each call transferring no more than 2,147,483,646 bytes. Setting the *nNumberOfBytesToWrite* parameter to zero for a file larger than 2,147,483,646 bytes will also fail since in this case the **TransmitFile** function will use the size of the file as the value for the number of bytes to transmit.

> **Note** The function pointer for the **TransmitFile** function must be obtained at run time by making a call to the WSAIoctl function with the **SIO_GET_EXTENSION_FUNCTION_POINTER** opcode specified. The input buffer passed to the **WSAIoctl** function must contain **WSAID_TRANSMITFILE**, a globally unique identifier (GUID) whose value identifies the **TransmitFile** extension function. On success, the output returned by the **WSAIoctl** function contains a pointer to the **TransmitFile** function. The **WSAID_TRANSMITFILE** GUID is defined in the *Mswsock.h* header file.

> **Note** **TransmitFile** is not functional on transports that perform their own buffering. Transports with the TDI_SERVICE_INTERNAL_BUFFERING flag set, such as ADSP, perform their own buffering. Because **TransmitFile** achieves its performance gains by sending data directly from the file cache. Transports that run out of buffer space on a particular connection are not handled by **TransmitFile**, and as a result of running out of buffer space on the connection, **TransmitFile** returns STATUS_DEVICE_NOT_READY.

The **TransmitFile** function was primarily added to Winsock for use by high-performance server applications (web and ftp servers, for example).

Workstation and client versions of Windows optimize the **TransmitFile** function for minimum memory and resource utilization by limiting the number of concurrent **TransmitFile** operations allowed on the system to a maximum of two. On Windows Vista, Windows XP, Windows 2000 Professional, and Windows NT Workstation 3.51 and later only two outstanding **TransmitFile** requests are handled simultaneously; the third request will wait until one of the previous requests is completed.

Server versions of Windows optimize the **TransmitFile** function for high performance. On server versions, there are no default limits placed on the number of concurrent **TransmitFile** operations allowed on the system. Expect better performance results when using **TransmitFile** on server versions of Windows. On server versions of Windows, it is possible to set a limit on the maximum number of concurrent **TransmitFile** operations by creating a registry entry and setting a value for the following **REG_DWORD**:

HKEY_LOCAL_MACHINE\CurrentControlSet\Services\AFD\Parameters\MaxActiveTransmitFileCount

If the **TransmitFile** function is called with TCP socket (protocol of IPPROTO_TCP) with both the **TF_DISCONNECT** and **TF_REUSE_SOCKET** flags specified, the call will not complete until the two following conditions are met.

- All pending receive data sent by remote side (received prior to a FIN from the remote side) on the TCP socket has been read.
- The remote side has closed the connection (completed the graceful TCP connection closure).

If the **TransmitFile** function is called with the *lpOverlapped* parameter set to **NULL**, the operation is executed as synchronous I/O. The function will not complete until the file has been sent.

Windows Phone 8: This function is supported for Windows Phone Store apps on Windows Phone 8 and later.

Windows 8.1 and **Windows Server 2012 R2**: This function is supported for Windows Store apps on Windows 8.1, Windows Server 2012 R2, and later.

Notes for QoS

The **TransmitFile** function allows the setting of two flags, TF_DISCONNECT or TF_REUSE_SOCKET, that return the socket to a "disconnected, reusable" state after the file has been transmitted. These flags should not be used on a socket where quality of service has been requested, since the service provider may immediately delete any quality of service associated with the socket before the file transfer has completed. The best approach for a QoS-enabled socket is to simply call the closesocket function when the file transfer has completed, rather than relying on these flags.

Requirements

Minimum supported client	Windows 8.1, Windows Vista [desktop apps | UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps | UWP apps]
Target Platform	Windows
Header	winsock.h (include Mswsock.h)
Library	Mswsock.lib
DLL	Mswsock.dll

See also

ExitThread

OVERLAPPED

TRANSMIT_FILE_BUFFERS

TransmitPackets

WSASend

closesocket

WSAAsyncGetHostByAddr function (winsock.h)

10/22/2022 • 4 minutes to read • Edit Online

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

> **Note** The **WSAAsyncGetHostByAddr** function is not designed to provide parallel resolution of several addresses. Therefore, applications that issue several requests should not expect them to be executed concurrently. Alternatively, applications can start another thread and use the getnameinfo function to resolve addresses in an IP-version agnostic manner. Developers creating Windows Sockets 2 applications are urged to use the **getnameinfo** function to enable smooth transition to IPv6 compatibility.

Syntax

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

Parameters

`hWnd`

TBD

`wMsg`

TBD

`addr`

TBD

`len`

TBD

`type`

TBD

`buf`

TBD

`buflen`

TBD

Return value

The return value specifies whether or not the asynchronous operation was successfully initiated. It does not imply success or failure of the operation itself.

If no error occurs, **WSAAsyncGetHostByAddr** returns a nonzero value of type HANDLE that is the asynchronous task handle (not to be confused with a Windows HTASK) for the request. This value can be used in two ways. It can be used to cancel the operation using WSACancelAsyncRequest, or it can be used to match up asynchronous operations and completion messages by examining the *wParam* message parameter.

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

The following error codes can be set when an application window receives a message. As described above, they can be extracted from the *lParam* in the reply message using the **WSAGETASYNCERROR** macro.

ERROR CODE	MEANING
WSAENETDOWN	The network subsystem has failed.
WSAENOBUFS	Insufficient buffer space is available.
WSAEFAULT	The *addr* or *buf* parameter is not in a valid part of the process address space.
WSAHOST_NOT_FOUND	Authoritative answer host not found.
WSATRY_AGAIN	Nonauthoritative host not found, or SERVERFAIL.
WSANO_RECOVERY	Nonrecoverable errors: FORMERR, REFUSED, NOTIMP.
WSANO_DATA	Valid name, no data record of requested type.

The following errors can occur at the time of the function call, and indicate that the asynchronous operation could not be initiated.

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

WSAEWOULDBLOCK	The asynchronous operation cannot be scheduled at this time due to resource or other constraints within the Windows Sockets implementation.

Remarks

The **WSAAsyncGetHostByAddr** function is an asynchronous version of gethostbyaddr. It is used to retrieve the host name and address information that corresponds to a network address. Windows Sockets initiates the operation and returns to the caller immediately, passing back an opaque, asynchronous task handle that the application can use to identify the operation. When the operation is completed, the results (if any) are copied into the buffer provided by the caller and a message is sent to the application's window.

When the asynchronous operation has completed, the application window indicated by the *hWnd* parameter receives message in the *wMsg* parameter. The *wParam* parameter contains the asynchronous task handle as returned by the original function call. The high 16 bits of *lParam* contain any error code. The error code can be any error as defined in Winsock2.h. An error code of zero indicates successful completion of the asynchronous operation.

On successful completion, the buffer specified to the original function call contains a hostent structure. To access the members of this structure, the original buffer address is cast to a **hostent** structure pointer and accessed as appropriate.

If the error code is WSAENOBUFS, the size of the buffer specified by *buflen* in the original call was too small to contain all the resulting information. In this case, the low 16 bits of *lParam* contain the size of buffer required to supply all the requisite information. If the application decides that the partial data is inadequate, it can reissue the **WSAAsyncGetHostByAddr** function call with a buffer large enough to receive all the desired information (that is, no smaller than the low 16 bits of *lParam*).

The buffer specified to this function is used by Windows Sockets to construct a structure together with the contents of data areas referenced by members of the same hostent structure. To avoid the WSAENOBUFS error, the application should provide a buffer of at least MAXGETHOSTSTRUCT bytes (as defined in Winsock2.h).

The error code and buffer length should be extracted from the *lParam* using the macros **WSAGETASYNCERROR** and **WSAGETASYNCBUFLEN**, defined in Winsock2.h as:

```
#include <windows.h>

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

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

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winsock.h (include Winsock2.h, Winsock.h)

Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

WSACancelAsyncRequest

Winsock Functions

Winsock Reference

getaddrinfo

gethostbyaddr

getnameinfo

hostent

WSAAsyncGetHostByName function (winsock.h)

10/22/2022 • 4 minutes to read • Edit Online

The **WSAAsyncGetHostByName** function asynchronously retrieves host information that corresponds to a host name.

> **Note** The **WSAAsyncGetHostByName** function is not designed to provide parallel resolution of several names. Therefore, applications that issue several requests should not expect them to be executed concurrently. Alternatively, applications can start another thread and use the getaddrinfo function to resolve names in an IP-version agnostic manner. Developers creating Windows Sockets 2 applications are urged to use the **getaddrinfo** function to enable smooth transition to IPv6 compatibility.

Syntax

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

Parameters

`hWnd`

TBD

`wMsg`

TBD

`name`

TBD

`buf`

TBD

`buflen`

TBD

Return value

The return value specifies whether or not the asynchronous operation was successfully initiated. It does not imply success or failure of the operation itself.

If no error occurs, **WSAAsyncGetHostByName** returns a nonzero value of type HANDLE that is the asynchronous task handle (not to be confused with a Windows HTASK) for the request. This value can be used in two ways. It can be used to cancel the operation using WSACancelAsyncRequest, or it can be used to match up

asynchronous operations and completion messages by examining the *wParam* message parameter.

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

The following error codes can be set when an application window receives a message. As described above, they can be extracted from the *lParam* in the reply message using the WSAGETASYNCERROR macro.

ERROR CODE	MEANING
WSAENETDOWN	The network subsystem has failed.
WSAENOBUFS	Insufficient buffer space is available.
WSAEFAULT	The *name* or *buf* parameter is not in a valid part of the process address space.
WSAHOST_NOT_FOUND	Authoritative answer host not found.
WSATRY_AGAIN	A nonauthoritative host not found, or SERVERFAIL.
WSANO_RECOVERY	Nonrecoverable errors: FORMERR, REFUSED, NOTIMP.
WSANO_DATA	Valid name, no data record of requested type.

The following errors can occur at the time of the function call, and indicate that the asynchronous operation could not be initiated.

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

Remarks

The **WSAAsyncGetHostByName** function is an asynchronous version of gethostbyname, and is used to retrieve host name and address information corresponding to a host name. Windows Sockets initiates the operation and returns to the caller immediately, passing back an opaque asynchronous task handle that which

the application can use to identify the operation. When the operation is completed, the results (if any) are copied into the buffer provided by the caller and a message is sent to the application's window.

When the asynchronous operation has completed, the application window indicated by the *hWnd* parameter receives message in the *wMsg* parameter. The *wParam* parameter contains the asynchronous task handle as returned by the original function call. The high 16 bits of *lParam* contain any error code. The error code can be any error as defined in Winsock2.h. An error code of zero indicates successful completion of the asynchronous operation.

On successful completion, the buffer specified to the original function call contains a hostent structure. To access the elements of this structure, the original buffer address should be cast to a **hostent** structure pointer and accessed as appropriate.

If the error code is WSAENOBUFS, the size of the buffer specified by *buflen* in the original call was too small to contain all the resulting information. In this case, the low 16 bits of *lParam* contain the size of buffer required to supply all the requisite information. If the application decides that the partial data is inadequate, it can reissue the **WSAAsyncGetHostByName** function call with a buffer large enough to receive all the desired information (that is, no smaller than the low 16 bits of *lParam*).

The buffer specified to this function is used by Windows Sockets to construct a hostent structure together with the contents of data areas referenced by members of the same **hostent** structure. To avoid the WSAENOBUFS error, the application should provide a buffer of at least MAXGETHOSTSTRUCT bytes (as defined in Winsock2.h).

The error code and buffer length should be extracted from the *lParam* using the macros **WSAGETASYNCERROR** and **WSAGETASYNCBUFLEN**, defined in Winsock2.h as:

```
#include <windows.h>

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

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

WSAAsyncGetHostByName is guaranteed to resolve the string returned by a successful call to gethostname.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winsock.h (include Winsock2.h, Winsock.h)
Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

WSACancelAsyncRequest

Winsock Functions

Winsock Reference

getaddrinfo

gethostbyname

getnameinfo

hostent

WSAAsyncGetProtoByName function (winsock.h)

10/22/2022 • 4 minutes to read • Edit Online

The **WSAAsyncGetProtoByName** function asynchronously retrieves protocol information that corresponds to a protocol name.

Syntax

```
HANDLE WSAAsyncGetProtoByName(
  [in]  HWND       hWnd,
  [in]  u_int      wMsg,
  [in]  const char *name,
  [out] char       *buf,
  [out] int        buflen
);
```

Parameters

`[in] hWnd`

Handle of the window that will receive a message when the asynchronous request completes.

`[in] wMsg`

Message to be received when the asynchronous request completes.

`[in] name`

Pointer to the null-terminated protocol name to be resolved.

`[out] buf`

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

`[out] buflen`

Size of data area for the *buf* parameter, in bytes.

Return value

The return value specifies whether or not the asynchronous operation was successfully initiated. It does not imply success or failure of the operation itself.

If no error occurs, **WSAAsyncGetProtoByName** returns a nonzero value of type HANDLE that is the asynchronous task handle for the request (not to be confused with a Windows HTASK). This value can be used in two ways. It can be used to cancel the operation using WSACancelAsyncRequest, or it can be used to match up asynchronous operations and completion messages, by examining the *wParam* message parameter.

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

The following error codes can be set when an application window receives a message. As described above, they can be extracted from the *lParam* in the reply message using the **WSAGETASYNCERROR** macro.

ERROR CODE	MEANING
WSAENETDOWN	The network subsystem has failed.
WSAENOBUFS	Insufficient buffer space is available.
WSAEFAULT	The *name* or *buf* parameter is not in a valid part of the process address space.
WSAHOST_NOT_FOUND	Authoritative answer protocol not found.
WSATRY_AGAIN	A nonauthoritative protocol not found, or server failure.
WSANO_RECOVERY	Nonrecoverable errors, the protocols database is not accessible.
WSANO_DATA	Valid name, no data record of requested type.

The following errors can occur at the time of the function call, and indicate that the asynchronous operation could not be initiated.

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

Remarks

The **WSAAsyncGetProtoByName** function is an asynchronous version of getprotobyname. It is used to retrieve the protocol name and number from the Windows Sockets database corresponding to a given protocol name. Windows Sockets initiates the operation and returns to the caller immediately, passing back an opaque, asynchronous task handle that the application can use to identify the operation. When the operation is completed, the results (if any) are copied into the buffer provided by the caller and a message is sent to the application's window.

When the asynchronous operation has completed, the application window indicated by the *hWnd* parameter receives message in the *wMsg* parameter. The *wParam* parameter contains the asynchronous task handle as returned by the original function call. The high 16 bits of *lParam* contain any error code. The error code can be any error as defined in Winsock2.h. An error code of zero indicates successful completion of the asynchronous operation.

On successful completion, the buffer specified to the original function call contains a protoent structure. To access the members of this structure, the original buffer address should be cast to a **protoent** structure pointer and accessed as appropriate.

If the error code is WSAENOBUFS, the size of the buffer specified by *buflen* in the original call was too small to contain all the resulting information. In this case, the low 16 bits of *lParam* contain the size of buffer required to supply all the requisite information. If the application decides that the partial data is inadequate, it can reissue the **WSAAsyncGetProtoByName** function call with a buffer large enough to receive all the desired information (that is, no smaller than the low 16 bits of *lParam*).

The buffer specified to this function is used by Windows Sockets to construct a protoent structure together with the contents of data areas referenced by members of the same **protoent** structure. To avoid the WSAENOBUFS error noted above, the application should provide a buffer of at least MAXGETHOSTSTRUCT bytes (as defined in Winsock2.h).

The error code and buffer length should be extracted from the *lParam* using the macros **WSAGETASYNCERROR** and **WSAGETASYNCBUFLEN**, defined in Winsock2.h as:

```
#include <windows.h>

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

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

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winsock.h (include Winsock2.h)
Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

WSACancelAsyncRequest

Winsock Functions

Winsock Reference

getprotobyname

WSAAsyncGetProtoByNumber function (winsock.h)

10/22/2022 • 4 minutes to read • Edit Online

The **WSAAsyncGetProtoByNumber** function asynchronously retrieves protocol information that corresponds to a protocol number.

Syntax

```
HANDLE WSAAsyncGetProtoByNumber(
  [in]  HWND  hWnd,
  [in]  u_int wMsg,
  [in]  int   number,
  [out] char  *buf,
  [in]  int   buflen
);
```

Parameters

`[in] hWnd`

Handle of the window that will receive a message when the asynchronous request completes.

`[in] wMsg`

Message to be received when the asynchronous request completes.

`[in] number`

Protocol number to be resolved, in host byte order.

`[out] buf`

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

`[in] buflen`

Size of data area for the *buf* parameter, in bytes.

Return value

The return value specifies whether or not the asynchronous operation was successfully initiated. It does not imply success or failure of the operation itself.

If no error occurs, **WSAAsyncGetProtoByNumber** returns a nonzero value of type **HANDLE** that is the asynchronous task handle for the request (not to be confused with a Windows HTASK). This value can be used in two ways. It can be used to cancel the operation using WSACancelAsyncRequest, or it can be used to match up asynchronous operations and completion messages, by examining the *wParam* message parameter.

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

The following error codes can be set when an application window receives a message. As described above, they can be extracted from the *lParam* in the reply message using the **WSAGETASYNCERROR** macro.

ERROR CODE	MEANING
WSAENETDOWN	The network subsystem has failed.
WSAENOBUFS	Insufficient buffer space is available.
WSAEFAULT	The *buf* parameter is not in a valid part of the process address space.
WSAHOST_NOT_FOUND	Authoritative answer protocol not found.
WSATRY_AGAIN	Nonauthoritative protocol not found, or server failure.
WSANO_RECOVERY	Nonrecoverable errors, the protocols database is not accessible.
WSANO_DATA	Valid name, no data record of requested type.

The following errors can occur at the time of the function call, and indicate that the asynchronous operation could not be initiated.

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

Remarks

The **WSAAsyncGetProtoByNumber** function is an asynchronous version of getprotobynumber, and is used to retrieve the protocol name and number corresponding to a protocol number. Windows Sockets initiates the operation and returns to the caller immediately, passing back an opaque, asynchronous task handle that the application can use to identify the operation. When the operation is completed, the results (if any) are copied into the buffer provided by the caller and a message is sent to the application's window.

When the asynchronous operation has completed, the application window indicated by the *hWnd* parameter

receives message in the *wMsg* parameter. The *wParam* parameter contains the asynchronous task handle as returned by the original function call. The high 16 bits of *lParam* contain any error code. The error code can be any error as defined in Winsock2.h. An error code of zero indicates successful completion of the asynchronous operation.

On successful completion, the buffer specified to the original function call contains a protoent structure. To access the members of this structure, the original buffer address is cast to a **protoent** structure pointer and accessed as appropriate.

If the error code is WSAENOBUFS, the size of the buffer specified by *buflen* in the original call was too small to contain all the resulting information. In this case, the low 16 bits of *lParam* contain the size of buffer required to supply all the requisite information. If the application decides that the partial data is inadequate, it can reissue the **WSAAsyncGetProtoByNumber** function call with a buffer large enough to receive all the desired information (that is, no smaller than the low 16 bits of *lParam*).

The buffer specified to this function is used by Windows Sockets to construct a protoent structure together with the contents of data areas referenced by members of the same **protoent** structure. To avoid the WSAENOBUFS error noted above, the application should provide a buffer of at least MAXGETHOSTSTRUCT bytes (as defined in Winsock2.h).

The error code and buffer length should be extracted from the *lParam* using the macros **WSAGETASYNCERROR** and **WSAGETASYNCBUFLEN**, defined in Winsock2.h as:

```
#include <windows.h>

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

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

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winsock.h (include Winsock2.h)
Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

WSACancelAsyncRequest

Winsock Functions

Winsock Reference

WSAAsyncGetServByName function (winsock.h)

10/22/2022 • 4 minutes to read • Edit Online

The **WSAAsyncGetServByName** function asynchronously retrieves service information that corresponds to a service name and port.

Syntax

```
HANDLE WSAAsyncGetServByName(
  [in]  HWND       hWnd,
  [in]  u_int      wMsg,
  [in]  const char *name,
  [in]  const char *proto,
  [out] char       *buf,
  [in]  int        buflen
);
```

Parameters

`[in] hWnd`

Handle of the window that should receive a message when the asynchronous request completes.

`[in] wMsg`

Message to be received when the asynchronous request completes.

`[in] name`

Pointer to a **null**-terminated service name.

`[in] proto`

Pointer to a protocol name. This can be **NULL**, in which case **WSAAsyncGetServByName** will search for the first service entry for which *s_name* or one of the *s_aliases* matches the given *name*. Otherwise, **WSAAsyncGetServByName** matches both *name* and *proto*.

`[out] buf`

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

`[in] buflen`

Size of data area for the *buf* parameter, in bytes.

Return value

The return value specifies whether or not the asynchronous operation was successfully initiated. It does not imply success or failure of the operation itself.

If no error occurs, **WSAAsyncGetServByName** returns a nonzero value of type **HANDLE** that is the

asynchronous task handle for the request (not to be confused with a Windows HTASK). This value can be used in two ways. It can be used to cancel the operation using WSACancelAsyncRequest, or it can be used to match up asynchronous operations and completion messages, by examining the *wParam* message parameter.

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

The following error codes can be set when an application window receives a message. As described above, they can be extracted from the *lParam* in the reply message using the **WSAGETASYNCERROR** macro.

ERROR CODE	MEANING
WSAENETDOWN	The network subsystem has failed.
WSAENOBUFS	Insufficient buffer space is available.
WSAEFAULT	The *buf* parameter is not in a valid part of the process address space.
WSAHOST_NOT_FOUND	Authoritative answer host not found.
WSATRY_AGAIN	Nonauthoritative service not found, or server failure.
WSANO_RECOVERY	Nonrecoverable errors, the services database is not accessible.
WSANO_DATA	Valid name, no data record of requested type.

The following errors can occur at the time of the function call, and indicate that the asynchronous operation could not be initiated.

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

Remarks

The **WSAAsyncGetServByName** function is an asynchronous version of getservbyname and is used to

retrieve service information corresponding to a service name. Windows Sockets initiates the operation and returns to the caller immediately, passing back an opaque, asynchronous task handle that the application can use to identify the operation. When the operation is completed, the results (if any) are copied into the buffer provided by the caller and a message is sent to the application's window.

When the asynchronous operation has completed, the application window indicated by the *hWnd* parameter receives message in the *wMsg* parameter. The *wParam* parameter contains the asynchronous task handle as returned by the original function call. The high 16 bits of *lParam* contain any error code. The error code can be any error as defined in Winsock2.h. An error code of zero indicates successful completion of the asynchronous operation.

On successful completion, the buffer specified to the original function call contains a servent structure. To access the members of this structure, the original buffer address should be cast to a **servent** structure pointer and accessed as appropriate.

If the error code is WSAENOBUFS, the size of the buffer specified by *buflen* in the original call was too small to contain all the resulting information. In this case, the low 16 bits of *lParam* contain the size of buffer required to supply all the requisite information. If the application decides that the partial data is inadequate, it can reissue the **WSAAsyncGetServByName** function call with a buffer large enough to receive all the desired information (that is, no smaller than the low 16 bits of *lParam*).

The buffer specified to this function is used by Windows Sockets to construct a servent structure together with the contents of data areas referenced by members of the same **servent** structure. To avoid the WSAENOBUFS error, the application should provide a buffer of at least MAXGETHOSTSTRUCT bytes (as defined in Winsock2.h).

The error code and buffer length should be extracted from the *lParam* using the macros **WSAGETASYNCERROR** and **WSAGETASYNCBUFLEN**, defined in Winsock2.h as:

```
#include <windows.h>

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

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

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winsock.h (include Winsock2.h)
Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

WSACancelAsyncRequest

Winsock Functions

Winsock Reference

getservbyname

WSAAsyncGetServByPort function (winsock.h)

10/22/2022 • 4 minutes to read • Edit Online

The **WSAAsyncGetServByPort** function asynchronously retrieves service information that corresponds to a port and protocol.

Syntax

```
HANDLE WSAAsyncGetServByPort(
  [in]  HWND       hWnd,
  [in]  u_int      wMsg,
  [in]  int        port,
  [in]  const char *proto,
  [out] char       *buf,
  [in]  int        buflen
);
```

Parameters

`[in] hWnd`

Handle of the window that should receive a message when the asynchronous request completes.

`[in] wMsg`

Message to be received when the asynchronous request completes.

`[in] port`

Port for the service, in network byte order.

`[in] proto`

Pointer to a protocol name. This can be **NULL**, in which case **WSAAsyncGetServByPort** will search for the first service entry for which *s_port* match the given *port*. Otherwise, **WSAAsyncGetServByPort** matches both *port* and *proto*.

`[out] buf`

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

`[in] buflen`

Size of data area for the *buf* parameter, in bytes.

Return value

The return value specifies whether or not the asynchronous operation was successfully initiated. It does not imply success or failure of the operation itself.

If no error occurs, **WSAAsyncGetServByPort** returns a nonzero value of type **HANDLE** that is the

asynchronous task handle for the request (not to be confused with a Windows HTASK). This value can be used in two ways. It can be used to cancel the operation using WSACancelAsyncRequest, or it can be used to match up asynchronous operations and completion messages, by examining the *wParam* message parameter.

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

The following error codes can be set when an application window receives a message. As described above, they can be extracted from the *lParam* in the reply message using the **WSAGETASYNCERROR** macro.

ERROR CODE	MEANING
WSAENETDOWN	The network subsystem has failed.
WSAENOBUFS	Insufficient buffer space is available.
WSAEFAULT	The *proto* or *buf* parameter is not in a valid part of the process address space.
WSAHOST_NOT_FOUND	Authoritative answer port not found.
WSATRY_AGAIN	Nonauthoritative port not found, or server failure.
WSANO_RECOVERY	Nonrecoverable errors, the services database is not accessible.
WSANO_DATA	Valid name, no data record of requested type.

The following errors can occur at the time of the function call, and indicate that the asynchronous operation could not be initiated.

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

Remarks

The **WSAAsyncGetServByPort** function is an asynchronous version of getservbyport, and is used to retrieve

service information corresponding to a port number. Windows Sockets initiates the operation and returns to the caller immediately, passing back an opaque, asynchronous task handle that the application can use to identify the operation. When the operation is completed, the results (if any) are copied into the buffer provided by the caller and a message is sent to the application's window.

When the asynchronous operation has completed, the application window indicated by the *hWnd* parameter receives message in the *wMsg* parameter. The *wParam* parameter contains the asynchronous task handle as returned by the original function call. The high 16 bits of *lParam* contain any error code. The error code can be any error as defined in Winsock2.h. An error code of zero indicates successful completion of the asynchronous operation.

On successful completion, the buffer specified to the original function call contains a servent structure. To access the members of this structure, the original buffer address should be cast to a **servent** structure pointer and accessed as appropriate.

If the error code is WSAENOBUFS, the size of the buffer specified by *buflen* in the original call was too small to contain all the resulting information. In this case, the low 16 bits of *lParam* contain the size of buffer required to supply all the requisite information. If the application decides that the partial data is inadequate, it can reissue the **WSAAsyncGetServByPort** function call with a buffer large enough to receive all the desired information (that is, no smaller than the low 16 bits of *lParam*).

The buffer specified to this function is used by Windows Sockets to construct a servent structure together with the contents of data areas referenced by members of the same **servent** structure. To avoid the WSAENOBUFS error, the application should provide a buffer of at least MAXGETHOSTSTRUCT bytes (as defined in Winsock2.h).

The error code and buffer length should be extracted from the *lParam* using the macros **WSAGETASYNCERROR** and **WSAGETASYNCBUFLEN**, defined in Winsock2.h as:

```
#include <windows.h>

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

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

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winsock.h (include Winsock2.h)
Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

WSACancelAsyncRequest

Winsock Functions

Winsock Reference

getservbyport

WSAAsyncSelect function (winsock.h)

10/22/2022 • 14 minutes to read • Edit Online

[The **WSAAsyncSelect** function is available for use in the operating systems specified in the Requirements section. It may be altered or unavailable in subsequent versions. Rather than use Select-style I/O, use Overlapped I/O and Event Objects with WinSock2.]

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

Syntax

```
int WSAAsyncSelect(
  [in] SOCKET s,
  [in] HWND   hWnd,
  [in] u_int  wMsg,
  [in] long   lEvent
);
```

Parameters

`[in] s`

A descriptor that identifies the socket for which event notification is required.

`[in] hWnd`

A handle that identifies the window that will receive a message when a network event occurs.

`[in] wMsg`

A message to be received when a network event occurs.

`[in] lEvent`

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

Return value

If the **WSAAsyncSelect** function succeeds, the return value is zero, provided that the application's declaration of interest in the network event set was successful. Otherwise, the value SOCKET_ERROR is returned, and a specific error number can be retrieved by calling WSAGetLastError.

ERROR CODE	MEANING
WSANOTINITIALISED	A successful WSAStartup call must occur before using this function.
WSAENETDOWN	The network subsystem failed.

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

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

Event: FD_CONNECT

ERROR CODE	MEANING
WSAEAFNOSUPPORT	Addresses in the specified family cannot be used with this socket.
WSAECONNREFUSED	The attempt to connect was rejected.
WSAENETUNREACH	The network cannot be reached from this host at this time.
WSAEFAULT	The *namelen* parameter is invalid.
WSAEINVAL	The socket is already bound to an address.
WSAEISCONN	The socket is already connected.
WSAEMFILE	No more file descriptors are available.
WSAENOBUFS	No buffer space is available. The socket cannot be connected.
WSAENOTCONN	The socket is not connected.
WSAETIMEDOUT	Attempt to connect timed out without establishing a connection.

Event: FD_CLOSE

ERROR CODE	MEANING
WSAENETDOWN	The network subsystem failed.
WSAECONNRESET	The connection was reset by the remote side.
WSAECONNABORTED	The connection was terminated due to a time-out or other failure.

ERROR CODE	MEANING
WSAENETDOWN	The network subsystem failed.

Event: FD_ROUTING_INTERFACE_CHANGE

ERROR CODE	MEANING
WSAENETUNREACH	The specified destination is no longer reachable.
WSAENETDOWN	The network subsystem failed.

Remarks

The **WSAAsyncSelect** function is used to request that WS2_32.DLL should send a message to the window *hWnd* when it detects any network event specified by the *lEvent* parameter. The message that should be sent is specified by the *wMsg* parameter. The socket for which notification is required is identified by the *s* parameter.

The **WSAAsyncSelect** function automatically sets socket *s* to nonblocking mode, regardless of the value of *lEvent*. To set socket *s* back to blocking mode, it is first necessary to clear the event record associated with socket *s* via a call to **WSAAsyncSelect** with *lEvent* set to zero. You can then call ioctlsocket or WSAIoctl to set the socket back to blocking mode. For more information about how to set the nonblocking socket back to blocking mode, see the **ioctlsocket** and **WSAIoctl** functions.

The *lEvent* parameter is constructed by using the bitwise OR operator with any value listed in the following table.

VALUE	MEANING
FD_READ	Set to receive notification of readiness for reading.
FD_WRITE	Wants to receive notification of readiness for writing.
FD_OOB	Wants to receive notification of the arrival of OOB data.
FD_ACCEPT	Wants to receive notification of incoming connections.
FD_CONNECT	Wants to receive notification of completed connection or multipoint join operation.
FD_CLOSE	Wants to receive notification of socket closure.
FD_QOS	Wants to receive notification of socket Quality of Service (QoS) changes.
FD_GROUP_QOS	Wants to receive notification of socket group Quality of Service (QoS) changes (reserved for future use with socket groups). Reserved.
FD_ROUTING_INTERFACE_CHANGE	Wants to receive notification of routing interface changes for the specified destination(s).

FD_ADDRESS_LIST_CHANGE	Wants to receive notification of local address list changes for the socket protocol family.

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

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

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

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

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

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

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

The socket created by the accept function has the same properties as the listening socket used to accept it. Consequently, **WSAAsyncSelect** events set for the listening socket also apply to the accepted socket. For example, if a listening socket has **WSAAsyncSelect** events **FD_ACCEPT**, **FD_READ**, and **FD_WRITE**, then any socket accepted on that listening socket will also have **FD_ACCEPT**, **FD_READ**, and **FD_WRITE** events with the same *wMsg* value used for messages. If a different *wMsg* or events are desired, the application should call **WSAAsyncSelect**, passing the accepted socket and the desired new data.

When one of the nominated network events occurs on the specified socket *s*, the application window *hWnd* receives message *wMsg*. The *wParam* parameter identifies the socket on which a network event has occurred. The low word of *lParam* specifies the network event that has occurred. The high word of *lParam* contains any error code. The error code be any error as defined in Winsock2.h.

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

The error and event codes can be extracted from the *lParam* using the macros **WSAGETSELECTERROR** and **WSAGETSELECTEVENT**, defined in Winsock2.h as:

```
#include <windows.h>

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

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

The possible network event codes that can be returned are listed in the following table.

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

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

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

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

6. Application issues `recv(s,...)`, which fails with the error WSAEWOULDBLOCK.

Other sequences are also possible.

The WS2_32.DLL will not continually flood an application with messages for a particular network event. Having successfully posted notification of a particular event to an application window, no further message(s) for that network event will be posted to the application window until the application makes the function call that implicitly reenables notification of that network event.

EVENT	REENABLING FUNCTION
FD_READ	recv, recvfrom, WSARecv, or WSARecvFrom.
FD_WRITE	send, sendto, WSASend, or WSASendTo.
FD_OOB	recv, recvfrom, WSARecv, or WSARecvFrom.
FD_ACCEPT	accept or WSAAccept unless the error code is WSATRY_AGAIN indicating that the condition function returned CF_DEFER.
FD_CONNECT	None.
FD_CLOSE	None.
FD_QOS	WSAIoctl with command SIO_GET_QOS.
FD_GROUP_QOS	Reserved. WSAIoctl with command SIO_GET_GROUP_QOS (reserved for future use with socket groups).
FD_ROUTING_INTERFACE_CHANGE	WSAIoctl with command SIO_ROUTING_INTERFACE_CHANGE.
FD_ADDRESS_LIST_CHANGE	WSAIoctl with command SIO_ADDRESS_LIST_CHANGE.

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

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

1. Network transport stack receives 100 bytes of data on socket *s* and causes Windows Sockets 2 to post an **FD_READ** message.
2. The application issues recv(*s*, *buffptr*, 50, 0) to read 50 bytes.
3. Another **FD_READ** message is posted because there is still data to be read.

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

The **FD_QOS** and **FD_GROUP_QOS** events are considered edge triggered. A message will be posted exactly

once when a quality of service change occurs. Further messages will not be forthcoming until either the provider detects a further change in quality of service or the application renegotiates the quality of service for the socket.

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

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

If any event has occurred when the application calls **WSAAsyncSelect** or when the reenabling function is called, then a message is posted as appropriate. For example, consider the following sequence:

1. An application calls listen.
2. A connect request is received, but not yet accepted.
3. The application calls **WSAAsyncSelect** specifying that it requires receiving **FD_ACCEPT** messages for the socket. Due to the persistence of events, Windows Sockets 2 posts an **FD_ACCEPT** message immediately.

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

The **FD_OOB** event is used only when a socket is configured to receive OOB data separately. If the socket is configured to receive OOB data inline, the OOB (expedited) data is treated as normal data and the application should register an interest in, and will receive, **FD_READ** events, not **FD_OOB** events. An application can set or inspect the way in which OOB data is to be handled by using setsockopt or getsockopt for the SO_OOBINLINE option.

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

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

Be aware that the application will only receive an **FD_CLOSE** message to indicate closure of a virtual circuit, and only when all the received data has been read if this is a graceful close. It will not receive an **FD_READ** message to indicate this condition.

The **FD_QOS** or **FD_GROUP_QOS** message is posted when any parameter in the flow specification associated with socket *s* or the socket group that *s* belongs to has changed, respectively. Applications should use WSAIoctl with command SIO_GET_QOS or SIO_GET_GROUP_QOS to get the current quality of service for socket *s* or for the socket group *s* belongs to, respectively.

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

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

- **FD_READ**:
  1. When **WSAAsyncSelect** is called, if there is data currently available to receive.
  2. When data arrives, if **FD_READ** is not already posted.
  3. After recv or recvfrom is called, with or without MSG_PEEK), if data is still available to receive.

  > **Note** When setsockopt SO_OOBINLINE is enabled, data includes both normal data and OOB data in the instances noted above.

- **FD_WRITE**:
  1. When **WSAAsyncSelect** called, if a send or sendto is possible.
  2. After connect or accept called, when connection established.
  3. After send or sendto fail with WSAEWOULDBLOCK, when **send** or **sendto** are likely to succeed.
  4. After bind on a connectionless socket. **FD_WRITE** may or may not occur at this time (implementation-dependent). In any case, a connectionless socket is always writable immediately after a **bind** operation.
- **FD_OOB**: Only valid when setsockopt SO_OOBINLINE is disabled (default).
  1. When **WSAAsyncSelect** called, if there is OOB data currently available to receive with the MSG_OOB flag.
  2. When OOB data arrives, if **FD_OOB** not already posted.
  3. After recv or recvfrom called with or without MSG_OOB flag, if OOB data is still available to receive.
- **FD_ACCEPT**:
  1. When **WSAAsyncSelect** called, if there is currently a connection request available to accept.
  2. When a connection request arrives, if **FD_ACCEPT** not already posted.
  3. After accept called, if there is another connection request available to accept.
- **FD_CONNECT**:
  1. When **WSAAsyncSelect** called, if there is currently a connection established.
  2. After connect called, when connection is established, even when **connect** succeeds immediately, as is typical with a datagram socket.
  3. After calling WSAJoinLeaf, when join operation completes.
  4. After connect, WSAConnect, or WSAJoinLeaf was called with a nonblocking, connection-oriented socket. The initial operation returned with a specific error of WSAEWOULDBLOCK, but the network operation went ahead. Whether the operation eventually succeeds or not, when the outcome has been determined, **FD_CONNECT** happens. The client should check the error code to determine whether the outcome was successful or failed.
- **FD_CLOSE**: Only valid on connection-oriented sockets (for example, SOCK_STREAM)
  1. When **WSAAsyncSelect** called, if socket connection has been closed.
  2. After remote system initiated graceful close, when no data currently available to receive (Be aware that, if data has been received and is waiting to be read when the remote system initiates a graceful close, the **FD_CLOSE** is not delivered until all pending data has been read).
  3. After local system initiates graceful close with shutdown and remote system has responded with "End of Data" notification (for example, TCP FIN), when no data currently available to receive.
  4. When remote system terminates connection (for example, sent TCP RST), and *lParam* will contain WSAECONNRESET error value.

  > **Note** **FD_CLOSE** is not posted after closesocket is called.

- **FD_QOS**:
  1. When **WSAAsyncSelect** called, if the quality of service associated with the socket has been changed.
  2. After WSAIoctl with SIO_GET_QOS called, when the quality of service is changed.
- **FD_GROUP_QOS**: Reserved.
- **FD_ROUTING_INTERFACE_CHANGE**:
  - After WSAIoctl with SIO_ROUTING_INTERFACE_CHANGE called, when the local interface that should be used to reach the destination specified in the IOCTL changes.
- **FD_ADDRESS_LIST_CHANGE**:
  - After WSAIoctl with SIO_ADDRESS_LIST_CHANGE called, when the list of local addresses to which the application can bind changes.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winsock.h (include Winsock2.h)
Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

WSAEventSelect

Winsock Functions

Winsock Reference

select

WSACancelAsyncRequest function (winsock.h)

10/22/2022 • 2 minutes to read • Edit Online

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

Syntax

```
int WSACancelAsyncRequest(
  [in] HANDLE hAsyncTaskHandle
);
```

Parameters

`[in] hAsyncTaskHandle`

Handle that specifies the asynchronous operation to be canceled.

Return value

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

ERROR CODE	MEANING
WSANOTINITIALISED	A successful WSAStartup call must occur before using this function.
WSAENETDOWN	The network subsystem has failed.
WSAEINVAL	Indicates that the specified asynchronous task handle was invalid.
WSAEINPROGRESS	A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function.
WSAEALREADY	The asynchronous routine being canceled has already completed.

Note It is unclear whether the application can usefully distinguish between WSAEINVAL and WSAEALREADY, since in both cases the error indicates that there is no asynchronous operation in progress with the indicated handle. (Trivial exception: zero is always an invalid asynchronous task handle.) The Windows Sockets specification does not prescribe how a conformant Windows Sockets provider should distinguish between the two cases. For maximum portability, a Windows Sockets application should treat the two errors as equivalent.

Remarks

The **WSACancelAsyncRequest** function is used to cancel an asynchronous operation that was initiated by one of the **WSAAsyncGetXByY** functions such as WSAAsyncGetHostByName. The operation to be canceled is identified by the *hAsyncTaskHandle* parameter, which should be set to the asynchronous task handle as returned by the initiating **WSAAsyncGetXByY** function.

An attempt to cancel an existing asynchronous **WSAAsyncGetXByY** operation can fail with an error code of WSAEALREADY for two reasons. First, the original operation has already completed and the application has dealt with the resultant message. Second, the original operation has already completed but the resultant message is still waiting in the application window queue.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winsock.h (include Winsock2.h)
Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

WSAAsyncGetHostByAddr

WSAAsyncGetHostByName

WSAAsyncGetProtoByName

WSAAsyncGetProtoByNumber

WSAAsyncGetServByName

WSAAsyncGetServByPort

Winsock Functions

Winsock Reference

WSACleanup function (winsock.h)

10/22/2022 • 3 minutes to read • Edit Online

The **WSACleanup** function terminates use of the Winsock 2 DLL (*Ws2_32.dll*).

Syntax

```
int WSACleanup();
```

Return value

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

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

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

Remarks

An application or DLL is required to perform a successful WSAStartup call before it can use Windows Sockets services. When it has completed the use of Windows Sockets, the application or DLL must call **WSACleanup** to deregister itself from a Windows Sockets implementation and allow the implementation to free any resources allocated on behalf of the application or DLL.

When **WSACleanup** is called, any pending blocking or asynchronous Windows Sockets calls issued by any thread in this process are canceled without posting any notification messages or without signaling any event objects. Any pending overlapped send or receive operations (WSASend, WSASendTo, WSARecv, or WSARecvFrom with an overlapped socket, for example) issued by any thread in this process are also canceled without setting the event object or invoking the completion routine, if one was specified. In this case, the pending overlapped operations fail with the error status **WSA_OPERATION_ABORTED**.

Sockets that were open when **WSACleanup** was called are reset and automatically deallocated as if closesocket were called. Sockets that have been closed with **closesocket** but that still have pending data to be sent can be affected when **WSACleanup** is called. In this case, the pending data can be lost if the WS2_32.DLL is unloaded from memory as the application exits. To ensure that all pending data is sent, an application should use shutdown to close the connection, then wait until the close completes before calling **closesocket** and **WSACleanup**. All resources and internal state, such as queued unposted or posted messages, must be deallocated so as to be available to the next user.

There must be a call to **WSACleanup** for each successful call to WSAStartup. Only the final **WSACleanup** function call performs the actual cleanup. The preceding calls simply decrement an internal reference count in the WS2_32.DLL.

> **Note** **WSACleanup** does not unregister names (peer names, for example) that may have been registered with a Windows Sockets namespace provider such as Peer Name Resolution Protocol (PNRP) namespace provider.

In Windows Sockets 1.1, attempting to call **WSACleanup** from within a blocking hook and then failing to check the return code was a common programming error. If a Winsock 1.1 application needs to quit while a blocking call is outstanding, the application has to first cancel the blocking call with WSACancelBlockingCall then issue the **WSACleanup** call once control has been returned to the application. In Windows Sockets 2, this issue does not exist and the **WSACancelBlockingCall** function has been removed.

The **WSACleanup** function typically leads to protocol-specific helper DLLs being unloaded. As a result, the **WSACleanup** function should not be called from the DllMain function in a application DLL. This can potentially cause deadlocks. For more information, please see the DLL Main Function.

Windows Phone 8: This function is supported for Windows Phone Store apps on Windows Phone 8 and later.

Windows 8.1 and **Windows Server 2012 R2**: This function is supported for Windows Store apps on Windows 8.1, Windows Server 2012 R2, and later.

Requirements

Minimum supported client	Windows 8.1, Windows Vista [desktop apps | UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps | UWP apps]
Target Platform	Windows
Header	winsock.h (include Winsock2.h)
Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

PNRP Namespace Provider API

WSAStartup

Winsock Functions

Winsock Reference

closesocket

shutdown

WSADATA structure (winsock.h)

10/22/2022 • 4 minutes to read • Edit Online

The **WSADATA** structure contains information about the Windows Sockets implementation.

Syntax

```
typedef struct WSAData {
  WORD           wVersion;
  WORD           wHighVersion;
#if ...
  unsigned short iMaxSockets;
#if ...
  unsigned short iMaxUdpDg;
#if ...
  char           *lpVendorInfo;
#if ...
  char           szDescription[WSADESCRIPTION_LEN + 1];
#if ...
  char           szSystemStatus[WSASYS_STATUS_LEN + 1];
#else
  char           szDescription[WSADESCRIPTION_LEN + 1];
#endif
#else
  char           szSystemStatus[WSASYS_STATUS_LEN + 1];
#endif
#else
  unsigned short iMaxSockets;
#endif
#else
  unsigned short iMaxUdpDg;
#endif
#else
  char           *lpVendorInfo;
#endif
} WSADATA;
```

Members

`wVersion`

Type: **WORD**

The version of the Windows Sockets specification that the *Ws2_32.dll* expects the caller to use. The high-order byte specifies the minor version number; the low-order byte specifies the major version number.

`wHighVersion`

Type: **WORD**

The highest version of the Windows Sockets specification that the *Ws2_32.dll* can support. The high-order byte specifies the minor version number; the low-order byte specifies the major version number.

This is the same value as the **wVersion** member when the version requested in the *wVersionRequested* parameter passed to the WSAStartup function is the highest version of the Windows Sockets specification that the *Ws2_32.dll* can support.

`iMaxSockets`

Type: **unsigned short**

The maximum number of sockets that may be opened. This member should be ignored for Windows Sockets version 2 and later.

The **iMaxSockets** member is retained for compatibility with Windows Sockets specification 1.1, but should not be used when developing new applications. No single value can be appropriate for all underlying service providers. The architecture of Windows Sockets changed in version 2 to support multiple providers, and the **WSADATA** structure no longer applies to a single vendor's stack.

`iMaxUdpDg`

Type: **unsigned short**

The maximum datagram message size. This member is ignored for Windows Sockets version 2 and later.

The **iMaxUdpDg** member is retained for compatibility with Windows Sockets specification 1.1, but should not be used when developing new applications. The architecture of Windows Sockets changed in version 2 to support multiple providers, and the **WSADATA** structure no longer applies to a single vendor's stack. For the actual maximum message size specific to a particular Windows Sockets service provider and socket type, applications should use getsockopt to retrieve the value of option SO_MAX_MSG_SIZE after a socket has been created.

`lpVendorInfo`

Type: **char FAR***

A pointer to vendor-specific information. This member should be ignored for Windows Sockets version 2 and later.

The **lpVendorInfo** member is retained for compatibility with Windows Sockets specification 1.1. The architecture of Windows Sockets changed in version 2 to support multiple providers, and the **WSADATA** structure no longer applies to a single vendor's stack. Applications needing to access vendor-specific configuration information should use getsockopt to retrieve the value of option PVD_CONFIG for vendor-specific information.

`szDescription[WSADESCRIPTION_LEN + 1]`

Type: **char[WSADESCRIPTION_LEN+1]**

A **NULL**-terminated ASCII string into which the *Ws2_32.dll* copies a description of the Windows Sockets implementation. The text (up to 256 characters in length) can contain any characters except control and formatting characters. The most likely use that an application would have for this member is to display it (possibly truncated) in a status message.

`szSystemStatus[WSASYS_STATUS_LEN + 1]`

Type: **char[WSASYS_STATUS_LEN+1]**

A **NULL**-terminated ASCII string into which the *Ws2_32.dll* copies relevant status or configuration information. The *Ws2_32.dll* should use this parameter only if the information might be useful to the user or support staff. This member should not be considered as an extension of the **szDescription** parameter.

Remarks

The WSAStartup function initiates the use of the Windows Sockets DLL by a process. The **WSAStartup** function returns a pointer to the

WSADATA structure in the *lpWSAData* parameter.

The current version of the Windows Sockets specification returned in the **wHighVersion** member of the **WSADATA** structure is version 2.2 encoded with the major version number in the low-byte and the minor version number in the high-byte. This version of the current Winsock DLL, *Ws2_32.dll*, supports applications that request any of the following versions of the Windows Sockets specification:

- 1.0
- 1.1
- 2.0
- 2.1
- 2.2

Depending on the version requested by the application, one of the above version numbers is the value encoded as the major version number in the low-byte and the minor version number in the high-byte that is returned in the **wVersion** member of the **WSADATA** structure.

> **Note** An application should ignore the **iMaxsockets**, **iMaxUdpDg**, and **lpVendorInfo** members in **WSADATA** if the value in **wVersion** after a successful call to WSAStartup is at least 2. This is because the architecture of Windows Sockets changed in version 2 to support multiple providers, and **WSADATA** no longer applies to a single vendor's stack. Two new socket options are introduced to supply provider-specific information: SO_MAX_MSG_SIZE (replaces the **iMaxUdpDg** member) and PVD_CONFIG (allows any other provider-specific configuration to occur).

Examples

The following example demonstrates the use of the **WSADATA** structure.

```
WORD wVersionRequested;
WSADATA wsaData;
int err;

wVersionRequested = MAKEWORD( 2, 2 );

err = WSAStartup( wVersionRequested, &wsaData );
if ( err != 0 ) {
    /* Tell the user that we could not find a usable */
    /* WinSock DLL.                                  */
    return;
}

/* Confirm that the WinSock DLL supports 2.2.*/
/* Note that if the DLL supports versions greater    */
/* than 2.2 in addition to 2.2, it will still return */
/* 2.2 in wVersion since that is the version we      */
/* requested.                                        */

if ( LOBYTE( wsaData.wVersion ) != 2 ||
        HIBYTE( wsaData.wVersion ) != 2 ) {
    /* Tell the user that we could not find a usable */
    /* WinSock DLL.                                  */
    WSACleanup( );
    return;
}

/* The WinSock DLL is acceptable. Proceed. */
```

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	winsock.h (include Winsock2.h)

See also

SOL_SOCKET Socket Options

Socket Options and IOCTLs

WSAStartup

getsockopt

WSAGetLastError function (winsock.h)

10/22/2022 • 2 minutes to read • Edit Online

The **WSAGetLastError** function returns the error status for the last Windows Sockets operation that failed.

Syntax

```
int WSAGetLastError();
```

Return value

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

Remarks

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. This extended error code can be different from the error code obtained from getsockopt when called with an *optname* parameter of **SO_ERROR**, which is socket-specific since **WSAGetLastError** is for all thread-specific sockets.

If a function call's return value indicates that error or other relevant data was returned in the error code, **WSAGetLastError** should be called immediately. This is necessary because some functions may reset the last extended error code to 0 if they succeed, overwriting the extended error code returned by a previously failed function. To specifically reset the extended error code, use the WSASetLastError function call with the *iError* parameter set to zero. A getsockopt function when called with an *optname* parameter of **SO_ERROR** also resets the extended error code to zero.

The **WSAGetLastError** function should not be used to check for an extended error value on receipt of an asynchronous message. In this case, the extended error value is passed in the *lParam* parameter of the message, and this can differ from the value returned by **WSAGetLastError**.

> **Note** An application can call the **WSAGetLastError** function to determine the extended error code for other Windows sockets functions as is normally done in Windows Sockets even if the WSAStartup function fails or the **WSAStartup** function was not called to properly initialize Windows Sockets before calling a Windows Sockets function. The **WSAGetLastError** function is one of the only functions in the Winsock 2.2 DLL that can be called in the case of a **WSAStartup** failure.

The Windows Sockets extended error codes returned by this function and the text description of the error are listed under Windows Sockets Error Codes. 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.

Windows Phone 8: This function is supported for Windows Phone Store apps on Windows Phone 8 and later.

Windows 8.1 and **Windows Server 2012 R2**: This function is supported for Windows Store apps on Windows 8.1, Windows Server 2012 R2, and later.

Requirements

Minimum supported client	Windows 8.1, Windows Vista [desktop apps | UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps | UWP apps]
Target Platform	Windows
Header	winsock.h (include Winsock2.h)
Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

Error Codes - errno, h_errno and WSAGetLastError

FormatMessage

WSASetLastError

WSAStartup

Windows Sockets Error Codes

Winsock Functions

Winsock Reference

getsockopt

WSARecvEx function (winsock.h)

10/22/2022 • 5 minutes to read • Edit Online

The **WSARecvEx** function receives data from a connected socket or a bound connectionless socket. The **WSARecvEx** function is similar to the recv function, except that the *flags* parameter is used only to return information. When a partial message is received while using datagram protocol, the MSG_PARTIAL bit is set in the *flags* parameter on return from the function.

> **Note** The **WSARecvEx** function is a Microsoft-specific extension to the Windows Sockets specification.

Syntax

```
int WSARecvEx(
  [in]      SOCKET s,
  [out]     char   *buf,
  [in]      int    len,
  [in, out] int    *flags
);
```

Parameters

`[in] s`

A descriptor that identifies a connected socket.

`[out] buf`

A pointer to the buffer to receive the incoming data.

`[in] len`

The length, in bytes, of the buffer pointed to by the *buf* parameter.

`[in, out] flags`

An indicator specifying whether the message is fully or partially received for datagram sockets.

Return value

If no error occurs, **WSARecvEx** returns the number of bytes received. If the connection has been closed, it returns zero. Additionally, if a partial message was received, the MSG_PARTIAL bit is set in the *flags* parameter. If a complete message was received, MSG_PARTIAL is not set in *flags*

Otherwise, a value of SOCKET_ERROR is returned, and a specific error code can be retrieved by calling WSAGetLastError.

> **Important** For a stream oriented-transport protocol, MSG_PARTIAL is never set on return from **WSARecvEx**. This function behaves identically to the recv function for stream-transport protocols.

ERROR CODE	MEANING
WSAECONNABORTED	The virtual circuit was terminated due to a time-out or other failure. The application should close the socket as it is no longer usable.
WSAECONNRESET	The virtual circuit was reset by the remote side executing a hard or abortive close. The application should close the socket as it is no longer usable. On a UPD-datagram socket this error would indicate that a previous send operation resulted in an ICMP "Port Unreachable" message.
WSAEFAULT	The *buf* parameter is not completely contained in a valid part of the user address space.
WSAEINPROGRESS	A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function.
WSAEINTR	The (blocking) call was canceled by the WSACancelBlockingCall call.
WSAEINVAL	The socket has not been bound with bind, or an unknown flag was specified, or MSG_OOB was specified for a socket with SO_OOBINLINE enabled or (for byte stream sockets only) *len* was zero or negative.
WSAENETDOWN	The network subsystem has failed.
WSAENETRESET	For a connection-oriented socket, this error indicates that the connection has been broken due to *keep-alive* activity that detected a failure while the operation was in progress. For a datagram socket, this error indicates that the time to live has expired.
WSAENOTCONN	The socket is not connected.
WSAENOTSOCK	The descriptor is not a socket.
WSAEOPNOTSUPP	MSG_OOB was specified, but the socket is not stream-style such as type SOCK_STREAM, OOB data is not supported in the communication domain associated with this socket, or the socket is unidirectional and supports only send operations.
WSAESHUTDOWN	The socket has been shut down; it is not possible to use WSARecvEx on a socket after shutdown has been invoked with *how* set to SD_RECEIVE or SD_BOTH.
WSAETIMEDOUT	The connection has been dropped because of a network failure or because the peer system failed to respond.

WSAEWOULDBLOCK	The socket is marked as nonblocking and the receive operation would block.
WSANOTINITIALISED	A successful WSAStartup call must occur before using this function.

Remarks

The **WSARecvEx** function that is part of the Microsoft implementation of Windows Sockets 2 is similar to the more common recv function except that the *flags* parameter is used for a single specific purpose. The *flags* parameter is used to indicate whether a partial or complete message is received when a message-oriented protocol is being used.

The value pointed to by the *flags* parameter is ignored on input. So no flags can be passed to the **WSARecvEx** function to modify its behavior. The value pointed to by the *flags* parameter is set on output. This differs from the recv and WSARecv functions where the value pointed to by the *flags* parameter on input can modify the behavior of the function.

The **WSARecvEx** and recv functions behave identically for stream-oriented protocols.

The *flags* parameter accommodates two common situations in which a partial message will be received:

- When the application's data buffer size is smaller than the message size and the message coincidentally arrives in two pieces.
- When the message is rather large and must arrive in several pieces.

The MSG_PARTIAL bit is set in the value pointed to by the *flags* parameter on return from **WSARecvEx** when a partial message was received. If a complete message was received, MSG_PARTIAL is not set in the value pointed to by the *flags* parameter.

The recv function is different from the **WSARecvEx** and WSARecv functions in that the **recv** function always receives a single message for each call for message-oriented transport protocols. The **recv** function also does not have a means to indicate to the application that the data received is only a partial message. An application must build its own protocol for checking whether a message is partial or complete by checking for the error code WSAEMSGSIZE after each call to **recv**. When the application buffer is smaller than the data being sent, as much of the message as will fit is copied into the user's buffer and **recv** returns with the error code WSAEMSGSIZE. A subsequent call to **recv** will get the next part of the message.

Applications written for message-oriented transport protocols should be coded for this possibility if message sizing is not guaranteed by the application's data transfer protocol. An application can use recv and manage the protocol itself. Alternatively, an application can use **WSARecvEx** and check that the MSG_PARTIAL bit is set in the *flags* parameter.

The **WSARecvEx** function provides the developer with a more effective way of checking whether a message received is partial or complete when a very large message arrives incrementally. For example, if an application sends a one-megabyte message, the transport protocol must break up the message in order to send it over the physical network. It is theoretically possible for the transport protocol on the receiving side to buffer all the data in the message, but this would be quite expensive in terms of resources. Instead, **WSARecvEx** can be used, minimizing overhead and eliminating the need for an application-based protocol.

> **Note** All I/O initiated by a given thread is canceled when that thread exits. For overlapped sockets, pending asynchronous operations can fail if the thread is closed before the operations complete. See the ExitThread function for more information.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winsock.h (include Mswsock.h)
Library	Mswsock.lib
DLL	Mswsock.dll

See also

WSAAsyncSelect

WSARecv

Winsock Functions

Winsock Reference

recv

recvfrom

select

send

socket

WSASetLastError function (winsock.h)

10/22/2022 • 2 minutes to read • Edit Online

The **WSASetLastError** function sets the error code that can be retrieved through the WSAGetLastError function.

Syntax

```
void WSASetLastError(
  [in] int iError
);
```

Parameters

`[in] iError`

Integer that specifies the error code to be returned by a subsequent WSAGetLastError call.

Return value

This function generates no return values.

ERROR CODE	MEANING
WSANOTINITIALISED	A successful WSAStartup call must occur before using this function.

Remarks

The **WSASetLastError** function allows an application to set the error code to be returned by a subsequent WSAGetLastError call for the current thread. Note that any subsequent Windows Sockets routine called by the application will override the error code as set by this routine.

The error code set by **WSASetLastError** is different from the error code reset by calling the function getsockopt with SO_ERROR.

The Windows Sockets error codes used by this function are listed under Windows Sockets Error Codes.

Windows Phone 8: This function is supported for Windows Phone Store apps on Windows Phone 8 and later.

Windows 8.1 and **Windows Server 2012 R2**: This function is supported for Windows Store apps on Windows 8.1, Windows Server 2012 R2, and later.

Requirements

Minimum supported client	Windows 8.1, Windows Vista [desktop apps | UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps | UWP apps]

Target Platform	Windows
Header	winsock.h (include Winsock2.h)
Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

WSAGetLastError

Windows Sockets Error Codes

Winsock Functions

Winsock Reference

getsockopt

WSAStartup function (winsock.h)

10/22/2022 • 7 minutes to read • Edit Online

The **WSAStartup** function initiates use of the Winsock DLL by a process.

Syntax

```
int WSAStartup(
        WORD      wVersionRequired,
  [out] LPWSADATA lpWSAData
);
```

Parameters

`wVersionRequired`

TBD

`[out] lpWSAData`

A pointer to the WSADATA data structure that is to receive details of the Windows Sockets implementation.

Return value

If successful, the **WSAStartup** function returns zero. Otherwise, it returns one of the error codes listed below.

The **WSAStartup** function directly returns the extended error code in the return value for this function. A call to the WSAGetLastError function is not needed and should not be used.

ERROR CODE	MEANING
WSASYSNOTREADY	The underlying network subsystem is not ready for network communication.
WSAVERNOTSUPPORTED	The version of Windows Sockets support requested is not provided by this particular Windows Sockets implementation.
WSAEINPROGRESS	A blocking Windows Sockets 1.1 operation is in progress.
WSAEPROCLIM	A limit on the number of tasks supported by the Windows Sockets implementation has been reached.
WSAEFAULT	The *lpWSAData* parameter is not a valid pointer.

Remarks

The **WSAStartup** function must be the first Windows Sockets function called by an application or DLL. It allows

an application or DLL to specify the version of Windows Sockets required and retrieve details of the specific Windows Sockets implementation. The application or DLL can only issue further Windows Sockets functions after successfully calling **WSAStartup**.

In order to support various Windows Sockets implementations and applications that can have functional differences from the latest version of Windows Sockets specification, a negotiation takes place in **WSAStartup**. The caller of **WSAStartup** passes in the *wVersionRequested* parameter the highest version of the Windows Sockets specification that the application supports. The Winsock DLL indicates the highest version of the Windows Sockets specification that it can support in its response. The Winsock DLL also replies with version of the Windows Sockets specification that it expects the caller to use.

When an application or DLL calls the **WSAStartup** function, the Winsock DLL examines the version of the Windows Sockets specification requested by the application passed in the *wVersionRequested* parameter. If the version requested by the application is equal to or higher than the lowest version supported by the Winsock DLL, the call succeeds and the Winsock DLL returns detailed information in the WSADATA structure pointed to by the *lpWSAData* parameter. The **wHighVersion** member of the **WSADATA** structure indicates the highest version of the Windows Sockets specification that the Winsock DLL supports. The **wVersion** member of the **WSADATA** structure indicates the version of the Windows Sockets specification that the Winsock DLL expects the caller to use.

If the **wVersion** member of the WSADATA structure is unacceptable to the caller, the application or DLL should call WSACleanup to release the Winsock DLL resources and fail to initialize the Winsock application. In order to support this application or DLL, it will be necessary to search for an updated version of the Winsock DLL to install on the platform.

The current version of the Windows Sockets specification is version 2.2. The current Winsock DLL, *Ws2_32.dll*, supports applications that request any of the following versions of Windows Sockets specification:

- 1.0
- 1.1
- 2.0
- 2.1
- 2.2

To get full access to the new syntax of a higher version of the Windows Sockets specification, the application must negotiate for this higher version. In this case, the *wVersionRequested* parameter should be set to request version 2.2. The application must also fully conform to that higher version of the Windows Socket specification, such as compiling against the appropriate header file, linking with a new library, or other special cases. The *Winsock2.h header* file for Winsock 2 support is included with the Microsoft Windows Software Development Kit (SDK).

Windows Sockets version 2.2 is supported on Windows Server 2008, Windows Vista, Windows Server 2003, Windows XP, Windows 2000, Windows NT 4.0 with Service Pack 4 (SP4) and later, Windows Me, Windows 98, and Windows 95 OSR2. Windows Sockets version 2.2 is also supported on
Windows 95 with the Windows Socket 2 Update. Applications on these platforms should normally request Winsock 2.2 by setting the *wVersionRequested* parameter accordingly.

On Windows 95 and versions of Windows NT 3.51 and earlier, Windows Sockets version 1.1 is the highest version of the Windows Sockets specification supported.

It is legal and possible for an application or DLL written to use a lower version of the Windows Sockets specification that is supported by the Winsock DLL to successfully negotiate this lower version using the **WSAStartup** function. For example, an application can request version 1.1 in the *wVersionRequested* parameter passed to the **WSAStartup** function on a platform with the Winsock 2.2 DLL. In this case, the application should only rely on features that fit within the version requested. New Ioctl codes, new behavior of

existing functions, and new functions should not be used. The version negotiation provided by the **WSAStartup** was primarily used to allow older Winsock 1.1 applications developed for Windows 95 and Windows NT 3.51 and earlier to run with the same behavior on later versions of Windows. The *Winsock.h header* file for Winsock 1.1 support is included with the Windows SDK.

This negotiation in the **WSAStartup** function allows both the application or DLL that uses Windows Sockets and the Winsock DLL to support a range of Windows Sockets versions. An application or DLL can use the Winsock DLL if there is any overlap in the version ranges. Detailed information on the Windows Sockets implementation is provided in the WSADATA structure returned by the **WSAStartup** function.

The following table shows how **WSAStartup** works with different applications and Winsock DLL versions.

CALLER VERSION SUPPORT	WINSOCK DLL VERSION SUPPORT	WVERSION REQUESTED	WVERSION RETURNED	WHIGHVERSION RETURNED	END RESULT
1.1	1.1	1.1	1.1	1.1	use 1.1
1.0 1.1	1.0	1.1	1.0	1.0	use 1.0
1.0	1.0 1.1	1.0	1.0	1.1	use 1.0
1.1	1.0 1.1	1.1	1.1	1.1	use 1.1
1.1	1.0	1.1	1.0	1.0	Application fails
1.0	1.1	1.0	—	—	WSAVERNOTSUPPORTED
1.0 1.1	1.0 1.1	1.1	1.1	1.1	use 1.1
1.1 2.0	1.0 1.1	2.0	1.1	1.1	use 1.1
2.0	1.0 1.1 2.0	2.0	2.0	2.0	use 2.0
2.0 2.2	1.0 1.1 2.0	2.2	2.0	2.0	use 2.0
2.2	1.0 1.1 2.0 2.1 2.2	2.2	2.2	2.2	use 2.2

Once an application or DLL has made a successful **WSAStartup** call, it can proceed to make other Windows Sockets calls as needed. When it has finished using the services of the Winsock DLL, the application must call WSACleanup to allow the Winsock DLL to free internal Winsock resources used by the application.

An application can call **WSAStartup** more than once if it needs to obtain the WSADATA structure information more than once. On each such call, the application can specify any version number supported by the Winsock DLL.

The **WSAStartup** function typically leads to protocol-specific helper DLLs being loaded. As a result, the **WSAStartup** function should not be called from the DllMain function in a application DLL. This can potentially cause deadlocks. For more information, please see the DLL Main Function.

An application must call the WSACleanup function for every successful time the **WSAStartup** function is called. This means, for example, that if an application calls **WSAStartup** three times, it must call **WSACleanup** three times. The first two calls to **WSACleanup** do nothing except decrement an internal counter; the final

WSACleanup call for the task does all necessary resource deallocation for the task.

> **Note** An application can call the WSAGetLastError function to determine the extended error code for other Windows sockets functions as is normally done in Windows Sockets even if the **WSAStartup** function fails or the **WSAStartup** function was not called to properly initialize Windows Sockets before calling a Windows Sockets function. The **WSAGetLastError** function is one of the only functions in the Winsock 2.2 DLL that can be called in the case of a **WSAStartup** failure.

Windows Phone 8: This function is supported for Windows Phone Store apps on Windows Phone 8 and later.

Windows 8.1 and **Windows Server 2012 R2**: This function is supported for Windows Store apps on Windows 8.1, Windows Server 2012 R2, and later.

Examples

The following code fragment demonstrates how an application that supports only version 2.2 of Windows Sockets makes a **WSAStartup** call:

```
#define WIN32_LEAN_AND_MEAN

#include <windows.h>
#include <winsock2.h>
#include <ws2tcpip.h>
#include <stdio.h>

// Need to link with Ws2_32.lib
#pragma comment(lib, "ws2_32.lib")


int __cdecl main()
{

    WORD wVersionRequested;
    WSADATA wsaData;
    int err;

/* Use the MAKEWORD(lowbyte, highbyte) macro declared in Windef.h */
    wVersionRequested = MAKEWORD(2, 2);

    err = WSAStartup(wVersionRequested, &wsaData);
    if (err != 0) {
        /* Tell the user that we could not find a usable */
        /* Winsock DLL.                                  */
        printf("WSAStartup failed with error: %d\n", err);
        return 1;
    }

/* Confirm that the WinSock DLL supports 2.2.*/
/* Note that if the DLL supports versions greater    */
/* than 2.2 in addition to 2.2, it will still return */
/* 2.2 in wVersion since that is the version we      */
/* requested.                                        */

    if (LOBYTE(wsaData.wVersion) != 2 || HIBYTE(wsaData.wVersion) != 2) {
        /* Tell the user that we could not find a usable */
        /* WinSock DLL.                                  */
        printf("Could not find a usable version of Winsock.dll\n");
        WSACleanup();
        return 1;
    }
    else
        printf("The Winsock 2.2 dll was found okay\n");


/* The Winsock DLL is acceptable. Proceed to use it. */

/* Add network programming using Winsock here */

/* then call WSACleanup when done using the Winsock dll */

    WSACleanup();

}
```

Requirements

Minimum supported client	Windows 8.1, Windows Vista [desktop apps | UWP apps]

Minimum supported server	Windows Server 2003 [desktop apps | UWP apps]
Target Platform	Windows
Header	winsock.h (include Winsock2.h)
Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

MAKEWORD

WSACleanup

WSAGetLastError

Winsock Functions

Winsock Reference

send

sendto

winsock2.h header

10/22/2022 • 13 minutes to read • Edit Online

This header is used by multiple technologies. For more information, see:

- Network Drivers Prior to Windows Vista
- Quality of Service (QOS)
- Windows Sockets 2

winsock2.h contains the following programming interfaces:

Functions

__WSAFDIsSet The __WSAFDIsSet function (winsock2.h) specifies whether a socket is included in a set of socket descriptors.
accept The accept function permits an incoming connection attempt on a socket.
bind The bind function associates a local address with a socket. (bind function (winsock2.h))
closesocket The closesocket function closes an existing socket. (closesocket function (winsock2.h))
connect The connect function establishes a connection to a specified socket.
FD_SET The FD_SET macro (winsock2.h) is used by Windows Sockets (Winsock) functions and service providers to place sockets into a set.
gethostbyaddr The gethostbyaddr function retrieves the host information corresponding to a network address. (gethostbyaddr is no longer recommended, use getnameinfo instead)
gethostbyname The gethostbyname function (winsock2.h) function retrieves host information corresponding to a host name from a host database.
gethostname The gethostname function (winsock2.h) retrieves the standard host name for the local computer.

GetHostNameW

The GetHostNameW function retrieves the standard host name for the local computer as a Unicode string.

getpeername

The getpeername function (winsock2.h) retrieves the address of the peer to which a socket is connected.

getprotobyname

The getprotobyname function (winsock2.h) retrieves the protocol information corresponding to a protocol name.

getprotobynumber

The getprotobynumber function (winsock2.h) retrieves protocol information corresponding to a protocol number.

getservbyname

The getservbyname function (winsock2.h) retrieves service information corresponding to a service name and protocol.

getservbyport

The getservbyport function (winsock2.h) retrieves service information corresponding to a port and protocol.

getsockname

The getsockname function (winsock2.h) retrieves the local name for the specified socket descriptor.

getsockopt

The getsockopt function retrieves the current value for a socket option associated with a socket of any type, in any state, and stores the result in optval.

htond

Converts a double from host to TCP/IP network byte order (which is big-endian).

htonf

Converts a float from host to TCP/IP network byte order (which is big-endian).

htonl

The htonl function (winsock2.h) converts a u_long from host to TCP/IP network byte order, which is big-endian.

htonll

Converts an unsigned __int64 from host to TCP/IP network byte order (which is big-endian).

htons

The htons function (winsock2.h) converts a u_short from host to TCP/IP network byte order, which is big-endian.

inet_addr

The inet_addr function (winsock2.h) converts a string containing an IPv4 dotted-decimal address into a proper address for the IN_ADDR structure.

inet_ntoa

The inet_ntoa function (winsock2.h) converts an (Ipv4) Internet network address into an ASCII string in Internet standard dotted-decimal format.

ioctlsocket

The ioctlsocket function (winsock2.h) controls the I/O mode of a socket and can be used on any socket in any state.

listen

The listen function places a socket in a state in which it is listening for an incoming connection.

ntohd

Converts an unsigned __int64 from TCP/IP network order to host byte order (which is little-endian on Intel processors) and returns a double.

ntohf

Converts an unsigned __int32 from TCP/IP network order to host byte order (which is little-endian on Intel processors) and returns a float.

ntohl

The ntohl function (winsock2.h) converts a u_long from TCP/IP network order to host byte order (which is little-endian on Intel processors).

ntohll

Converts an unsigned __int64 from TCP/IP network order to host byte order (which is little-endian on Intel processors).

ntohs

The ntohs function (winsock2.h) converts a u_short from TCP/IP network byte order to host byte order (which is little-endian on Intel processors).

ProcessSocketNotifications

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.

recv

The recv function (winsock2.h) receives data from a connected socket or a bound connectionless socket.

recvfrom

The recvfrom function receives a datagram, and stores the source address.

select

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

send

Sends data on a connected socket. (send)

sendto

The sendto function (winsock2.h) is used to write outgoing data on a socket.

setsockopt

The setsockopt function (winsock2.h) sets the current value for a socket option associated with a socket of any type, in any state.

shutdown

The shutdown function (winsock2.h) is used on all types of sockets to disable reception, transmission, or both.

socket

The socket function creates a socket that is bound to a specific transport service provider.

SocketNotificationRetrieveEvents

This inline helper function is provided as a convenience to retrieve the events mask from an OVERLAPPED_ENTRY.

WSAAccept

The WSAAccept function 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.

WSAAddressToStringA

Converts all components of a sockaddr structure into a human-readable string representation of the address. (ANSI)

WSAAddressToStringW

Converts all components of a sockaddr structure into a human-readable string representation of the address. (Unicode)

WSAAsyncGetHostByAddr

The WSAAsyncGetHostByAddr function (winsock2.h) asynchronously retrieves host information that corresponds to an address.

WSAAsyncGetHostByName

The WSAAsyncGetHostByName function (winsock2.h) asynchronously retrieves host information that corresponds to a host name.

WSAAsyncGetProtoByName

The WSAAsyncGetProtoByName function (winsock2.h) asynchronously retrieves protocol information that corresponds to a protocol name.

WSAAsyncGetProtoByNumber

The WSAAsyncGetProtoByNumber function (winsock2.h) asynchronously retrieves protocol information that corresponds to a protocol number.

WSAAsyncGetServByName

The WSAAsyncGetServByName function (winsock2.h) asynchronously retrieves service information that corresponds to a service name and port.

WSAAsyncGetServByPort

The WSAAsyncGetServByPort function (winsock2.h) asynchronously retrieves service information that corresponds to a port and protocol.

WSAAsyncSelect

The WSAAsyncSelect function (winsock2.h) requests Windows message based notification of network events for a socket.

WSACancelAsyncRequest

The WSACancelAsyncRequest function (winsock2.h) cancels an incomplete asynchronous operation.

WSACancelBlockingCall

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

WSACleanup

The WSACleanup function (winsock2.h) terminates use of the WS2_32.dll.

WSACloseEvent

The WSACloseEvent function closes an open event object handle.

WSAConnect

The WSAConnect function establishes a connection to another socket application, exchanges connect data, and specifies required quality of service based on the specified FLOWSPEC structure.

WSAConnectByList

Establishes a connection to one out of a collection of possible endpoints represented by a set of destination addresses (host names and ports).

WSAConnectByNameA

Establishes a connection to a specified host and port. (ANSI)

WSAConnectByNameW

Establishes a connection to a specified host and port. (Unicode)

WSACreateEvent

The WSACreateEvent function creates a new event object.

WSADuplicateSocketA

The WSADuplicateSocket function returns a WSAPROTOCOL_INFO structure that can be used to create a new socket descriptor for a shared socket. The WSADuplicateSocket function cannot be used on a QOS-enabled socket. (ANSI)

WSADuplicateSocketW

The WSADuplicateSocket function returns a WSAPROTOCOL_INFO structure that can be used to create a new socket descriptor for a shared socket. The WSADuplicateSocket function cannot be used on a QOS-enabled socket. (Unicode)

WSAEnumNameSpaceProvidersA

The WSAEnumNameSpaceProviders function retrieves information on available namespace providers. (ANSI)

WSAEnumNameSpaceProvidersExA

Retrieves information on available namespace providers. (ANSI)

WSAEnumNameSpaceProvidersExW

Retrieves information on available namespace providers. (Unicode)

WSAEnumNameSpaceProvidersW

The WSAEnumNameSpaceProviders function retrieves information on available namespace providers. (Unicode)

WSAEnumNetworkEvents

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

WSAEnumProtocolsA

The WSAEnumProtocols function retrieves information about available transport protocols. (ANSI)

WSAEnumProtocolsW

The WSAEnumProtocols function retrieves information about available transport protocols. (Unicode)

WSAEventSelect

The WSAEventSelect function specifies an event object to be associated with the specified set of FD_XXX network events.

WSAGetLastError

The WSAGetLastError function (winsock2.h) returns the error status for the last Windows Sockets operation that failed.

WSAGetOverlappedResult

The WSAGetOverlappedResult function retrieves the results of an overlapped operation on the specified socket.

WSAGetQOSByName

The WSAGetQOSByName function initializes a QOS structure based on a named template, or it supplies a buffer to retrieve an enumeration of the available template names.

WSAGetServiceClassInfoA

The WSAGetServiceClassInfo function retrieves the class information (schema) pertaining to a specified service class from a specified namespace provider. (ANSI)

WSAGetServiceClassInfoW

The WSAGetServiceClassInfo function retrieves the class information (schema) pertaining to a specified service class from a specified namespace provider. (Unicode)

WSAGetServiceClassNameByClassIdA

The WSAGetServiceClassNameByClassId function retrieves the name of the service associated with the specified type. This name is the generic service name, like FTP or SNA, and not the name of a specific instance of that service. (ANSI)

WSAGetServiceClassNameByClassIdW

The WSAGetServiceClassNameByClassId function retrieves the name of the service associated with the specified type. This name is the generic service name, like FTP or SNA, and not the name of a specific instance of that service. (Unicode)

WSAHtonl

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

WSAHtons

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

WSAInstallServiceClassA

The WSAInstallServiceClass function registers a service class schema within a namespace. (ANSI)

WSAInstallServiceClassW

The WSAInstallServiceClass function registers a service class schema within a namespace. (Unicode)

WSAIoctl

The WSAIoctl function controls the mode of a socket.

WSAIsBlocking

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

WSAJoinLeaf

The WSAJoinLeaf function joins a leaf node into a multipoint session, exchanges connect data, and specifies needed quality of service based on the specified FLOWSPEC structures.

WSALookupServiceBeginA

The WSALookupServiceBegin function initiates a client query that is constrained by the information contained within a WSAQUERYSET structure. (ANSI)

WSALookupServiceBeginW

The WSALookupServiceBegin function initiates a client query that is constrained by the information contained within a WSAQUERYSET structure. (Unicode)

WSALookupServiceEnd

The WSALookupServiceEnd function is called to free the handle after previous calls to WSALookupServiceBegin and WSALookupServiceNext.

WSALookupServiceNextA

The WSALookupServiceNext function is called after obtaining a handle from a previous call to WSALookupServiceBegin in order to retrieve the requested service information. (ANSI)

WSALookupServiceNextW

The WSALookupServiceNext function is called after obtaining a handle from a previous call to WSALookupServiceBegin in order to retrieve the requested service information. (Unicode)

WSANSPIoctl

Enables developers to make I/O control calls to a registered namespace.

WSANtohl

The WSANtohl function converts a u_long from network byte order to host byte order.

WSANtohs

The WSANtohs function converts a u_short from network byte order to host byte order.

WSAPoll

The WSAPoll function determines status of one or more sockets.

WSAProviderConfigChange

The WSAProviderConfigChange function notifies the application when the provider configuration is changed.

WSARecv

Receives data from a connected socket or a bound connectionless socket. (WSARecv)

WSARecvDisconnect

The WSARecvDisconnect function terminates reception on a socket, and retrieves the disconnect data if the socket is connection oriented.

WSARecvFrom

Receives a datagram and stores the source address.

WSARemoveServiceClass

The WSARemoveServiceClass function permanently removes the service class schema from the registry.

WSAResetEvent

The WSAResetEvent function resets the state of the specified event object to nonsignaled.

WSASend

Sends data on a connected socket. (WSASend)

WSASendDisconnect

The WSASendDisconnect function initiates termination of the connection for the socket and sends disconnect data.

WSASendMsg

Sends data and optional control information from connected and unconnected sockets. Note This function is a Microsoft-specific extension to the Windows Sockets specification. .

WSASendTo

Sends data to a specific destination, using overlapped I/O where applicable.

WSASetBlockingHook

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

WSASetEvent

The WSASetEvent function sets the state of the specified event object to signaled.

WSASetLastError

The WSASetLastError function (winsock2.h) sets the error code that can be retrieved through the WSAGetLastError function.

WSASetServiceA

The WSASetService function registers or removes from the registry a service instance within one or more namespaces. (ANSI)

WSASetServiceW

The WSASetService function registers or removes from the registry a service instance within one or more namespaces. (Unicode)

WSASocketA

The WSASocket function creates a socket that is bound to a specific transport-service provider. (ANSI)

WSASocketW

The WSASocket function creates a socket that is bound to a specific transport-service provider. (Unicode)

WSAStartup

The WSAStartup function (winsock2.h) initiates use of the Winsock DLL by a process.

WSAStringToAddressA

The WSAStringToAddress function converts a network address in its standard text presentation form into its numeric binary form in a sockaddr structure, suitable for passing to Windows Sockets routines that take such a structure. (ANSI)

WSAStringToAddressW

The WSAStringToAddress function converts a network address in its standard text presentation form into its numeric binary form in a sockaddr structure, suitable for passing to Windows Sockets routines that take such a structure. (Unicode)

WSAUnhookBlockingHook

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

WSAWaitForMultipleEvents

Returns when one or all of the specified event objects are in the signaled state, when the time-out interval expires, or when an I/O completion routine has executed.

Callback functions

LPWSAOVERLAPPED_COMPLETION_ROUTINE

TBD (LPWSAOVERLAPPED_COMPLETION_ROUTINE)

Structures

AFPROTOCOLS

The AFPROTOCOLS structure supplies a list of protocols to which application programmers can constrain queries. The AFPROTOCOLS structure is used for query purposes only.

BLOB

The BLOB structure (winsock2.h), derived from Binary Large Object, contains information about a block of data.

fd_set

The Fd_set structure (winsock2.h) is used by Windows Sockets (Winsock) functions and service providers to place sockets into a set.

HOSTENT

The HOSTENT structure (winsock2.h) is used by functions to store information about a given host, such as host name, IPv4 address, and so forth.

in_addr

The in_addr structure represents an IPv4 Internet address.

LINGER

The LINGER structure (winsock2.h) maintains information about a specific socket that specifies how that socket should behave when data is queued.

PROTOENT

The PROTOENT structure (winsock2.h) contains the name and protocol numbers that correspond to a given protocol name.

QOS

The QOS structure provides the means by which QOS-enabled applications can specify quality of service parameters for sent and received traffic on a particular flow.

SERVENT

The SERVENT structure (winsock2.h) is used to store or return the name and service number for a given service name.

SOCK_NOTIFY_REGISTRATION

Represents info supplied to the ProcessSocketNotifications function.

TIMEVAL

The TIMEVAL structure (winsock2.h) is used to specify a time interval. It is associated with the Berkeley Software Distribution (BSD) Time.h header file.

WSACOMPLETION

Specifies completion notification settings for I/O control calls made to a registered namespace.

WSADATA

The WSADATA structure (winsock2.h) contains information about the Windows Sockets implementation.

WSANAMESPACE_INFOA

Contains all registration information for a namespace provider. (WSANAMESPACE_INFOA)

WSANAMESPACE_INFOEXA

Contains all registration information for a namespace provider. (WSANAMESPACE_INFOEXA)

WSANAMESPACE_INFOEXW

Contains all registration information for a namespace provider. (WSANAMESPACE_INFOEXW)

WSANAMESPACE_INFOW

Contains all registration information for a namespace provider. (WSANAMESPACE_INFOW)

WSANETWORKEVENTS

The WSANETWORKEVENTS structure is used to store a socket's internal information about network events.

WSANSCLASSINFOA

The WSANSCLASSINFO structure provides individual parameter information for a specific Windows Sockets namespace. (ANSI)

WSANSCLASSINFOW

The WSANSCLASSINFO structure provides individual parameter information for a specific Windows Sockets namespace. (Unicode)

WSAOVERLAPPED

Provides a communication medium between the initiation of an overlapped I/O operation and its subsequent completion.

WSAPOLLFD

Stores socket information used by the WSAPoll function.

WSAPROTOCOL_INFOA

Used to store or retrieve complete information for a given protocol. (ANSI)

WSAPROTOCOL_INFOW

Used to store or retrieve complete information for a given protocol. (Unicode)

WSAPROTOCOLCHAIN

The WSAPROTOCOLCHAIN structure contains a counted list of Catalog Entry identifiers that comprise a protocol chain.

WSAQUERYSET2A

Provides relevant information about a given service, including service class ID, service name , applicable namespace identifier and protocol information, as well as a set of transport addresses at which the service listens. (ANSI)

WSAQUERYSET2W

Provides relevant information about a given service, including service class ID, service name , applicable namespace identifier and protocol information, as well as a set of transport addresses at which the service listens. (Unicode)

WSAQUERYSETA

Provides relevant information about a given service, including service class ID, service name, applicable namespace identifier and protocol information, as well as a set of transport addresses at which the service listens. (ANSI)

WSAQUERYSETW

Provides relevant information about a given service, including service class ID, service name, applicable namespace identifier and protocol information, as well as a set of transport addresses at which the service listens. (Unicode)

WSASERVICECLASSINFOA

The WSASERVICECLASSINFO structure contains information about a specified service class. For each service class in Windows Sockets 2, there is a single WSASERVICECLASSINFO structure. (ANSI)

WSASERVICECLASSINFOW

The WSASERVICECLASSINFO structure contains information about a specified service class. For each service class in Windows Sockets 2, there is a single WSASERVICECLASSINFO structure. (Unicode)

WSAVERSION

The WSAVERSION structure provides version comparison in Windows Sockets.

Enumerations

WSAECOMPARATOR

The Windows Sockets WSAECOMPARATOR enumeration type is used for version-comparison semantics in Windows Sockets 2.

__WSAFDIsSet function (winsock2.h)

10/22/2022 • 2 minutes to read • Edit Online

The **__WSAFDIsSet** function specifies whether a socket is included in a set of socket descriptors.

Syntax

```
int __WSAFDIsSet(
  SOCKET fd,
  fd_set *unnamedParam2
);
```

Parameters

`fd`

Descriptor identifying a socket.

`unnamedParam2`

Pointer to an fd_set structure containing the set of socket descriptors. The **__WSAFDIsSet** function determines whether the socket specified in the *fd* parameter is a member of that set.

Return value

None

Remarks

Windows Phone 8: This function is supported for Windows Phone Store apps on Windows Phone 8 and later.

Windows 8.1 and **Windows Server 2012 R2**: This function is supported for Windows Store apps on Windows 8.1, Windows Server 2012 R2, and later.

Requirements

Minimum supported client	Windows 8.1, Windows Vista [desktop apps | UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps | UWP apps]
Target Platform	Windows
Header	winsock2.h (include Winsock2.h)
Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

WSAAsyncSelect

WSAEventSelect

fd_set

select

accept function (winsock2.h)

10/22/2022 • 6 minutes to read • Edit Online

The **accept** function permits an incoming connection attempt on a socket.

Syntax

```
SOCKET WSAAPI accept(
  [in]      SOCKET   s,
  [out]     sockaddr *addr,
  [in, out] int      *addrlen
);
```

Parameters

`[in] s`

A descriptor that identifies a socket that has been placed in a listening state with the listen function. The connection is actually made with the socket that is returned by **accept**.

`[out] addr`

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

`[in, out] addrlen`

An optional pointer to an integer that contains the length of structure pointed to by the *addr* parameter.

Return value

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

Otherwise, a value of **INVALID_SOCKET** is returned, and a specific error code can be retrieved by calling WSAGetLastError.

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

ERROR CODE	MEANING
WSANOTINITIALISED	A successful WSAStartup call must occur before using this function.
WSAECONNRESET	An incoming connection was indicated, but was subsequently terminated by the remote peer prior to accepting the call.
WSAEFAULT	The *addrlen* parameter is too small or *addr* is not a valid part of the user address space.

WSAEINTR	A blocking Windows Sockets 1.1 call was canceled through WSACancelBlockingCall.
WSAEINVAL	The listen function was not invoked prior to accept.
WSAEINPROGRESS	A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function.
WSAEMFILE	The queue is nonempty upon entry to accept and there are no descriptors available.
WSAENETDOWN	The network subsystem has failed.
WSAENOBUFS	No buffer space is available.
WSAENOTSOCK	The descriptor is not a socket.
WSAEOPNOTSUPP	The referenced socket is not a type that supports connection-oriented service.
WSAEWOULDBLOCK	The socket is marked as nonblocking and no connections are present to be accepted.

Remarks

The **accept** function extracts the first connection on the queue of pending connections on socket *s*. It then creates and returns a handle to the new socket. The newly created socket is the socket that will handle the actual connection; it has the same properties as socket *s*, including the asynchronous events registered with the WSAAsyncSelect or WSAEventSelect functions.

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

The parameter *addr* is a result parameter that is filled in with the address of the connecting entity, as known to the communications layer. The exact format of the *addr* parameter is determined by the address family in which the communication is occurring. The *addrlen* is a value-result parameter; it should initially contain the amount of space pointed to by *addr*; on return it will contain the actual length (in bytes) of the address returned.

The **accept** function is used with connection-oriented socket types such as SOCK_STREAM. If *addr* and/or *addrlen* are equal to **NULL**, then no information about the remote address of the accepted socket is returned.

> **Note** When issuing a blocking Winsock call such as **accept**, Winsock may need to wait for a network event before the call can complete. Winsock performs an alertable wait in this situation, which can be interrupted by an asynchronous procedure call (APC) scheduled on the same thread. Issuing another blocking Winsock call inside an APC that interrupted an ongoing blocking Winsock call on the same thread will lead to undefined behavior, and must never be attempted by Winsock clients.

Example Code

The following example demonstrates the use of the **accept** function.

```
#ifndef UNICODE
#define UNICODE
#endif

#include <winsock2.h>
#include <stdio.h>
#include <windows.h>

// Need to link with Ws2_32.lib
#pragma comment(lib, "Ws2_32.lib")

int wmain(void)
{

    //----------------------
    // Initialize Winsock.
    WSADATA wsaData;
    int iResult = WSAStartup(MAKEWORD(2, 2), &wsaData);
    if (iResult != NO_ERROR) {
        wprintf(L"WSAStartup failed with error: %ld\n", iResult);
        return 1;
    }
    //----------------------
    // Create a SOCKET for listening for
    // incoming connection requests.
    SOCKET ListenSocket;
    ListenSocket = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
    if (ListenSocket == INVALID_SOCKET) {
        wprintf(L"socket failed with error: %ld\n", WSAGetLastError());
        WSACleanup();
        return 1;
    }
    //----------------------
    // The sockaddr_in structure specifies the address family,
    // IP address, and port for the socket that is being bound.
    sockaddr_in service;
    service.sin_family = AF_INET;
    service.sin_addr.s_addr = inet_addr("127.0.0.1");
    service.sin_port = htons(27015);

    if (bind(ListenSocket,
             (SOCKADDR *) & service, sizeof (service)) == SOCKET_ERROR) {
        wprintf(L"bind failed with error: %ld\n", WSAGetLastError());
        closesocket(ListenSocket);
        WSACleanup();
        return 1;
    }
    //----------------------
    // Listen for incoming connection requests.
    // on the created socket
    if (listen(ListenSocket, 1) == SOCKET_ERROR) {
        wprintf(L"listen failed with error: %ld\n", WSAGetLastError());
        closesocket(ListenSocket);
        WSACleanup();
        return 1;
```

```
    }
    //----------------------
    // Create a SOCKET for accepting incoming requests.
    SOCKET AcceptSocket;
    wprintf(L"Waiting for client to connect...\n");

    //----------------------
    // Accept the connection.
    AcceptSocket = accept(ListenSocket, NULL, NULL);
    if (AcceptSocket == INVALID_SOCKET) {
        wprintf(L"accept failed with error: %ld\n", WSAGetLastError());
        closesocket(ListenSocket);
        WSACleanup();
        return 1;
    } else
        wprintf(L"Client connected.\n");

    // No longer need server socket
    closesocket(ListenSocket);

    WSACleanup();
    return 0;
}
```

For another example that uses the **accept** function, see Getting Started With Winsock.

Notes for ATM

The following are important issues associated with connection setup, and must be considered when using Asynchronous Transfer Mode (ATM) with Windows Sockets 2:

- The **accept** and WSAAccept functions do not necessarily set the remote address and address length parameters. Therefore, when using ATM, the caller should use the **WSAAccept** function and place ATM_CALLING_PARTY_NUMBER_IE in the **ProviderSpecific** member of the QoS structure, which itself is included in the *lpSQOS* parameter of the callback function used in accordance with **WSAAccept**.
- When using the **accept** function, realize that the function may return before connection establishment has traversed the entire distance between sender and receiver. This is because the **accept** function returns as soon as it receives a CONNECT ACK message; in ATM, a CONNECT ACK message is returned by the next switch in the path as soon as a CONNECT message is processed (rather than the CONNECT ACK being sent by the end node to which the connection is ultimately established). As such, applications should realize that if data is sent immediately following receipt of a CONNECT ACK message, data loss is possible, since the connection may not have been established all the way between sender and receiver.

Windows 8.1 and **Windows Server 2012 R2**: This function is supported for Windows Store apps on Windows 8.1, Windows Server 2012 R2, and later.

Requirements

Minimum supported client	Windows 8.1, Windows Vista [desktop apps | UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps | UWP apps]
Target Platform	Windows
Header	winsock2.h

Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

WSAAccept

WSAAsyncSelect

Winsock Functions

Winsock Reference

bind

connect

listen

select

sockaddr

socket

AFPROTOCOLS structure (winsock2.h)

10/22/2022 • 2 minutes to read • Edit Online

The **AFPROTOCOLS** structure supplies a list of protocols to which application programmers can constrain queries. The **AFPROTOCOLS** structure is used for query purposes only.

Syntax

```
typedef struct _AFPROTOCOLS {
  INT iAddressFamily;
  INT iProtocol;
} AFPROTOCOLS, *PAFPROTOCOLS, *LPAFPROTOCOLS;
```

Members

`iAddressFamily`

Address family to which the query is to be constrained.

`iProtocol`

Protocol to which the query is to be constrained.

Remarks

The members of the **AFPROTOCOLS** structure are a functional pair, and only have meaning when used together, as protocol values have meaning only within the context of an address family.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	winsock2.h

See also

NSPLookupServiceBegin

WSALookupServiceBegin

WSAQuerySet

bind function (winsock2.h)

10/22/2022 • 9 minutes to read • Edit Online

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

Syntax

```
int WSAAPI bind(
  [in] SOCKET         s,
  [in] const sockaddr *name,
  [in] int            namelen
);
```

Parameters

`[in] s`

A descriptor identifying an unbound socket.

`[in] name`

A pointer to a sockaddr structure of the local address to assign to the bound socket .

`[in] namelen`

The length, in bytes, of the value pointed to by the *name* parameter.

Return value

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

ERROR CODE	MEANING
WSANOTINITIALISED	**Note** A successful WSAStartup call must occur before using this function.
WSAENETDOWN	The network subsystem has failed.
WSAEACCES	An attempt was made to access a socket in a way forbidden by its access permissions. This error is returned if nn attempt to bind a datagram socket to the broadcast address failed because the setsockopt option SO_BROADCAST is not enabled.

WSAEADDRINUSE	Only one usage of each socket address (protocol/network address/port) is normally permitted. This error is returned if a process on the computer is already bound to the same fully qualified address and the socket has not been marked to allow address reuse with SO_REUSEADDR. For example, the IP address and port specified in the *name* parameter are already bound to another socket being used by another application. For more information, see the SO_REUSEADDR socket option in the SOL_SOCKET Socket Options reference, Using SO_REUSEADDR and SO_EXCLUSIVEADDRUSE, and SO_EXCLUSIVEADDRUSE.
WSAEADDRNOTAVAIL	The requested address is not valid in its context. This error is returned if the specified address pointed to by the *name* parameter is not a valid local IP address on this computer.
WSAEFAULT	The system detected an invalid pointer address in attempting to use a pointer argument in a call. This error is returned if the *name* parameter is NULL, the *name* or *namelen* parameter is not a valid part of the user address space, the *namelen* parameter is too small, the *name* parameter contains an incorrect address format for the associated address family, or the first two bytes of the memory block specified by *name* do not match the address family associated with the socket descriptor *s*.
WSAEINPROGRESS	A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function.
WSAEINVAL	An invalid argument was supplied. This error is returned of the socket *s* is already bound to an address.
WSAENOBUFS	An operation on a socket could not be performed because the system lacked sufficient buffer space or because a queue was full. This error is returned of not enough buffers are available or there are too many connections.
WSAENOTSOCK	An operation was attempted on something that is not a socket. This error is returned if the descriptor in the *s* parameter is not a socket.

Remarks

The **bind** function is required on an unconnected socket before subsequent calls to the listen function. It is normally used to bind to either connection-oriented (stream) or connectionless (datagram) sockets. The **bind** function may also be used to bind to a raw socket (the socket was created by calling the socket function with the *type* parameter set to SOCK_RAW). The **bind** function may also be used on an unconnected socket before subsequent calls to the connect, ConnectEx, WSAConnect, WSAConnectByList, or WSAConnectByName

functions before send operations.

When a socket is created with a call to the socket function, it exists in a namespace (address family), but it has no name assigned to it. Use the **bind** function to establish the local association of the socket by assigning a local name to an unnamed socket.

A name consists of three parts when using the Internet address family:

- The address family.
- A host address.
- A port number that identifies the application.

In Windows Sockets 2, the *name* parameter is not strictly interpreted as a pointer to a sockaddr structure. It is cast this way for Windows Sockets 1.1 compatibility. Service providers are free to regard it as a pointer to a block of memory of size *namelen*. The first 2 bytes in this block (corresponding to the **sa_family** member of the **sockaddr** structure, the **sin_family** member of the **sockaddr_in** structure, or the **sin6_family** member of the **sockaddr_in6** structure) must contain the address family that was used to create the socket. Otherwise, an error WSAEFAULT occurs.

If an application does not care what local address is assigned, specify the constant value **INADDR_ANY** for an IPv4 local address or the constant value **in6addr_any** for an IPv6 local address in the **sa_data** member of the *name* parameter. This allows the underlying service provider to use any appropriate network address, potentially simplifying application programming in the presence of *multihomed* hosts (that is, hosts that have more than one network interface and address).

For TCP/IP, if the port is specified as zero, the service provider assigns a unique port to the application from the dynamic client port range. On Windows Vista and later, the dynamic client port range is a value between 49152 and 65535. This is a change from Windows Server 2003 and earlier where the dynamic client port range was a value between 1025 and 5000. The maximum value for the client dynamic port range can be changed by setting a value under the following registry key:

HKLM\SYSTEM\CurrentControlSet\Services\Tcpip\Parameters

The **MaxUserPort** registry value sets the value to use for the maximum value of the dynamic client port range. You must restart the computer for this setting to take effect.

On Windows Vista and later, the dynamic client port range can be viewed and changed using **netsh** commands. The dynamic client port range can be set differently for UDP and TCP and also for IPv4 and IPv6. For more information, see KB 929851.

The application can use getsockname after calling **bind** to learn the address and the port that has been assigned to the socket. If the Internet address is equal to **INADDR_ANY** or **in6addr_any**, **getsockname** cannot necessarily supply the address until the socket is connected, since several addresses can be valid if the host is multihomed. Binding to a specific port number other than port 0 is discouraged for client applications, since there is a danger of conflicting with another socket already using that port number on the local computer.

> **Note** When using **bind** with the SO_EXCLUSIVEADDRUSE or SO_REUSEADDR socket option, the socket option must be set prior to executing **bind** to have any affect. For more information, see SO_EXCLUSIVEADDRUSE and Using SO_REUSEADDR and SO_EXCLUSIVEADDRUSE.

For multicast operations, the preferred method is to call the **bind** function to associate a socket with a local IP address and then join the multicast group. Although this order of operations is not mandatory, it is strongly recommended. So a multicast application would first select an IPv4 or IPv6 address on the local computer, the wildcard IPv4 address (**INADDR_ANY**), or the wildcard IPv6 address (**in6addr_any**). The the multicast application would then call the **bind** function with this address in the in the **sa_data** member of the *name*

parameter to associate the local IP address with the socket. If a wildcard address was specified, then Windows will select the local IP address to use. After the **bind** function completes, an application would then join the multicast group of interest. For more information on how to join a multicast group, see the section on Multicast Programming. This socket can then be used to receive multicast packets from the multicast group using the recv, recvfrom, WSARecv, WSARecvEx, WSARecvFrom, or LPFN_WSARECVMSG (WSARecvMsg) functions.

The **bind** function is not normally required for send operations to a multicast group. The sendto,WSASendMsg, and WSASendTo functions implicitly bind the socket to the wildcard address if the socket is not already bound. The **bind** function is required before the use of the send or WSASend functions which do not perform an implicit bind and are allowed only on connected sockets, which means the socket must have already been bound for it to be connected. The **bind** function might be used before send operations using the **sendto**,**WSASendMsg**, or **WSASendTo** functions if an application wanted to select a specific local IP address on a local computer with multiple network interfaces and local IP addresses. Otherwise an implicit bind to the wildcard address using the **sendto**,**WSASendMsg** , or **WSASendTo** functions might result in a different local IP address being used for send operations.

> **Note** When issuing a blocking Winsock call such as **bind**, Winsock may need to wait for a network event before the call can complete. Winsock performs an alertable wait in this situation, which can be interrupted by an asynchronous procedure call (APC) scheduled on the same thread. Issuing another blocking Winsock call inside an APC that interrupted an ongoing blocking Winsock call on the same thread will lead to undefined behavior, and must never be attempted by Winsock clients.

Notes for IrDA Sockets

- The Af_irda.h header file must be explicitly included.
- Local names are not exposed in IrDA. IrDA client sockets therefore, must never call the **bind** function before the connect function. If the IrDA socket was previously bound to a service name using **bind**, the **connect** function will fail with SOCKET_ERROR.
- If the service name is of the form "LSAP-SELxxx," where xxx is a decimal integer in the range 1-127, the address indicates a specific LSAP-SEL xxx rather than a service name. Service names such as these allow server applications to accept incoming connections directed to a specific LSAP-SEL, without first performing an ISA service name query to get the associated LSAP-SEL. One example of this service name type is a non-Windows device that does not support IAS.

Windows Phone 8: This function is supported for Windows Phone Store apps on Windows Phone 8 and later.

Windows 8.1 and **Windows Server 2012 R2**: This function is supported for Windows Store apps on Windows 8.1, Windows Server 2012 R2, and later.

Examples

The following example demonstrates the use of the **bind** function. For another example that uses the **bind** function, see Getting Started With Winsock.

```
#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 main()
```

```
{

    // Declare some variables
    WSADATA wsaData;

    int iResult = 0;            // used to return function results

    // the listening socket to be created
    SOCKET ListenSocket = INVALID_SOCKET;

    // The socket address to be passed to bind
    sockaddr_in service;

    //----------------------
    // Initialize Winsock
    iResult = WSAStartup(MAKEWORD(2, 2), &wsaData);
    if (iResult != NO_ERROR) {
        wprintf(L"Error at WSAStartup()\n");
        return 1;
    }
    //----------------------
    // Create a SOCKET for listening for
    // incoming connection requests
    ListenSocket = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
    if (ListenSocket == INVALID_SOCKET) {
        wprintf(L"socket function failed with error: %u\n", WSAGetLastError());
        WSACleanup();
        return 1;
    }
    //----------------------
    // The sockaddr_in structure specifies the address family,
    // IP address, and port for the socket that is being bound.
    service.sin_family = AF_INET;
    service.sin_addr.s_addr = inet_addr("127.0.0.1");
    service.sin_port = htons(27015);

    //----------------------
    // Bind the socket.
    iResult = bind(ListenSocket, (SOCKADDR *) &service, sizeof (service));
    if (iResult == SOCKET_ERROR) {
        wprintf(L"bind failed with error %u\n", WSAGetLastError());
        closesocket(ListenSocket);
        WSACleanup();
        return 1;
    }
    else
        wprintf(L"bind returned success\n");

    WSACleanup();
    return 0;
}
```

Requirements

Minimum supported client	Windows 8.1, Windows Vista [desktop apps | UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps | UWP apps]
Target Platform	Windows

Header	winsock2.h (include Winsock2.h)
Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

Multicast Programming

SOL_SOCKET Socket Options

SO_EXCLUSIVEADDRUSE

TCP/IP Raw Sockets

Using SO_REUSEADDR and SO_EXCLUSIVEADDRUSE

WSACancelBlockingCall

Winsock Functions

Winsock Reference

connect

getsockname

listen

setsockopt

sockaddr

socket

BLOB structure (winsock2.h)

10/22/2022 • 2 minutes to read • Edit Online

The **BLOB** structure, derived from Binary Large Object, contains information about a block of data.

Syntax

```
typedef struct _BLOB {
  ULONG cbSize;
#if ...
  BYTE  *pBlobData;
#else
  BYTE  *pBlobData;
#endif
} BLOB, *LPBLOB;
```

Members

`cbSize`

Size of the block of data pointed to by **pBlobData**, in bytes.

`pBlobData`

Pointer to a block of data.

Remarks

The structure name **BLOB** comes from the acronym BLOB, which stands for Binary Large Object.

This structure does not describe the nature of the data pointed to by **pBlobData**.

> **Note** Windows Sockets defines a similar **BLOB** structure in Wtypes.h. Using both header files in the same source code file creates redefinition–compile time errors.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	winsock2.h (include Wtypes.h, Nspapi.h, Winsock2.h, Wtypes.h, Nspapi.h, Winsock2.h)

See also

Bluetooth and BLOB

closesocket function (winsock2.h)

10/22/2022 • 9 minutes to read • Edit Online

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

Syntax

```
int WSAAPI closesocket(
  [in] SOCKET s
);
```

Parameters

`[in] s`

A descriptor identifying the socket to close.

Return value

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

ERROR CODE	MEANING
WSANOTINITIALISED	A successful WSAStartup call must occur before using this function.
WSAENETDOWN	The network subsystem has failed.
WSAENOTSOCK	The descriptor is not a socket.
WSAEINPROGRESS	A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function.
WSAEINTR	The (blocking) Windows Socket 1.1 call was canceled through WSACancelBlockingCall.
WSAEWOULDBLOCK	The socket is marked as nonblocking, but the **l_onoff** member of the linger structure is set to nonzero and the **l_linger** member of the **linger** structure is set to a nonzero timeout value.

Remarks

The **closesocket** function closes a socket. Use it to release the socket descriptor passed in the *s* parameter. Note that the socket descriptor passed in the *s* parameter may immediately be reused by the system as soon as **closesocket** function is issued. As a result, it is not reliable to expect further references to the socket descriptor

passed in the *s* parameter to fail with the error WSAENOTSOCK. A Winsock client must never issue **closesocket** on *s* concurrently with another Winsock function call.

Any pending overlapped send and receive operations (WSASend/ WSASendTo/ WSARecv/ WSARecvFrom with an overlapped socket) issued by any thread in this process are also canceled. Any event, completion routine, or completion port action specified for these overlapped operations is performed. The pending overlapped operations fail with the error status WSA_OPERATION_ABORTED.

An application should not assume that any outstanding I/O operations on a socket will all be guaranteed to completed when **closesocket** returns. The **closesocket** function will initiate cancellation on the outstanding I/O operations, but that does not mean that an application will receive I/O completion for these I/O operations by the time the **closesocket** function returns. Thus, an application should not cleanup any resources (WSAOVERLAPPED structures, for example) referenced by the outstanding I/O requests until the I/O requests are indeed completed.

An application should always have a matching call to **closesocket** for each successful call to socket to return any socket resources to the system.

The linger structure maintains information about a specific socket that specifies how that socket should behave when data is queued to be sent and the **closesocket** function is called on the socket.

The **l_onoff** member of the **linger** structure determines whether a socket should remain open for a specified amount of time after a **closesocket** function call to enable queued data to be sent. This member can be modified in two ways:

- Call the setsockopt function with the *optname* parameter set to **SO_DONTLINGER**. The *optval* parameter determines how the **l_onoff** member is modified.
- Call the setsockopt function with the *optname* parameter set to **SO_LINGER**. The *optval* parameter specifies how both the **l_onoff** and **l_linger** members are modified.

The **l_linger** member of the **linger** structure determines the amount of time, in seconds, a socket should remain open. This member is only applicable if the **l_onoff** member of the **linger** structure is nonzero.

The default parameters for a socket are the **l_onoff** member of the **linger** structure is zero, indicating that the socket should not remain open. The default value for the **l_linger** member of the **linger** structure is zero, but this value is ignored when the **l_onoff** member is set to zero.

To enable a socket to remain open, an application should set the **l_onoff** member to a nonzero value and set the **l_linger** member to the desired timeout in seconds. To disable a socket from remaining open, an application only needs to set the **l_onoff** member of the **linger** structure to zero.

If an application calls the setsockopt function with the *optname* parameter set to **SO_DONTLINGER** to set the **l_onoff** member to a nonzero value, the value for the **l_linger** member is not specified. In this case, the timeout used is implementation dependent. If a previous timeout has been established for a socket (by previously calling the **setsockopt** function with the *optname* parameter set to **SO_LINGER**), this timeout value should be reinstated by the service provider.

The semantics of the **closesocket** function are affected by the socket options that set members of **linger** structure.

L_ONOFF	L_LINGER	TYPE OF CLOSE	WAIT FOR CLOSE?
zero	Do not care	Graceful close	No
nonzero	zero	Hard	No

nonzero	nonzero	Graceful if all data is sent within timeout value specified in the **l_linger** member. Hard if all data could not be sent within timeout value specified in the **l_linger** member.	Yes

If the **l_onoff** member of the LINGER structure is zero on a stream socket, the **closesocket** call will return immediately and does not receive WSAEWOULDBLOCK whether the socket is blocking or nonblocking. However, any data queued for transmission will be sent, if possible, before the underlying socket is closed. This is also called a graceful disconnect or close. In this case, the Windows Sockets provider cannot release the socket and other resources for an arbitrary period, thus affecting applications that expect to use all available sockets. This is the default behavior for a socket.

If the **l_onoff** member of the linger structure is nonzero and **l_linger** member is zero, **closesocket** is not blocked even if queued data has not yet been sent or acknowledged. This is called a hard or abortive close, because the socket's virtual circuit is reset immediately, and any unsent data is lost. On Windows, any **recv** call on the remote side of the circuit will fail with WSAECONNRESET.

If the **l_onoff** member of the linger structure is set to nonzero and **l_linger** member is set to a nonzero timeout on a blocking socket, the **closesocket** call blocks until the remaining data has been sent or until the timeout expires. This is called a graceful disconnect or close if all of the data is sent within timeout value specified in the **l_linger** member. If the timeout expires before all data has been sent, the Windows Sockets implementation terminates the connection before **closesocket** returns and this is called a hard or abortive close.

Setting the **l_onoff** member of the linger structure to nonzero and the **l_linger** member with a nonzero timeout interval on a nonblocking socket is not recommended. In this case, the call to **closesocket** will fail with an error of WSAEWOULDBLOCK if the close operation cannot be completed immediately. If **closesocket** fails with WSAEWOULDBLOCK the socket handle is still valid, and a disconnect is not initiated. The application must call **closesocket** again to close the socket.

If the **l_onoff** member of the linger structure is nonzero and the **l_linger** member is a nonzero timeout interval on a blocking socket, the result of the **closesocket** function can't be used to determine whether all data has been sent to the peer. If the data is sent before the timeout specified in the **l_linger** member expires or if the connection was aborted, the **closesocket** function won't return an error code (the return value from the **closesocket** function is zero).

The **closesocket** call will only block until all data has been delivered to the peer or the timeout expires. If the connection is reset because the timeout expires, then the socket will not go into TIME_WAIT state. If all data is sent within the timeout period, then the socket can go into TIME_WAIT state.

If the **l_onoff** member of the linger structure is nonzero and the **l_linger** member is a zero timeout interval on a blocking socket, then a call to **closesocket** will reset the connection. The socket will not go to the TIME_WAIT state.

The getsockopt function can be called with the *optname* parameter set to **SO_LINGER** to retrieve the current value of the **linger** structure associated with a socket.

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

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

- If the **l_onoff** member of the LINGER structure is zero (the default for a socket), **closesocket** returns immediately and the connection is gracefully closed in the background.
- If the **l_onoff** member of the linger structure is set to nonzero and the **l_linger** member is set to zero (no timeout) **closesocket** returns immediately and the connection is reset or terminated.
- If the **l_onoff** member of the linger structure is set to nonzero and the **l_linger** member is set to a nonzero timeout:– For a blocking socket, **closesocket** blocks until all data is sent or the timeout expires.

  – For a nonblocking socket, **closesocket** returns immediately indicating failure.

For additional information please see Graceful Shutdown, Linger Options, and Socket Closure for more information.

> **Note** When issuing a blocking Winsock call such as **closesocket**, Winsock may need to wait for a network event before the call can complete. Winsock performs an alertable wait in this situation, which can be interrupted by an asynchronous procedure call (APC) scheduled on the same thread. Issuing another blocking Winsock call inside an APC that interrupted an ongoing blocking Winsock call on the same thread will lead to undefined behavior, and must never be attempted by Winsock clients.

Notes for IrDA Sockets

Keep the following in mind:

- The Af_irda.h header file must be explicitly included.
- The standard linger options are supported.
- Although IrDA does not provide a graceful close, IrDA will defer closing until receive queues are purged. Thus, an application can send data and immediately call the socket function, and be confident that the receiver will copy the data before receiving an FD_CLOSE message.

Notes for ATM

The following are important issues associated with connection teardown when using Asynchronous Transfer Mode (ATM) and Windows Sockets 2:

- Using the **closesocket** or shutdown functions with SD_SEND or SD_BOTH results in a RELEASE signal being sent out on the control channel. Due to ATM's use of separate signal and data channels, it is possible that a RELEASE signal could reach the remote end before the last of the data reaches its destination, resulting in a loss of that data. One possible solutions is programming a sufficient delay between the last data sent and the **closesocket** or shutdown function calls for an ATM socket.
- Half close is not supported by ATM.
- Both abortive and graceful disconnects result in a RELEASE signal being sent out with the same cause field. In either case, received data at the remote end of the socket is still delivered to the application. See Graceful Shutdown, Linger Options, and Socket Closure for more information.

Windows Phone 8: This function is supported for Windows Phone Store apps on Windows Phone 8 and later.

Windows 8.1 and **Windows Server 2012 R2**: This function is supported for Windows Store apps on Windows 8.1, Windows Server 2012 R2, and later.

Requirements

Minimum supported client	Windows 8.1, Windows Vista [desktop apps | UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps | UWP apps]
Target Platform	Windows
Header	winsock2.h (include Winsock2.h)
Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

Graceful Shutdown, Linger Options, and Socket Closure

WSAAsyncSelect

WSADuplicateSocket

WSAOVERLAPPED

Winsock Functions

Winsock Reference

accept

getsockopt

ioctlsocket

linger

setsockopt

socket

connect function (winsock2.h)

10/22/2022 • 8 minutes to read • Edit Online

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

Syntax

```
int WSAAPI connect(
  [in] SOCKET         s,
  [in] const sockaddr *name,
  [in] int            namelen
);
```

Parameters

`[in] s`

A descriptor identifying an unconnected socket.

`[in] name`

A pointer to the sockaddr structure to which the connection should be established.

`[in] namelen`

The length, in bytes, of the sockaddr structure pointed to by the *name* parameter.

Return value

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

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

With a nonblocking socket, the connection attempt cannot be completed immediately. In this case, **connect** will return SOCKET_ERROR, and WSAGetLastError will return WSAEWOULDBLOCK. In this case, there are three possible scenarios:

- Use the select function to determine the completion of the connection request by checking to see if the socket is writable.
- If the application is using WSAAsyncSelect to indicate interest in connection events, then the application will receive an FD_CONNECT notification indicating that the **connect** operation is complete (successfully or not).
- If the application is using WSAEventSelect to indicate interest in connection events, then the associated event object will be signaled indicating that the **connect** operation is complete (successfully or not).

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

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

ERROR CODE	MEANING
WSANOTINITIALISED	A successful WSAStartup call must occur before using this function.
WSAENETDOWN	The network subsystem has failed.
WSAEADDRINUSE	The socket's local address is already in use and the socket was not marked to allow address reuse with SO_REUSEADDR. This error usually occurs when executing bind, but could be delayed until the connect function if the **bind** was to a wildcard address (**INADDR_ANY** or **in6addr_any**) for the local IP address. A specific address needs to be implicitly bound by the **connect** function.
WSAEINTR	The blocking Windows Socket 1.1 call was canceled through WSACancelBlockingCall.
WSAEINPROGRESS	A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function.
WSAEALREADY	A nonblocking connect call is in progress on the specified socket. **Note** In order to preserve backward compatibility, this error is reported as WSAEINVAL to Windows Sockets 1.1 applications that link to either Winsock.dll or Wsock32.dll.
WSAEADDRNOTAVAIL	The remote address is not a valid address (such as **INADDR_ANY** or **in6addr_any**) .
WSAEAFNOSUPPORT	Addresses in the specified family cannot be used with this socket.
WSAECONNREFUSED	The attempt to connect was forcefully rejected.
WSAEFAULT	The sockaddr structure pointed to by the *name* contains incorrect address format for the associated address family or the *namelen* parameter is too small. This error is also returned if the **sockaddr** structure pointed to by the *name* parameter with a length specified in the *namelen* parameter is not in a valid part of the user address space.
WSAEINVAL	The parameter *s* is a listening socket.

WSAEISCONN	The socket is already connected (connection-oriented sockets only).
WSAENETUNREACH	The network cannot be reached from this host at this time.
WSAEHOSTUNREACH	A socket operation was attempted to an unreachable host.
WSAENOBUFS	**Note** No buffer space is available. The socket cannot be connected.
WSAENOTSOCK	The descriptor specified in the *s* parameter is not a socket.
WSAETIMEDOUT	An attempt to connect timed out without establishing a connection.
WSAEWOULDBLOCK	The socket is marked as nonblocking and the connection cannot be completed immediately.
WSAEACCES	An attempt to connect a datagram socket to broadcast address failed because setsockopt option SO_BROADCAST is not enabled.

Remarks

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

For connection-oriented sockets (for example, type SOCK_STREAM), an active connection is initiated to the foreign host using *name* (an address in the namespace of the socket; for a detailed description, see bind and sockaddr).

> **Note** If a socket is opened, a setsockopt call is made, and then a sendto call is made, Windows Sockets performs an implicit bind function call.

When the socket call completes successfully, the socket is ready to send and receive data. If the address member of the structure specified by the *name* parameter is filled with zeros, **connect** will return the error WSAEADDRNOTAVAIL. Any attempt to reconnect an active connection will fail with the error code WSAEISCONN.

For connection-oriented, nonblocking sockets, it is often not possible to complete the connection immediately. In such a case, this function returns the error WSAEWOULDBLOCK. However, the operation proceeds.

When the success or failure outcome becomes known, it may be reported in one of two ways, depending on how the client registers for notification.

- If the client uses the select function, success is reported in the writefds set and failure is reported in the

exceptfds set.

- If the client uses the functions WSAAsyncSelect or WSAEventSelect, the notification is announced with FD_CONNECT and the error code associated with the FD_CONNECT indicates either success or a specific reason for failure.

If the connection is not completed immediately, the client should wait for connection completion before attempting to set socket options using setsockopt. Calling setsockopt while a connection is in progress is not supported.

For a connectionless socket (for example, type SOCK_DGRAM), the operation performed by **connect** is merely to establish a default destination address that can be used on subsequent send/ WSASend and recv/ WSARecv calls. Any datagrams received from an address other than the destination address specified will be discarded. If the address member of the structure specified by *name* is filled with zeros, the socket will be disconnected. Then, the default remote address will be indeterminate, so send/ WSASend and recv/ WSARecv calls will return the error code WSAENOTCONN. However, sendto/ WSASendTo and recvfrom/ WSARecvFrom can still be used. The default destination can be changed by simply calling **connect** again, even if the socket is already connected. Any datagrams queued for receipt are discarded if *name* is different from the previous **connect**.

For connectionless sockets, *name* can indicate any valid address, including a broadcast address. However, to connect to a broadcast address, a socket must use setsockopt to enable the SO_BROADCAST option. Otherwise, **connect** will fail with the error code WSAEACCES.

When a connection between sockets is broken, the socket that was connected should be discarded and new socket should be created. When a problem develops on a connected socket, the application must discard the socket and create the socket again in order to return to a stable point.

> **Note** When issuing a blocking Winsock call such as **connect**, Winsock may need to wait for a network event before the call can complete. Winsock performs an alertable wait in this situation, which can be interrupted by an asynchronous procedure call (APC) scheduled on the same thread. Issuing another blocking Winsock call inside an APC that interrupted an ongoing blocking Winsock call on the same thread will lead to undefined behavior, and must never be attempted by Winsock clients.

Example Code

The following example demonstrates the use of the **connect** function.

```
#ifndef UNICODE
#define UNICODE
#endif

#define WIN32_LEAN_AND_MEAN

#include <winsock2.h>
#include <ws2tcpip.h>
#include <stdio.h>

// Need to link with Ws2_32.lib
#pragma comment(lib, "ws2_32.lib")

int wmain()
{
    //-----------------------
    // Initialize Winsock
    WSADATA wsaData;
    int iResult = WSAStartup(MAKEWORD(2, 2), &wsaData);
    if (iResult != NO_ERROR) {
        wprintf(L"WSAStartup function failed with error: %d\n", iResult);
        return 1;
    }
    //-----------------------
    // Create a SOCKET for connecting to server
    SOCKET ConnectSocket;
    ConnectSocket = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
    if (ConnectSocket == INVALID_SOCKET) {
        wprintf(L"socket function failed with error: %ld\n", WSAGetLastError());
        WSACleanup();
        return 1;
    }
    //-----------------------
    // The sockaddr_in structure specifies the address family,
    // IP address, and port of the server to be connected to.
    sockaddr_in clientService;
    clientService.sin_family = AF_INET;
    clientService.sin_addr.s_addr = inet_addr("127.0.0.1");
    clientService.sin_port = htons(27015);

    //-----------------------
    // Connect to server.
    iResult = connect(ConnectSocket, (SOCKADDR *) & clientService, sizeof (clientService));
    if (iResult == SOCKET_ERROR) {
        wprintf(L"connect function failed with error: %ld\n", WSAGetLastError());
        iResult = closesocket(ConnectSocket);
        if (iResult == SOCKET_ERROR)
            wprintf(L"closesocket function failed with error: %ld\n", WSAGetLastError());
        WSACleanup();
        return 1;
    }

    wprintf(L"Connected to server.\n");

    iResult = closesocket(ConnectSocket);
    if (iResult == SOCKET_ERROR) {
        wprintf(L"closesocket function failed with error: %ld\n", WSAGetLastError());
        WSACleanup();
        return 1;
    }

    WSACleanup();
    return 0;
}
```

For another example that uses the **connect** function, see Getting Started With Winsock.

Notes for IrDA Sockets

- The Af_irda.h header file must be explicitly included.
- If an existing IrDA connection is detected at the media-access level, WSAENETDOWN is returned.
- If active connections to a device with a different address exist, WSAEADDRINUSE is returned.
- If the socket is already connected or an exclusive/multiplexed mode change failed, WSAEISCONN is returned.
- If the socket was previously bound to a local service name to accept incoming connections using bind, WSAEINVAL is returned. Note that once a socket is bound, it cannot be used for establishing an outbound connection.

IrDA implements the connect function with addresses of the form sockaddr_irda. Typically, a client application will create a socket with the socket function, scan the immediate vicinity for IrDA devices with the IRLMP_ENUMDEVICES socket option, choose a device from the returned list, form an address, and then call **connect**. There is no difference between blocking and nonblocking semantics.

Requirements

Minimum supported client	Windows Vista [desktop apps | UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps | UWP apps]
Target Platform	Windows
Header	winsock2.h
Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

WSAAsyncSelect

WSAConnect

ConnectEx

Winsock Functions

Winsock Reference

accept

bind

getsockname

select

sockaddr

socket

fd_set structure (winsock2.h)

10/22/2022 • 2 minutes to read • Edit Online

The **fd_set** structure is used by various Windows Sockets functions and service providers, such as the select function, to place sockets into a "set" for various purposes, such as testing a given socket for readability using the *readfds* parameter of the **select** function.

Syntax

```
typedef struct fd_set {
  u_int  fd_count;
  SOCKET fd_array[FD_SETSIZE];
} fd_set, FD_SET, *PFD_SET, *LPFD_SET;
```

Members

`fd_count`

The number of sockets in the set.

`fd_array[FD_SETSIZE]`

An array of sockets that are in the set.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	winsock2.h (include Winsock2.h, Winsock.h)

See also

WSAAsyncSelect

WSAEventSelect

select

FD_SET macro (winsock2.h)

10/22/2022 • 2 minutes to read • Edit Online

The **fd_set** structure is used by various Windows Sockets functions and service providers, such as the select function, to place sockets into a "set" for various purposes, such as testing a given socket for readability using the *readfds* parameter of the **select** function.

Syntax

```
void FD_SET(
   fd,
   set
);
```

Parameters

`fd`

`set`

Return value

None

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winsock2.h (include Winsock2.h, Winsock.h)

See also

WSAAsyncSelect

WSAEventSelect

select

gethostbyaddr function (winsock2.h)

10/22/2022 • 4 minutes to read • Edit Online

[**gethostbyaddr** is no longer recommended for use as of Windows Sockets 2. Instead, use getnameinfo.]

The **gethostbyaddr** function retrieves the host information corresponding to a network address.

Syntax

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

Parameters

`addr`

TBD

`len`

TBD

`type`

TBD

Return value

If no error occurs, **gethostbyaddr** returns a pointer to the hostent structure. Otherwise, it returns a null pointer, and a specific error code can be retrieved by calling WSAGetLastError.

ERROR CODE	MEANING
WSANOTINITIALISED	A successful WSAStartup call must occur before using this function.
WSAEINVAL	An invalid argument was supplied. This error is returned if **AF_INET6** was specified in the *type* parameter and the *len* parameter was not set equal to the size of an IPv6 address.
WSAENETDOWN	The network subsystem has failed.
WSAHOST_NOT_FOUND	Authoritative answer host not found.
WSATRY_AGAIN	Nonauthoritative host not found, or server failed.

WSANO_RECOVERY	A nonrecoverable error occurred.
WSANO_DATA	Valid name, no data record of requested type.
WSAEINPROGRESS	A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function.
WSAEAFNOSUPPORT	The *type* specified is not supported by the Windows Sockets implementation.
WSAEFAULT	The *addr* parameter is not a valid part of the user address space, or the *len* parameter is too small.
WSAEINTR	A blocking Windows Socket 1.1 call was canceled through WSACancelBlockingCall.

Remarks

The **gethostbyaddr** function returns a pointer to the hostent structure that contains the name and address corresponding to the given network address.

The memory for the hostent structure returned by the **gethostbyaddr** function is allocated internally by the Winsock DLL from thread local storage. Only a single **hostent** structure is allocated and used, no matter how many times the **gethostbyaddr** or gethostbyname functions are called on the thread. The returned **hostent** structure must be copied to an application buffer if additional calls are to be made to the **gethostbyaddr** or **gethostbyname** functions on the same thread. Otherwise, the return value will be overwritten by subsequent **gethostbyaddr** or **gethostbyname** calls on the same thread. The internal memory allocated for the returned **hostent** structure is released by the Winsock DLL when the thread exits.

An application should not try to release the memory used by the returned hostent structure. The application must never attempt to modify this structure or to free any of its components. Furthermore, only one copy of this structure is allocated per thread, so the application should copy any information it needs before issuing any other function calls to **gethostbyaddr** or gethostbyname.

Although **gethostbyaddr** no longer recommended for use as of Windows Sockets 2 and the getnameinfo function should be used, **gethostbyaddr** is capable of returning a NetBIOS name; **getnameinfo** is not. Developers requiring NetBIOS name resolution may need to use **gethostbyaddr** until their applications are completely independent of NetBIOS names.

> **Note** The capability to perform reverse lookups using the **gethostbyaddr** function is convenient, but such lookups are considered inherently unreliable, and should be used only as a hint.

Example Code

The following example demonstrates the use of the **gethostbyaddr** function.

```
#include <winsock2.h>
#include <ws2tcpip.h>
#include <stdio.h>
```

```
int main(int argc, char **argv)
{

    //-----------------------------------------
    // Declare and initialize variables
    WSADATA wsaData;
    int iResult;

    DWORD dwError;
    int i = 0;
    int bIpv6 = 0;

    struct hostent *remoteHost;
    char *host_addr;
    struct in_addr addr = { 0 };
    IN6_ADDR addr6;

    char **pAlias;

    // Validate the parameters
    if (argc < 2) {
        printf("usage: %s 4 ipv4address\n", argv[0]);
        printf(" or\n");
        printf("usage: %s 6 ipv6address\n", argv[0]);
        printf("  to return the hostname\n");
        printf("       %s 4 127.0.0.1\n", argv[0]);
        printf("       %s 6 0::1\n", argv[0]);
        return 1;
    }
    // Validate parameters
    if (atoi(argv[1]) == 4)
        bIpv6 = 0;
    else if (atoi(argv[1]) == 6)
        bIpv6 = 1;
    else {
        printf("usage: %s 4 ipv4address\n", argv[0]);
        printf(" or\n");
        printf("usage: %s 6 ipv6address\n", argv[0]);
        printf("  to return the hostname\n");
        printf("       %s 4 127.0.0.1\n", argv[0]);
        printf("       %s 6 0::1\n", argv[0]);
        return 1;
    }

    // Initialize Winsock
    iResult = WSAStartup(MAKEWORD(2, 2), &wsaData);
    if (iResult != 0) {
        printf("WSAStartup failed: %d\n", iResult);
        return 1;
    }

    host_addr = argv[2];

    printf("Calling gethostbyaddr with %s\n", host_addr);
    if (bIpv6 == 1) {
        {
            iResult = inet_pton(AF_INET6, host_addr, &addr6);
            if (iResult == 0) {
                printf("The IPv6 address entered must be a legal address\n");
                return 1;
            } else
                remoteHost = gethostbyaddr((char *) &addr6, 16, AF_INET6);
        }
    } else {
        addr.s_addr = inet_addr(host_addr);
        if (addr.s_addr == INADDR_NONE) {
            printf("The IPv4 address entered must be a legal address\n");
            return 1;
        } else
```

```
            remoteHost = gethostbyaddr((char *) &addr, 4, AF_INET);
    }

    if (remoteHost == NULL) {
        dwError = WSAGetLastError();
        if (dwError != 0) {
            if (dwError == WSAHOST_NOT_FOUND) {
                printf("Host not found\n");
                return 1;
            } else if (dwError == WSANO_DATA) {
                printf("No data record found\n");
                return 1;
            } else {
                printf("Function failed with error: %ld\n", dwError);
                return 1;
            }
        }
    } else {
        printf("Function returned:\n");
        printf("\tOfficial name: %s\n", remoteHost->h_name);
        for (pAlias = remoteHost->h_aliases; *pAlias != 0; pAlias++) {
            printf("\tAlternate name #%d: %s\n", ++i, *pAlias);
        }
        printf("\tAddress type: ");
        switch (remoteHost->h_addrtype) {
        case AF_INET:
            printf("AF_INET\n");
            break;
        case AF_INET6:
            printf("AF_INET6\n");
            break;
        case AF_NETBIOS:
            printf("AF_NETBIOS\n");
            break;
        default:
            printf(" %d\n", remoteHost->h_addrtype);
            break;
        }
        printf("\tAddress length: %d\n", remoteHost->h_length);

        if (remoteHost->h_addrtype == AF_INET) {
            while (remoteHost->h_addr_list[i] != 0) {
                addr.s_addr = *(u_long *) remoteHost->h_addr_list[i++];
                printf("\tIPv4 Address #%d: %s\n", i, inet_ntoa(addr));
            }
        } else if (remoteHost->h_addrtype == AF_INET6)
            printf("\tRemotehost is an IPv6 address\n");
    }

    return 0;
}
```

Windows Phone 8: This function is supported for Windows Phone Store apps on Windows Phone 8 and later.

Windows 8.1 and **Windows Server 2012 R2**: This function is supported for Windows Store apps on Windows 8.1, Windows Server 2012 R2, and later.

Requirements

Minimum supported client	Windows 8.1, Windows Vista [desktop apps | UWP apps]

Minimum supported server	Windows Server 2003 [desktop apps | UWP apps]
Target Platform	Windows
Header	winsock2.h (include Winsock2.h, Winsock.h)
Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

GetAddrInfoEx

GetAddrInfoW

WSAAsyncGetHostByAddr

Winsock Functions

Winsock Reference

addrinfo

addrinfoW

getaddrinfo

gethostbyname

hostent

gethostbyname function (winsock2.h)

10/22/2022 • 5 minutes to read • Edit Online

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

> **Note** The **gethostbyname** function has been deprecated by the introduction of the getaddrinfo function. Developers creating Windows Sockets 2 applications are urged to use the **getaddrinfo** function instead of **gethostbyname**.

Syntax

```
hostent *WSAAPI gethostbyname(
  const char *name
);
```

Parameters

`name`

TBD

Return value

If no error occurs, **gethostbyname** returns a pointer to the hostent structure described above. Otherwise, it returns a **null** pointer and a specific error number can be retrieved by calling WSAGetLastError.

ERROR CODE	MEANING
WSANOTINITIALISED	A successful WSAStartup call must occur before using this function.
WSAENETDOWN	The network subsystem has failed.
WSAHOST_NOT_FOUND	Authoritative answer host not found.
WSATRY_AGAIN	Nonauthoritative host not found, or server failure.
WSANO_RECOVERY	A nonrecoverable error occurred.

WSANO_DATA	The requested name is valid, but no data of the requested type was found. This error is also returned if the *name* parameter contains a string representation of an IPv6 address or an illegal IPv4 address. This error should not be interpreted to mean that the *name* parameter contains a name string that has been validated for a particular protocol (an IP hostname, for example). Since Winsock supports multiple name service providers, a name may potentially be valid for one provider and not accepted by another provider.
WSAEINPROGRESS	A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function.
WSAEFAULT	The *name* parameter is not a valid part of the user address space.
WSAEINTR	A blocking Windows Socket 1.1 call was canceled through WSACancelBlockingCall.

Remarks

The **gethostbyname** function returns a pointer to a hostent structure—a structure allocated by Windows Sockets. The **hostent** structure contains the results of a successful search for the host specified in the *name* parameter.

If the host specified in the *name* parameter has both IPv4 and IPv6 addresses, only the IPv4 addresses will be returned. The **gethostbyname** function can only return IPv4 addresses for the *name* parameter. The getaddrinfo function and associated addrinfo structure should be used if IPv6 addresses for the host are required or if both IPv4 and IPv6 addresses for the host are required.

If the *name* parameter points to an empty string or *name* is **NULL**, the returned string is the same as the string returned by a successful gethostname function call (the standard host name for the local computer).

If the *name* parameter contains a string representation of a legal IPv4 address, then the binary IPv4 address that represents the string is returned in the hostent structure. The **h_name** member of the **hostent** structure contains the string representation of the IPv4 address and the **h_addr_list** contains the binary IPv4 address. If the *name* parameter contains a string representation of an IPv6 address or an illegal IPv4 address, then the **gethostbyname** function will fail and return WSANO_DATA.

The memory for the hostent structure returned by the **gethostbyname** function is allocated internally by the Winsock DLL from thread local storage. Only a single **hostent** structure is allocated and used, no matter how many times the gethostbyaddr or **gethostbyname** functions are called on the thread. The returned **hostent** structure must be copied to an application buffer if additional calls are to be made to the **gethostbyname** or **gethostbyaddr** functions on the same thread. Otherwise, the return value will be overwritten by subsequent **gethostbyname** or **gethostbyaddr** calls on the same thread. The internal memory allocated for the returned **hostent** structure is released by the Winsock DLL when the thread exits.

An application should not try to release the memory used by the returned hostent structure. The application must never attempt to modify this structure or to free any of its components. Furthermore, only one copy of this structure is allocated per thread, so the application should copy any information it needs before issuing any other function calls to **gethostbyname** or gethostbyaddr .

The **gethostbyname** function cannot take an IP address string as a parameter passed to it in the *name* and

resolve it to a host name. Such a request is treated exactly as a string representation of an IPv4 address or an unknown host name were passed. An application can use the inet_addr to convert an IPv4 address string to a binary IPv4 address, then use another function, gethostbyaddr, to resolve the IPv4 address to a host name.

> **Note** The **gethostbyname** function does not check the size of the *name* parameter before passing the buffer. With an improperly sized *name* parameter, heap corruption can occur.

Example Code

The following examples demonstrates the use of the **gethostbyname** function.

```
#include <winsock2.h>
#include <ws2tcpip.h>
#include <stdio.h>
#include <windows.h>
#pragma comment(lib, "ws2_32.lib")

int main(int argc, char **argv)
{

    //-----------------------------------------
    // Declare and initialize variables
    WSADATA wsaData;
    int iResult;

    DWORD dwError;
    int i = 0;

    struct hostent *remoteHost;
    char *host_name;
    struct in_addr addr;

    char **pAlias;

    // Validate the parameters
    if (argc != 2) {
        printf("usage: %s hostname\n", argv[0]);
        printf("  to return the IP addresses for the host\n");
        printf("       %s www.contoso.com\n", argv[0]);
        printf(" or\n");
        printf("       %s IPv4string\n", argv[0]);
        printf("  to return an IPv4 binary address for an IPv4string\n");
        printf("       %s 127.0.0.1\n", argv[0]);
        return 1;
    }
    // Initialize Winsock
    iResult = WSAStartup(MAKEWORD(2, 2), &wsaData);
    if (iResult != 0) {
        printf("WSAStartup failed: %d\n", iResult);
        return 1;
    }

    host_name = argv[1];

    printf("Calling gethostbyname with %s\n", host_name);
    remoteHost = gethostbyname(host_name);

    if (remoteHost == NULL) {
        dwError = WSAGetLastError();
        if (dwError != 0) {
            if (dwError == WSAHOST_NOT_FOUND) {
                printf("Host not found\n");
                return 1;
            } else if (dwError == WSANO_DATA) {
```

```
                printf("No data record found\n");
                return 1;
            } else {
                printf("Function failed with error: %ld\n", dwError);
                return 1;
            }
        }
    } else {
        printf("Function returned:\n");
        printf("\tOfficial name: %s\n", remoteHost->h_name);
        for (pAlias = remoteHost->h_aliases; *pAlias != 0; pAlias++) {
            printf("\tAlternate name #%d: %s\n", ++i, *pAlias);
        }
        printf("\tAddress type: ");
        switch (remoteHost->h_addrtype) {
        case AF_INET:
            printf("AF_INET\n");
            break;
        case AF_NETBIOS:
            printf("AF_NETBIOS\n");
            break;
        default:
            printf(" %d\n", remoteHost->h_addrtype);
            break;
        }
        printf("\tAddress length: %d\n", remoteHost->h_length);

        i = 0;
        if (remoteHost->h_addrtype == AF_INET)
        {
            while (remoteHost->h_addr_list[i] != 0) {
                addr.s_addr = *(u_long *) remoteHost->h_addr_list[i++];
                printf("\tIP Address #%d: %s\n", i, inet_ntoa(addr));
            }
        }
        else if (remoteHost->h_addrtype == AF_NETBIOS)
        {
            printf("NETBIOS address was returned\n");
        }
    }

    return 0;
}
```

Windows Phone 8: This function is supported for Windows Phone Store apps on Windows Phone 8 and later.

Windows 8.1 and **Windows Server 2012 R2**: This function is supported for Windows Store apps on Windows 8.1, Windows Server 2012 R2, and later.

Requirements

Minimum supported client	Windows 8.1, Windows Vista [desktop apps | UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps | UWP apps]
Target Platform	Windows
Header	winsock2.h (include Winsock2.h, Winsock.h)

Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

GetAddrInfoEx

GetAddrInfoW

WSAAsyncGetHostByName

Winsock Functions

Winsock Reference

addrinfo

addrinfoW

getaddrinfo

gethostbyaddr

gethostname

hostent

inet_addr

gethostname function (winsock2.h)

10/22/2022 • 2 minutes to read • Edit Online

The **gethostname** function retrieves the standard host name for the local computer.

Syntax

```
int WSAAPI gethostname(
  [out] char *name,
  [in]  int  namelen
);
```

Parameters

`[out] name`

A pointer to a buffer that receives the local host name.

`[in] namelen`

The length, in bytes, of the buffer pointed to by the *name* parameter.

Return value

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

ERROR CODE	MEANING
WSAEFAULT	The *name* parameter is a **NULL** pointer or is not a valid part of the user address space. This error is also returned if the buffer size specified by *namelen* parameter is too small to hold the complete host name.
WSANOTINITIALISED	A successful WSAStartup call must occur before using this function.
WSAENETDOWN	The network subsystem has failed.
WSAEINPROGRESS	A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function.

Remarks

The **gethostname** function returns the name of the local host into the buffer specified by the *name* parameter. The host name is returned as a **null**-terminated string. The form of the host name is dependent on the Windows Sockets provider—it can be a simple host name, or it can be a fully qualified domain name. However, it is guaranteed that the name returned will be successfully parsed by gethostbyname and WSAAsyncGetHostByName.

The maximum length of the name returned in the buffer pointed to by the *name* parameter is dependent on the namespace provider.

If the **gethostname** function is used on a cluster resource on Windows Server 2008, Windows Server 2003, or Windows 2000 Server and the *CLUSTER_NETWORK_NAME* environment variable is defined, then the value in this environment variable overrides the actual hostname and is returned. On a cluster resource, the *CLUSTER_NETWORK_NAME* environment variable contains the name of the cluster.

The **gethostname** function queries namespace providers to determine the local host name using the SVCID_HOSTNAME GUID defined in the *Svgguid.h* header file. If no namespace provider responds, then the **gethostname** function returns the NetBIOS name of the local computer.

The maximum length, in bytes, of the string returned in the buffer pointed to by the *name* parameter is dependent on the namespace provider, but this string must be 256 bytes or less. So if a buffer of 256 bytes is passed in the *name* parameter and the *namelen* parameter is set to 256, the buffer size will always be adequate.

> **Note** If no local host name has been configured, **gethostname** must succeed and return a token host name that gethostbyname or WSAAsyncGetHostByName can resolve.

Windows Phone 8: This function is supported for Windows Phone Store apps on Windows Phone 8 and later.

Windows 8.1 and **Windows Server 2012 R2**: This function is supported for Windows Store apps on Windows 8.1, Windows Server 2012 R2, and later.

Requirements

Minimum supported client	Windows 8.1, Windows Vista [desktop apps | UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps | UWP apps]
Target Platform	Windows
Header	winsock2.h (include Winsock2.h)
Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

GetAddrInfoW

WSAAsyncGetHostByName

Winsock Functions

Winsock Reference

gethostbyname

GetHostNameW function (winsock2.h)

10/22/2022 • 2 minutes to read • Edit Online

The **GetHostNameW** function retrieves the standard host name for the local computer as a Unicode string.

Syntax

```
int WSAAPI GetHostNameW(
  [out] PWSTR name,
  [in]  int   namelen
);
```

Parameters

`[out] name`

A pointer to a buffer that receives the local host name as a **null**-terminated Unicode string.

`[in] namelen`

The length, in wide characters, of the buffer pointed to by the *name* parameter.

Return value

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

ERROR CODE	MEANING
WSAEFAULT	The *name* parameter is a **NULL** pointer or is not a valid part of the user address space. This error is also returned if the buffer size specified by *namelen* parameter is too small to hold the complete host name.
WSANOTINITIALISED	A successful WSAStartup call must occur before using this function.
WSAENETDOWN	The network subsystem has failed.

Remarks

The **GetHostNameW** function returns the name of the local host into the buffer specified by the *name* parameter in Unicode (UTF-16). The host name is returned as a **null**-terminated Unicode string. The form of the host name is dependent on the Windows Sockets provider—it can be a simple host name, or it can be a fully qualified domain name. However, it is guaranteed that the name returned will be successfully parsed by GetAddrInfoW.

With the growth of the Internet, there is a growing need to identify Internet host names for other languages not represented by the ASCII character set. Identifiers which facilitate this need and allow non-ASCII characters (Unicode) to be represented as special ASCII character strings (Punycode) are known as Internationalized

Domain Names (IDNs). A mechanism called Internationalizing Domain Names in Applications (IDNA) is used to handle IDNs in a standard fashion. The **GetHostNameW** function does not convert the local hostname between Punycode and Unicode. The GetAddrInfoW function provides support for Internationalized Domain Name (IDN) parsing and performs Punycode/IDN encoding and conversion.

If the **GetHostNameW** function is used on a cluster resource on Windows Server 2012 and the *CLUSTER_NETWORK_NAME* environment variable is defined, then the value in this environment variable overrides the actual hostname and is returned. On a cluster resource, the *CLUSTER_NETWORK_NAME* environment variable contains the name of the cluster.

The **GetHostNameW** function queries namespace providers to determine the local host name using the SVCID_HOSTNAME GUID defined in the *Svgguid.h* header file. If no namespace provider responds, then the **GetHostNameW** function returns the NetBIOS name of the local computer in Unicode.

The maximum length, in wide characters, of the string returned in the buffer pointed to by the *name* parameter is dependent on the namespace provider, but this string must be 256 wide characters or less. So if a buffer of 256 wide characters is passed in the *name* parameter and the *namelen* parameter is set to 256, the buffer size will always be adequate.

> **Note** If no local host name has been configured, **GetHostNameW** must succeed and return a token host name that GetAddrInfoW can resolve.

Windows Phone 8: This function is supported for Windows Phone Store apps on Windows Phone 8 and later.

Windows 8.1 and **Windows Server 2012 R2**: This function is supported for Windows Store apps on Windows 8.1, Windows Server 2012 R2, and later.

Requirements

Minimum supported client	Windows 8.1, Windows 8 [desktop apps | UWP apps]
Minimum supported server	Windows Server 2012 [desktop apps | UWP apps]
Target Platform	Windows
Header	winsock2.h
Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

GetAddrInfoW

Winsock Functions

Winsock Reference

gethostname

getpeername function (winsock2.h)

10/22/2022 • 2 minutes to read • Edit Online

The **getpeername** function retrieves the address of the peer to which a socket is connected.

Syntax

```
int WSAAPI getpeername(
  [in]      SOCKET   s,
  [out]     sockaddr *name,
  [in, out] int      *namelen
);
```

Parameters

`[in] s`

A descriptor identifying a connected socket.

`[out] name`

The SOCKADDR structure that receives the address of the peer.

`[in, out] namelen`

A pointer to the size, in bytes, of the *name* parameter.

Return value

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

ERROR CODE	MEANING
WSANOTINITIALISED	A successful WSAStartup call must occur before using this function.
WSAENETDOWN	The network subsystem has failed.
WSAEFAULT	The *name* or the *namelen* parameter is not in a valid part of the user address space, or the *namelen* parameter is too small.
WSAEINPROGRESS	A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function.
WSAENOTCONN	The socket is not connected.

WSAENOTSOCK	The descriptor is not a socket.

Remarks

The **getpeername** function retrieves the address of the peer connected to the socket *s* and stores the address in the SOCKADDR structure identified by the *name* parameter. This function works with any address family and it simply returns the address to which the socket is connected. The **getpeername** function can be used only on a connected socket.

For datagram sockets, only the address of a peer specified in a previous connect call will be returned. Any address specified by a previous sendto call will not be returned by **getpeername**.

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

Windows Phone 8: This function is supported for Windows Phone Store apps on Windows Phone 8 and later.

Windows 8.1 and **Windows Server 2012 R2**: This function is supported for Windows Store apps on Windows 8.1, Windows Server 2012 R2, and later.

Requirements

Minimum supported client	Windows 8.1, Windows Vista [desktop apps | UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps | UWP apps]
Target Platform	Windows
Header	winsock2.h (include Winsock2.h)
Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

Winsock Functions

Winsock Reference

bind

connect

getsockname

sendto

socket

getprotobyname function (winsock2.h)

10/22/2022 • 2 minutes to read • Edit Online

The **getprotobyname** function retrieves the protocol information corresponding to a protocol name.

Syntax

```
protoent *WSAAPI getprotobyname(
  [in] const char *name
);
```

Parameters

`[in] name`

Pointer to a null-terminated protocol name.

Return value

If no error occurs, **getprotobyname** returns a pointer to the protoent. Otherwise, it returns a null pointer and a specific error number can be retrieved by calling WSAGetLastError.

ERROR CODE	MEANING
WSANOTINITIALISED	A successful WSAStartup call must occur before using this function.
WSAENETDOWN	The network subsystem has failed.
WSAHOST_NOT_FOUND	Authoritative answer protocol not found.
WSATRY_AGAIN	A nonauthoritative protocol not found, or server failure.
WSANO_RECOVERY	Nonrecoverable errors, the protocols database is not accessible.
WSANO_DATA	Valid name, no data record of requested type.
WSAEINPROGRESS	A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function.
WSAEFAULT	The *name* parameter is not a valid part of the user address space.

WSAEINTR	A blocking Windows Socket 1.1 call was canceled through WSACancelBlockingCall.

Remarks

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

Windows Phone 8: This function is supported for Windows Phone Store apps on Windows Phone 8 and later.

Windows 8.1 and **Windows Server 2012 R2**: This function is supported for Windows Store apps on Windows 8.1, Windows Server 2012 R2, and later.

Requirements

Minimum supported client	Windows 8.1, Windows Vista [desktop apps | UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps | UWP apps]
Target Platform	Windows
Header	winsock2.h (include Winsock2.h)
Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

WSAAsyncGetProtoByName

Winsock Functions

Winsock Reference

getprotobynumber

getprotobynumber function (winsock2.h)

10/22/2022 • 2 minutes to read • Edit Online

The **getprotobynumber** function retrieves protocol information corresponding to a protocol number.

Syntax

```
protoent *WSAAPI getprotobynumber(
  [in] int number
);
```

Parameters

`[in] number`

Protocol number, in host byte order.

Return value

If no error occurs, **getprotobynumber** returns a pointer to the protoent structure. Otherwise, it returns a null pointer and a specific error number can be retrieved by calling WSAGetLastError.

ERROR CODE	MEANING
WSANOTINITIALISED	A successful WSAStartup call must occur before using this function.
WSAENETDOWN	The network subsystem has failed.
WSAHOST_NOT_FOUND	Authoritative answer protocol not found.
WSATRY_AGAIN	A nonauthoritative Protocol not found, or server failure.
WSANO_RECOVERY	Nonrecoverable errors, the protocols database is not accessible.
WSANO_DATA	Valid name, no data record of requested type.
WSAEINPROGRESS	A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function.
WSAEINTR	A blocking Windows Socket 1.1 call was canceled through WSACancelBlockingCall.

Remarks

This **getprotobynumber** function returns a pointer to the protoent structure as previously described in getprotobyname. The contents of the structure correspond to the given protocol number.

The pointer that is returned points to the structure allocated by Windows Sockets. The application must never attempt to modify this structure or to free any of its components. Furthermore, only one copy of this structure is allocated per thread, so the application should copy any information that it needs before issuing any other Windows Sockets function calls.

Windows Phone 8: This function is supported for Windows Phone Store apps on Windows Phone 8 and later.

Windows 8.1 and **Windows Server 2012 R2**: This function is supported for Windows Store apps on Windows 8.1, Windows Server 2012 R2, and later.

Requirements

Minimum supported client	Windows 8.1, Windows Vista [desktop apps | UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps | UWP apps]
Target Platform	Windows
Header	winsock2.h (include Winsock2.h)
Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

WSAAsyncGetProtoByNumber

Winsock Functions

Winsock Reference

getprotobyname

getservbyname function (winsock2.h)

10/22/2022 • 2 minutes to read • Edit Online

The **getservbyname** function retrieves service information corresponding to a service name and protocol.

Syntax

```
servent *WSAAPI getservbyname(
  [in] const char *name,
  [in] const char *proto
);
```

Parameters

`[in] name`

A pointer to a **null**-terminated service name.

`[in] proto`

A pointer to a **null**-terminated protocol name. If this pointer is **NULL**, the **getservbyname** function returns the first service entry where *name* matches the **s_name** member of the servent structure or the **s_aliases** member of the **servent** structure. Otherwise, **getservbyname** matches both the *name* and the *proto*.

Return value

If no error occurs, **getservbyname** returns a pointer to the servent structure. Otherwise, it returns a **null** pointer and a specific error number can be retrieved by calling WSAGetLastError.

ERROR CODE	MEANING
WSANOTINITIALISED	A successful WSAStartup call must occur before using this function.
WSAENETDOWN	The network subsystem has failed.
WSAHOST_NOT_FOUND	Authoritative Answer Service not found.
WSATRY_AGAIN	A nonauthoritative Service not found, or server failure.
WSANO_RECOVERY	Nonrecoverable errors, the services database is not accessible.
WSANO_DATA	Valid name, no data record of requested type.

WSAEINPROGRESS	A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function.
WSAEINTR	A blocking Windows Socket 1.1 call was canceled through WSACancelBlockingCall.

Remarks

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

The pointer that is returned points to the **servent** structure allocated by the Windows Sockets library. The application must never attempt to modify this structure or to free any of its components. Furthermore, only one copy of this structure is allocated per thread, so the application should copy any information it needs before issuing any other Windows Sockets function calls.

Windows Phone 8: This function is supported for Windows Phone Store apps on Windows Phone 8 and later.

Windows 8.1 and **Windows Server 2012 R2**: This function is supported for Windows Store apps on Windows 8.1, Windows Server 2012 R2, and later.

Requirements

Minimum supported client	Windows 8.1, Windows Vista [desktop apps | UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps | UWP apps]
Target Platform	Windows
Header	winsock2.h (include Winsock2.h)
Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

WSAAsyncGetServByName

Winsock Functions

Winsock Reference

getservbyport

getservbyport function (winsock2.h)

10/22/2022 • 2 minutes to read • Edit Online

The **getservbyport** function retrieves service information corresponding to a port and protocol.

Syntax

```
servent *WSAAPI getservbyport(
  [in] int        port,
  [in] const char *proto
);
```

Parameters

`[in] port`

Port for a service, in network byte order.

`[in] proto`

Optional pointer to a protocol name. If this is null, **getservbyport** returns the first service entry for which the *port* matches the **s_port** of the servent structure. Otherwise, **getservbyport** matches both the *port* and the *proto* parameters.

Return value

If no error occurs, **getservbyport** returns a pointer to the servent structure. Otherwise, it returns a null pointer and a specific error number can be retrieved by calling WSAGetLastError.

ERROR CODE	MEANING
WSANOTINITIALISED	A successful WSAStartup call must occur before using this function.
WSAENETDOWN	The network subsystem has failed.
WSAHOST_NOT_FOUND	Authoritative Answer Service not found.
WSATRY_AGAIN	A nonauthoritative Service not found, or server failure.
WSANO_RECOVERY	Nonrecoverable errors, the services database is not accessible.
WSANO_DATA	Valid name, no data record of requested type.

WSAEINPROGRESS	A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function.
WSAEFAULT	The *proto* parameter is not a valid part of the user address space.
WSAEINTR	A blocking Windows Socket 1.1 call was canceled through WSACancelBlockingCall.

Remarks

The **getservbyport** function returns a pointer to a servent structure as it does in the getservbyname function.

The **servent** structure is allocated by Windows Sockets. The application must never attempt to modify this structure or to free any of its components. Furthermore, only one copy of this structure is allocated per thread, so the application should copy any information it needs before issuing any other Windows Sockets function calls.

Windows Phone 8: This function is supported for Windows Phone Store apps on Windows Phone 8 and later.

Windows 8.1 and **Windows Server 2012 R2**: This function is supported for Windows Store apps on Windows 8.1, Windows Server 2012 R2, and later.

Requirements

Minimum supported client	Windows 8.1, Windows Vista [desktop apps | UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps | UWP apps]
Target Platform	Windows
Header	winsock2.h (include Winsock2.h)
Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

WSAAsyncGetServByPort

Winsock Functions

Winsock Reference

getservbyname

getsockname function (winsock2.h)

10/22/2022 • 2 minutes to read • Edit Online

The **getsockname** function retrieves the local name for a socket.

Syntax

```
int WSAAPI getsockname(
  [in]      SOCKET   s,
  [out]     sockaddr *name,
  [in, out] int      *namelen
);
```

Parameters

`[in] s`

Descriptor identifying a socket.

`[out] name`

Pointer to a SOCKADDR structure that receives the address (name) of the socket.

`[in, out] namelen`

Size of the *name* buffer, in bytes.

Return value

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

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

WSAEINVAL	The socket has not been bound to an address with bind, or ADDR_ANY is specified in **bind** but connection has not yet occurred.

Remarks

The **getsockname** function retrieves the current name for the specified socket descriptor in *name*. It is used on the bound or connected socket specified by the *s* parameter. The local association is returned. This call is especially useful when a connect call has been made without doing a bind first; the **getsockname** function provides the only way to determine the local association that has been set by the system.

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

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

Windows Phone 8: This function is supported for Windows Phone Store apps on Windows Phone 8 and later.

Windows 8.1 and **Windows Server 2012 R2**: This function is supported for Windows Store apps on Windows 8.1, Windows Server 2012 R2, and later.

Requirements

Minimum supported client	Windows 8.1, Windows Vista [desktop apps | UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps | UWP apps]
Target Platform	Windows
Header	winsock2.h (include Winsock2.h)
Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

SOCKADDR

Winsock Functions

Winsock Reference

bind

getpeername

socket

getsockopt function (winsock2.h)

10/22/2022 • 17 minutes to read • Edit Online

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

Syntax

```
int WSAAPI getsockopt(
  [in]      SOCKET s,
  [in]      int    level,
  [in]      int    optname,
  [out]     char   *optval,
  [in, out] int    *optlen
);
```

Parameters

`[in] s`

A descriptor identifying a socket.

`[in] level`

The level at which the option is defined. Example: SOL_SOCKET.

`[in] optname`

The socket option for which the value is to be retrieved. Example: SO_ACCEPTCONN. The *optname* value must be a socket option defined within the specified *level*, or behavior is undefined.

`[out] optval`

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

`[in, out] optlen`

A pointer to the size, in bytes, of the *optval* buffer.

Return value

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

ERROR CODE	MEANING
WSANOTINITIALISED	A successful WSAStartup call must occur before using this function.
WSAENETDOWN	**Note** The network subsystem has failed.

WSAEFAULT	One of the *optval* or the *optlen* parameters is not a valid part of the user address space, or the *optlen* parameter is too small.
WSAEINPROGRESS	A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function.
WSAEINVAL	The *level* parameter is unknown or invalid.
WSAENOPROTOOPT	The option is unknown or unsupported by the indicated protocol family.
WSAENOTSOCK	The descriptor is not a socket.

Remarks

The **getsockopt** function retrieves the current value for a socket option associated with a socket of any type, in any state, and stores the result in *optval*. Options can exist at multiple protocol levels, but they are always present at the uppermost socket level. Options affect socket operations, such as the packet routing and OOB data transfer.

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

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

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

The following options are supported for **getsockopt**. The Type column identifies the type of data addressed by *optval*.

For more information on socket options, see Socket Options.

The following table of value for the *optname* parameter are valid when the *level* parameter is set to **SOL_SOCKET**.

VALUE	TYPE	MEANING
SO_ACCEPTCONN	BOOL	The socket is listening.
SO_BROADCAST	BOOL	The socket is configured for the transmission and receipt of broadcast messages.
SO_BSP_STATE	CSADDR_INFO	Returns the local address, local port, remote address, remote port, socket type, and protocol used by a socket.
SO_CONDITIONAL_ACCEPT	BOOL	Returns current socket state, either from a previous call to setsockopt or the system default.

SO_CONNECT_TIME	DWORD	Returns the number of seconds a socket has been connected. This socket option is valid for connection oriented protocols only.
SO_DEBUG	BOOL	Debugging is enabled.
SO_DONTLINGER	BOOL	If **TRUE**, the SO_LINGER option is disabled.
SO_DONTROUTE	BOOL	Routing is disabled. Setting this succeeds but is ignored on AF_INET sockets; fails on AF_INET6 sockets with WSAENOPROTOOPT. This option is not supported on ATM sockets.
SO_ERROR	int	Retrieves error status and clear.
SO_EXCLUSIVEADDRUSE	BOOL	Prevents any other socket from binding to the same address and port. This option must be set before calling the bind function.
SO_GROUP_ID	GROUP	Reserved.
SO_GROUP_PRIORITY	int	Reserved.
SO_KEEPALIVE	BOOL	Keep-alives are being sent. Not supported on ATM sockets.
SO_LINGER	LINGER structure	Returns the current linger options.
SO_MAX_MSG_SIZE	unsigned int	The maximum size of a message for message-oriented socket types (for example, SOCK_DGRAM). Has no meaning for stream oriented sockets.
SO_OOBINLINE	BOOL	OOB data is being received in the normal data stream. (See section Windows Sockets 1.1 Blocking Routines and EINPROGRESS for a discussion of this topic.)
SO_PORT_SCALABILITY	BOOL	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.
SO_PROTOCOL_INFO	WSAPROTOCOL_INFO	A description of the protocol information for the protocol that is bound to this socket.

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.
SO_REUSEADDR	BOOL	The socket can be bound to an address which is already in use. Not applicable for ATM sockets.
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.
SO_TYPE	int	The type of the socket (for example, SOCK_STREAM).
PVD_CONFIG	Service Provider Dependent	An opaque data structure object from the service provider associated with socket *s*. This object stores the current configuration information of the service provider. The exact format of this data structure is service provider specific.

level = **IPPROTO_TCP**

See **TCP_NODELAY** in IPPROTO_TCP socket options. Also see that topic for more complete and detailed information about socket options for *level* = **IPPROTO_TCP**.

The following table of value for the *optname* parameter are valid when the *level* parameter is set to **NSPROTO_IPX**.

> **Note** Windows NT supports all IPX options. Windows Me, Windows 98, and Windows 95 support only the following options:
> IPX_PTYPE
> IPX_FILTERPTYPE
> IPX_DSTYPE
> IPX_RECVHDR
> IPX_MAXSIZE
> IPX_ADDRESS

VALUE	TYPE	MEANING
IPX_PTYPE	int	Retrieves the IPX packet type.
IPX_FILTERPTYPE	int	Retrieves the receive filter packet type
IPX_DSTYPE	int	Obtains the value of the data stream field in the SPX header on every packet sent.

IPX_EXTENDED_ADDRESS	BOOL	Finds out whether extended addressing is enabled.
IPX_RECVHDR	BOOL	Finds out whether the protocol header is sent up on all receive headers.
IPX_MAXSIZE	int	Obtains the maximum data size that can be sent.
IPX_ADDRESS	IPX_ADDRESS_DATA structure	Obtains information about a specific adapter to which IPX is bound. Adapter numbering is base zero. The **adapternum** member is filled in upon return.
IPX_GETNETINFO	IPX_NETNUM_DATA structure	Obtains information about a specific IPX network number. If not available in the cache, uses RIP to obtain information.
IPX_GETNETINFO_NORIP	IPX_NETNUM_DATA structure	Obtains information about a specific IPX network number. If not available in the cache, will not use RIP to obtain information, and returns error.
IPX_SPXGETCONNECTIONSTATUS	IPX_SPXCONNSTATUS_DATA structure	Retrieves information about a connected SPX socket.
IPX_ADDRESS_NOTIFY	IPX_ADDRESS_DATA structure	Retrieves status notification when changes occur on an adapter to which IPX is bound.
IPX_MAX_ADAPTER_NUM	int	Retrieves maximum number of adapters present, numbered as base zero.
IPX_RERIPNETNUMBER	IPX_NETNUM_DATA structure	Similar to IPX_GETNETINFO, but forces IPX to use RIP for resolution, even if the network information is in the local cache.
IPX_IMMEDIATESPXACK	BOOL	Directs SPX connections not to delay before sending an ACK. Applications without back-and-forth traffic should set this to **TRUE** to increase performance.
TCP_MAXSEG	int	Receives TCP maximum-segment size. Supported in Windows 10 and newer versions.

The following table lists value for the *optname* that represent BSD socket options that are not supported by the **getsockopt** function.

VALUE	TYPE	MEANING

SO_RCVLOWAT	int	Receives low watermark.
SO_RCVTIMEO	int	Receives time-out.
SO_SNDLOWAT	int	Sends low watermark.
SO_SNDTIMEO	int	Sends time-out.
TCP_MAXSEG	int	Receives TCP maximum-segment size. Not supported in versions before Windows 10.

> **Note** When using the recv function, if no data arrives during the period specified in SO_RCVTIMEO, the **recv** function completes. In Windows versions prior to Windows 2000, any data received subsequently fails with WSAETIMEDOUT. In Windows 2000 and later, if no data arrives within the period specified in SO_RCVTIMEO, the **recv** function returns WSAETIMEDOUT, and if data is received, **recv** returns SUCCESS.

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

More detailed information on some of the socket options for the *optname* parameter supported by the **getsockopt** function are listed below.

SO_CONNECT_TIME
This option returns the number of seconds a socket has been connected. This option is valid for connection oriented protocols only.
The SO_CONNECT_TIME option can be used with the **getsockopt** function to check whether a connection has been established. This option can also be used while a ConnectEx function call is in progress. If a connection is established, the SO_CONNECT_TIME option can determine how long the connection has been established. If the socket is not connected, the **getsockopt** returns SOCKET_ERROR. Checking a connection like this is necessary to see if connections that have been established for a while, without sending any data. It is recommended that applications terminate these connections.

SO_DEBUG

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

SO_ERROR
The SO_ERROR option returns and resets the per socket–based error code, which is different from the per thread based–error code that is handled using the WSAGetLastError and WSASetLastError function calls. A successful call using the socket does not reset the socket based error code returned by the SO_ERROR option.
SO_EXCLUSIVEADDRUSE
Prevents any other socket from binding to the same address and port. This option must be set before calling the bind function. See the SO_EXCLUSIVEADDRUSE reference for more information.
SO_GROUP_ID

> **Note** This option is reserved. This option is also exclusive to **getsockopt**; the value should be **NULL**.

SO_GROUP_PRIORITY

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

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

SO_KEEPALIVE

An application can request that a TCP/IP service provider enable the use of keep-alive packets on TCP connections by turning on the SO_KEEPALIVE socket option. This option queries the current value of the keep-alive option on a socket. A Windows Sockets provider need not support the use of keep-alive: if it does, the precise semantics are implementation-specific but should conform to section 4.2.3.6 on the *Requirements for Internet Hosts—Communication Layers* specified in RFC 1122 available at the IETF website. If a connection is dropped as the result of keep-alives the error code WSAENETRESET is returned to any calls in progress on the socket, and any subsequent calls will fail with WSAENOTCONN. SO_KEEPALIVE is not supported on ATM sockets, and requests to enable the use of keep-alive packets on an ATM socket results in an error being returned by the socket.

SO_LINGER

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

SO_MAX_MSG_SIZE

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

SO_PROTOCOL_INFO

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

SO_SNDBUF

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

SO_REUSEADDR

By default, a socket cannot be bound (see bind) to a local address that is already in use. On occasion, however, it can be necessary to reuse an address in this way. Because every connection is uniquely identified by the combination of local and remote addresses, there is no problem with having two sockets bound to the same local address as long as the remote addresses are different. To inform the Windows Sockets provider that a **bind** on a socket should not be disallowed because the desired address is already in use by another socket, the application should set the SO_REUSEADDR socket option for the socket before issuing the **bind**. Note that the option is interpreted only at the time of the **bind**: it is therefore unnecessary (but harmless) to set the option on a socket that is not to be bound to an existing address, and setting or resetting the option after the **bind** has no effect on this or any other socket. SO_REUSEADDR is not applicable for ATM sockets, and although requests to reuse and address do not result in an error, they have no effect on when an ATM socket is in use.

PVD_CONFIG

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

TCP_NODELAY

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

> **Note** When issuing a blocking Winsock call such as **getsockopt**, Winsock may need to wait for a network event before the call can complete. Winsock performs an alertable wait in this situation, which can be interrupted by an asynchronous procedure call (APC) scheduled on the same thread. Issuing another blocking Winsock call inside an APC that interrupted an ongoing blocking Winsock call on the same thread will lead to undefined behavior, and must never be attempted by Winsock clients.

Example Code

The following code sample demonstrates the use of the **getsockopt** function.

```
#include <stdio.h>
#include "winsock2.h"
#include <windows.h>

void main() {

  //----------------------------------------
  // Declare variables
  WSADATA wsaData;
  SOCKET ListenSocket;
  sockaddr_in service;

  //----------------------------------------
  // Initialize Winsock
  int iResult = WSAStartup( MAKEWORD(2,2), &wsaData );
  if( iResult != NO_ERROR )
    printf("Error at WSAStartup\n");

  //----------------------------------------
  // Create a listening socket
  ListenSocket = socket( AF_INET, SOCK_STREAM, IPPROTO_TCP );
  if (ListenSocket == INVALID_SOCKET) {
    printf("Error at socket()\n");
    WSACleanup();
    return;
  }

  //----------------------------------------
  // Bind the socket to the local IP address
  // and port 27015
  hostent* thisHost;
  char* ip;
  u_short port;
  port = 27015;
  thisHost = gethostbyname("");
  ip = inet_ntoa (*(struct in_addr *)*thisHost->h_addr_list);

  service.sin_family = AF_INET;
  service.sin_addr.s_addr = inet_addr(ip);
  service.sin_port = htons(port);
```

```
  if ( bind( ListenSocket,(SOCKADDR*) &service, sizeof(service) )  == SOCKET_ERROR ) {
    printf("bind failed\n");
    closesocket(ListenSocket);
    return;
  }

  //----------------------------------------
  // Initialize variables and call getsockopt.
  // The SO_ACCEPTCONN parameter is a socket option
  // that tells the function to check whether the
  // socket has been put in listening mode or not.
  // The various socket options return different
  // information about the socket. This call should
  // return 0 to the optVal parameter, since the socket
  // is not in listening mode.
  int optVal;
  int optLen = sizeof(int);

  if (getsockopt(ListenSocket,
    SOL_SOCKET,
    SO_ACCEPTCONN,
    (char*)&optVal,
    &optLen) != SOCKET_ERROR)
    printf("SockOpt Value: %ld\n", optVal);

  //----------------------------------------
  // Put the listening socket in listening mode.
  if (listen( ListenSocket, 100 ) == SOCKET_ERROR) {
    printf("error listening\n");
  }

  //----------------------------------------
  // Call getsockopt again to verify that
  // the socket is in listening mode.
  if (getsockopt(ListenSocket,
    SOL_SOCKET,
    SO_ACCEPTCONN,
    (char*)&optVal,
    &optLen) != SOCKET_ERROR)
    printf("SockOpt Value: %ld\n", optVal);

  WSACleanup();
  return;
}
```

Notes for IrDA Sockets

- The Af_irda.h header file must be explicitly included.
- Windows returns WSAENETDOWN to indicate the underlying transceiver driver failed to initialize with the IrDA protocol stack.
- IrDA supports several special socket options:

VALUE	TYPE	MEANING
IRLMP_ENUMDEVICES	*DEVICELIST	Describes devices in range.
IRLMP_IAS_QUERY	*IAS_QUERY	Retrieve IAS attributes.

Before an IrDA socket connection can be initiated, a device address must be obtained by performing a **getsockopt**(,,IRLMP_ENUMDEVICES,,) function call, which returns a list of all available IrDA devices. A device

address returned from the function call is copied into a SOCKADDR_IRDA structure, which in turn is used by a subsequent call to the connect function call.

Discovery can be performed in two ways:

1. First, performing a getsockopt function call with the IRLMP_ENUMDEVICES option causes a single discovery to be run on each idle adapter. The list of discovered devices and cached devices (on active adapters) is returned immediately.

   The following code demonstrates this approach.

```
#include <winsock2.h>
#include <ws2tcpip.h>
#include <af_irda.h>
#include <stdio.h>
#include <windows.h>

// link with Ws2_32.lib

int __cdecl main()
{

    //-----------------------------------------
    // Declare and initialize variables
    WSADATA wsaData;

    int iResult;
    int i;
    DWORD dwError;

    SOCKET Sock = INVALID_SOCKET;

#define DEVICE_LIST_LEN    10


    SOCKADDR_IRDA DestSockAddr = { AF_IRDA, 0, 0, 0, 0, "SampleIrDAService" };

    unsigned char DevListBuff[sizeof (DEVICELIST) -
                              sizeof (IRDA_DEVICE_INFO) +
                              (sizeof (IRDA_DEVICE_INFO) * DEVICE_LIST_LEN)];

    int DevListLen = sizeof (DevListBuff);
    PDEVICELIST pDevList;

    pDevList = (PDEVICELIST) & DevListBuff;

    // Initialize Winsock
    iResult = WSAStartup(MAKEWORD(2, 2), &wsaData);
    if (iResult != 0) {
        printf("WSAStartup failed: %d\n", iResult);
        return 1;
    }

    Sock = socket(AF_IRDA, SOCK_STREAM, 0);
    if (Sock == INVALID_SOCKET) {
        dwError = WSAGetLastError();
        printf
            ("socket failed trying to create an AF_IRDA socket with error %d\n",
             dwError);

        if (dwError == WSAEAFNOSUPPORT) {
            printf("Check that the local computer has an infrared device\n");
            printf
                ("and a device driver is installed for the infrared device\n");
        }
        WSACleanup();
        return 1;
```

```
    }
    // Sock is not in connected state
    iResult = getsockopt(Sock, SOL_IRLMP, IRLMP_ENUMDEVICES,
                         (char *) pDevList, &DevListLen);
    if (iResult == SOCKET_ERROR) {
        printf("getsockopt failed with error %d\n", WSAGetLastError());
        WSACleanup();
        return 1;
    }

    if (pDevList->numDevice == 0) {
        // no devices discovered or cached
        // not a bad idea to run a couple of times
        printf("No IRDA devices were discovered or cached\n");
    } else {
        // one per discovered device
        for (i = 0; i < (int) pDevList->numDevice; i++) {
            // typedef struct _IRDA_DEVICE_INFO
            // {
            //     u_char    irdaDeviceID[4];
            //     char      irdaDeviceName[22];
            //     u_char    irdaDeviceHints1;
            //     u_char    irdaDeviceHints2;
            //     u_char    irdaCharSet;
            // } _IRDA_DEVICE_INFO;

            // pDevList->Device[i]. see _IRDA_DEVICE_INFO for fields
            // display the device names and let the user select one
        }
    }

    // assume the user selected the first device [0]
    memcpy(&DestSockAddr.irdaDeviceID[0], &pDevList->Device[0].irdaDeviceID[0],
           4);

    iResult = connect(Sock, (const struct sockaddr *) &DestSockAddr,
                      sizeof (SOCKADDR_IRDA));
    if (iResult == SOCKET_ERROR) {
        printf("connect failed with error %d\n", WSAGetLastError());
    } else
        printf("connect to first IRDA device was successful\n");

    WSACleanup();
    return 0;
}
```

2. The second approach to performing discovery of IrDA device addresses is to perform a lazy discovery; in this approach, the application is not notified until the discovered devices list changes from the last discovery run by the stack.

The **DEVICELIST** structure shown in the Type column in the previous table is an extendible array of device descriptions. IrDA fills in as many device descriptions as can fit in the specified buffer. The device description consists of a device identifier necessary to form a sockaddr_irda structure, and a displayable string describing the device.

The **IAS_QUERY** structure shown in the Type column in the previous table is used to retrieve a single attribute of a single class from a peer device's IAS database. The application specifies the device and class to query and the attribute and attribute type. Note that the device would have been obtained previously by a call to **getsockopt**(IRLMP_ENUMDEVICES). It is expected that the application allocates a buffer, of the necessary size, for the returned parameters.

Many level socket options are not meaningful to IrDA; only SO_LINGER and SO_DONTLINGER are specifically supported.

Windows Phone 8: This function is supported for Windows Phone Store apps on Windows Phone 8 and later.

Windows 8.1 and **Windows Server 2012 R2**: This function is supported for Windows Store apps on Windows 8.1, Windows Server 2012 R2, and later.

Requirements

Minimum supported client	Windows 8.1, Windows Vista [desktop apps | UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps | UWP apps]
Target Platform	Windows
Header	winsock2.h (include Winsock2.h)
Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

IPPROTO_IP Socket Options

IPPROTO_IPV6 Socket Options

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

Socket Options

WSAAsyncSelect

WSAConnect

WSAGetLastError

WSAIoctl

WSASetLastError

Winsock Functions

ioctlsocket

recv

setsockopt

socket

HOSTENT structure (winsock2.h)

10/22/2022 • 3 minutes to read • Edit Online

The **hostent** structure is used by functions to store information about a given host, such as host name, IPv4 address, and so forth. An application should never attempt to modify this structure or to free any of its components. Furthermore, only one copy of the **hostent** structure is allocated per thread, and an application should therefore copy any information that it needs before issuing any other Windows Sockets API calls.

Syntax

```
typedef struct hostent {
  char  *h_name;
  char  **h_aliases;
  short h_addrtype;
  short h_length;
  char  **h_addr_list;
} HOSTENT, *PHOSTENT, *LPHOSTENT;
```

Members

`h_name`

The official name of the host (PC). If using the DNS or similar resolution system, it is the Fully Qualified Domain Name (FQDN) that caused the server to return a reply. If using a local hosts file, it is the first entry after the IPv4 address.

`h_aliases`

A **NULL**-terminated array of alternate names.

`h_addrtype`

The type of address being returned.

`h_length`

The length, in bytes, of each address.

`h_addr_list`

A **NULL**-terminated list of addresses for the host. Addresses are returned in network byte order. The macro **h_addr** is defined to be `h_addr_list[0]` for compatibility with older software.

Remarks

The gethostbyaddr and gethostbyname functions returns a pointer to a **hostent** structure—a structure allocated by Windows Sockets. The **hostent** structure contains the results of a successful search for the host specified in the *name* parameter.

The memory for the **hostent** structure returned by the gethostbyaddr and gethostbyname functions is allocated internally by the Winsock DLL from thread local storage. Only a single **hostent** structure is allocated and used, no matter how many times the **gethostbyaddr** or **gethostbyname** functions are called on the thread. The returned **hostent** structure must be copied to an application buffer if additional calls are to be made to the

gethostbyaddr or **gethostbyname** functions on the same thread. Otherwise, the return value will be overwritten by subsequent **gethostbyaddr** or **gethostbyname** calls on the same thread. The internal memory allocated for the returned **hostent** structure is released by the Winsock DLL when the thread exits.

An application should not try to release the memory used by the returned **hostent** structure. The application must never attempt to modify this structure or to free any of its components. Furthermore, only one copy of this structure is allocated per thread, so the application should copy any information it needs before issuing any other function calls to gethostbyaddr or gethostbyname.

Examples

The following examples demonstrates the use of the **hostent** structure with the gethostbyname function.

```
#define WIN32_LEAN_AND_MEAN

#include <winsock2.h>
#include <ws2tcpip.h>
#include <stdio.h>

// Need to link with Ws2_32.lib
#pragma comment(lib, "ws2_32.lib")

int main(int argc, char **argv)
{

    //-----------------------------------------
    // Declare and initialize variables
    WSADATA wsaData;
    int iResult;

    DWORD dwError;
    int i = 0;

    struct hostent *remoteHost;
    char *host_name;
    struct in_addr addr;

    char **pAlias;

    // Validate the parameters
    if (argc != 2) {
        printf("usage: %s ipv4address\n", argv[0]);
        printf(" or\n");
        printf("       %s hostname\n", argv[0]);
        printf("  to return the host\n");
        printf("       %s 127.0.0.1\n", argv[0]);
        printf("  to return the IP addresses for a host\n");
        printf("       %s www.contoso.com\n", argv[0]);
        return 1;
    }
    // Initialize Winsock
    iResult = WSAStartup(MAKEWORD(2, 2), &wsaData);
    if (iResult != 0) {
        printf("WSAStartup failed: %d\n", iResult);
        return 1;
    }

    host_name = argv[1];

// If the user input is an alpha name for the host, use gethostbyname()
// If not, get host by addr (assume IPv4)
    if (isalpha(host_name[0])) {        /* host address is a name */
        printf("Calling gethostbyname with %s\n", host_name);
        remoteHost = gethostbyname(host_name);
    } else {
        printf("Calling gethostbyaddr with %s\n", host_name);
        addr.s_addr = inet_addr(host_name);
```

```
        if (addr.s_addr == INADDR_NONE) {
            printf("The IPv4 address entered must be a legal address\n");
            return 1;
        } else
            remoteHost = gethostbyaddr((char *) &addr, 4, AF_INET);
    }

    if (remoteHost == NULL) {
        dwError = WSAGetLastError();
        if (dwError != 0) {
            if (dwError == WSAHOST_NOT_FOUND) {
                printf("Host not found\n");
                return 1;
            } else if (dwError == WSANO_DATA) {
                printf("No data record found\n");
                return 1;
            } else {
                printf("Function failed with error: %ld\n", dwError);
                return 1;
            }
        }
    } else {
        printf("Function returned:\n");
        printf("\tOfficial name: %s\n", remoteHost->h_name);
        for (pAlias = remoteHost->h_aliases; *pAlias != 0; pAlias++) {
            printf("\tAlternate name #%d: %s\n", ++i, *pAlias);
        }
        printf("\tAddress type: ");
        switch (remoteHost->h_addrtype) {
        case AF_INET:
            printf("AF_INET\n");
            break;
        case AF_INET6:
            printf("AF_INET6\n");
            break;
        case AF_NETBIOS:
            printf("AF_NETBIOS\n");
            break;
        default:
            printf(" %d\n", remoteHost->h_addrtype);
            break;
        }
        printf("\tAddress length: %d\n", remoteHost->h_length);

        if (remoteHost->h_addrtype == AF_INET) {
            while (remoteHost->h_addr_list[i] != 0) {
                addr.s_addr = *(u_long *) remoteHost->h_addr_list[i++];
                printf("\tIPv4 Address #%d: %s\n", i, inet_ntoa(addr));
            }
        } else if (remoteHost->h_addrtype == AF_INET6)
            printf("\tRemotehost is an IPv6 address\n");
    }

    return 0;
}
```

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]

Header	winsock2.h (include Winsock2.h)

See also

GetAddrInfoEx

GetAddrInfoW

addrinfo

addrinfoW

getaddrinfo

gethostbyaddr

gethostbyname

htond function (winsock2.h)

10/22/2022 • 2 minutes to read • Edit Online

The **htond** inline function converts a **double** from host to TCP/IP network byte order (which is big-endian).

Syntax

```
unsigned __int64 htond(
  double Value
);
```

Parameters

`Value`

A **double** that contains a number in host byte order.

Return value

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

Remarks

The **htond** inline function takes a **double** that contains number in host byte order and returns a 64-bit unsigned number in the network byte order used in TCP/IP networks (the AF_INET or AF_INET6 address family).

The **htond** inline function can be used to convert an IPv4 address in host byte order to the IPv4 address in network byte order. This function does not do any checking to determine if the *value* parameter is a valid IPv4 address.

The **htond** inline function does not require that the Winsock DLL has previously been loaded with a successful call to the WSAStartup function.

Windows Phone 8: This function is supported for Windows Phone Store apps on Windows Phone 8 and later.

Windows 8.1 and **Windows Server 2012 R2**: This function is supported for Windows Store apps on Windows 8.1, Windows Server 2012 R2, and later.

Requirements

Minimum supported client	Windows 8.1, Windows 8 [desktop apps | UWP apps]
Minimum supported server	Windows Server 2012 [desktop apps | UWP apps]
Target Platform	Windows
Header	winsock2.h

See also

InetNtop

InetPton

WSAHtonl

WSAHtons

WSANtohl

WSANtohs

htond

htonf

htonll

htons

inet_addr

inet_ntoa

ntohd

ntohf

ntohl

ntohll

ntohs

htonf function (winsock2.h)

10/22/2022 • 2 minutes to read • Edit Online

The **htonf** inline function converts a **float** from host to TCP/IP network byte order (which is big-endian).

Syntax

```
unsigned __int32 htonf(
  float Value
);
```

Parameters

`Value`

A float that contains a number in host byte order.

Return value

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

Remarks

The **htonf** inline function takes a **float** that contains number in host byte order and returns a 32-bit unsigned number in the network byte order used in TCP/IP networks (the AF_INET or AF_INET6 address family).

The **htonf** inline function can be used to convert an IPv4 address in host byte order to the IPv4 address in network byte order. This function does not do any checking to determine if the *value* parameter is a valid IPv4 address.

The **htonf** inline function does not require that the Winsock DLL has previously been loaded with a successful call to the WSAStartup function.

Windows Phone 8: This function is supported for Windows Phone Store apps on Windows Phone 8 and later.

Windows 8.1 and **Windows Server 2012 R2**: This function is supported for Windows Store apps on Windows 8.1, Windows Server 2012 R2, and later.

Requirements

Minimum supported client	Windows 8.1, Windows 8 [desktop apps | UWP apps]
Minimum supported server	Windows Server 2012 [desktop apps | UWP apps]
Target Platform	Windows
Header	winsock2.h

See also

InetNtop

InetPton

WSAHtonl

WSAHtons

WSANtohl

WSANtohs

htond

htonll

htons

inet_addr

inet_ntoa

ntohd

ntohf

ntohl

ntohll

ntohs

htonl function (winsock2.h)

10/22/2022 • 2 minutes to read • Edit Online

The **htonl** function converts a **u_long** from host to TCP/IP network byte order (which is big-endian).

Syntax

```
u_long WSAAPI htonl(
  [in] u_long hostlong
);
```

Parameters

`[in] hostlong`

A 32-bit number in host byte order.

Return value

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

Remarks

The **htonl** function takes a 32-bit number in host byte order and returns a 32-bit number in the network byte order used in TCP/IP networks (the AF_INET or AF_INET6 address family).

The **htonl** function can be used to convert an IPv4 address in host byte order to the IPv4 address in network byte order. This function does not do any checking to determine if the *hostlong* parameter is a valid IPv4 address.

The **htonl** function does not require that the Winsock DLL has previously been loaded with a successful call to the WSAStartup function.

Windows 8.1 and **Windows Server 2012 R2**: This function is supported for Windows Store apps on Windows 8.1, Windows Server 2012 R2, and later.

Requirements

Minimum supported client	Windows 8.1, Windows Vista [desktop apps | UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps | UWP apps]
Target Platform	Windows
Header	winsock2.h (include Winsock2.h)
Library	Ws2_32.lib

DLL	Ws2_32.dll

See also

InetNtop

InetPton

WSAHtonl

WSAHtons

WSANtohl

WSANtohs

htond

htonf

htonll

htons

inet_addr

inet_ntoa

ntohd

ntohf

ntohll

ntohs

htonll function (winsock2.h)

10/22/2022 • 2 minutes to read • Edit Online

The **htonll** inline function converts an **unsigned __int64** from host to TCP/IP network byte order (which is big-endian).

Syntax

```
unsigned __int64 htonll(
  unsigned __int64 Value
);
```

Parameters

`Value`

A 64-bit unsigned number in host byte order.

Return value

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

Remarks

The **htonll** inline function takes a 64-bit number in host byte order and returns a 64-bit number in the network byte order used in TCP/IP networks (the AF_INET or AF_INET6 address family).

The **htonll** inline function can be used to convert an IPv4 address in host byte order to the IPv4 address in network byte order. This function does not do any checking to determine if the *value* parameter is a valid IPv4 address.

The **htonll** inline function does not require that the Winsock DLL has previously been loaded with a successful call to the WSAStartup function.

Windows Phone 8: This function is supported for Windows Phone Store apps on Windows Phone 8 and later.

Windows 8.1 and **Windows Server 2012 R2**: This function is supported for Windows Store apps on Windows 8.1, Windows Server 2012 R2, and later.

Requirements

Minimum supported client	Windows 8.1, Windows 8 [desktop apps | UWP apps]
Minimum supported server	Windows Server 2012 [desktop apps | UWP apps]
Target Platform	Windows
Header	winsock2.h

See also

InetNtop

InetPton

WSAHtonl

WSAHtons

WSANtohl

WSANtohs

htond

htonf

htonl

htons

inet_addr

inet_ntoa

ntohd

ntohf

ntohl

ntohll

ntohs

htons function (winsock2.h)

10/22/2022 • 2 minutes to read • Edit Online

The **htons** function converts a **u_short** from host to TCP/IP network byte order (which is big-endian).

Syntax

```
u_short WSAAPI htons(
  [in] u_short hostshort
);
```

Parameters

`[in] hostshort`

A 16-bit number in host byte order.

Return value

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

Remarks

The **htons** function takes a 16-bit number in host byte order and returns a 16-bit number in network byte order used in TCP/IP networks (the AF_INET or AF_INET6 address family).

The **htons** function can be used to convert an IP port number in host byte order to the IP port number in network byte order.

The **htons** function does not require that the Winsock DLL has previously been loaded with a successful call to the WSAStartup function.

Windows 8.1 and **Windows Server 2012 R2**: This function is supported for Windows Store apps on Windows 8.1, Windows Server 2012 R2, and later.

Requirements

Minimum supported client	Windows 8.1, Windows Vista [desktop apps | UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps | UWP apps]
Target Platform	Windows
Header	winsock2.h (include Winsock2.h)
Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

InetNtop

InetPton

WSAHtonl

WSAHtons

WSANtohl

WSANtohs

htond

htonf

htonl

htonll

inet_addr

inet_ntoa

ntohd

ntohf

ntohl

ntohll

ntohs

in_addr structure (winsock2.h)

10/22/2022 • 2 minutes to read • Edit Online

The **in_addr** structure represents an IPv4 Internet address.

Syntax

```
struct in_addr {
  union {
    struct {
      u_char s_b1;
      u_char s_b2;
      u_char s_b3;
      u_char s_b4;
    } S_un_b;
    struct {
      u_short s_w1;
      u_short s_w2;
    } S_un_w;
    u_long S_addr;
  } S_un;
};
```

Members

`S_un`

`S_un.S_un_b`

An IPv4 address formatted as four **u_char**s.

`S_un.S_un_b.s_b1`

`S_un.S_un_b.s_b2`

`S_un.S_un_b.s_b3`

`S_un.S_un_b.s_b4`

`S_un.S_un_w`

An IPv4 address formatted as two **u_short**s.

`S_un.S_un_w.s_w1`

`S_un.S_un_w.s_w2`

`S_un.S_addr`

An IPv4 address formatted as a **u_long**.

Remarks

The **in_addr** structure is used with IPv4 addresses.

The **in_addr** structure is the IPv4 equivalent of the IPv6-based in6_addr structure.

> **Note** The **IN_ADDR**, **PIN_ADDR**, and **LPIN_ADDR** derived structures are only defined on the Windows SDK released with Windows Vista and later. The **IN_ADDR**, **PIN_ADDR**, and **LPIN_ADDR** derived structures are defined in the *Inaddr.h* header file. On earlier versions of the Windows SDK, variables of this type should be declared as **struct in_addr**.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	winsock2.h

See also

in6_addr

inet_addr

inet_ntoa

sockaddr

inet_addr function (winsock2.h)

10/22/2022 • 4 minutes to read • Edit Online

The **inet_addr** function converts a string containing an IPv4 dotted-decimal address into a proper address for the IN_ADDR structure.

Syntax

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

Parameters

`cp`

TBD

Return value

If no error occurs, the **inet_addr** function returns an unsigned long value containing a suitable binary representation of the Internet address given.

If the string in the *cp* parameter does not contain a legitimate Internet address, for example if a portion of an "a.b.c.d" address exceeds 255, then **inet_addr** returns the value **INADDR_NONE**.

On Windows Server 2003 and later if the string in the *cp* parameter is an empty string, then **inet_addr** returns the value **INADDR_NONE**. If **NULL** is passed in the *cp* parameter, then **inet_addr** returns the value **INADDR_NONE**.

On Windows XP and earlier if the string in the *cp* parameter is an empty string, then **inet_addr** returns the value **INADDR_ANY**. If **NULL** is passed in the *cp* parameter, then **inet_addr** returns the value **INADDR_NONE**.

Remarks

The **inet_addr** function interprets the character string specified by the *cp* parameter. This string represents a numeric Internet address expressed in the Internet standard "."' notation. The value returned is a number suitable for use as an Internet address. All Internet addresses are returned in IP's network order (bytes ordered from left to right). If you pass in " " (a space) to the **inet_addr** function, **inet_addr** returns zero.

On Windows Vista and later, the RtlIpv4StringToAddress function can be used to convert a string representation of an IPv4 address to a binary IPv4 address represented as an IN_ADDR structure. On Windows Vista and later, the RtlIpv6StringToAddress function can be used to convert a string representation of an IPv6 address to a binary IPv6 address represented as an **IN6_ADDR** structure.

Internet Addresses

Values specified using the "."' notation take one of the following forms:

a.b.c.d a.b.c a.b a

When four parts are specified, each is interpreted as a byte of data and assigned, from left to right, to the 4 bytes

of an Internet address. When an Internet address is viewed as a 32-bit integer quantity on the Intel architecture, the bytes referred to above appear as "d.c.b.a''. That is, the bytes on an Intel processor are ordered from right to left.

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

INTERNET ADDRESS VALUE	MEANING
"4.3.2.16"	Decimal
"004.003.002.020"	Octal
"0x4.0x3.0x2.0x10"	Hexadecimal
"4.003.002.0x10"	Mix

The **inet_addr** function supports the decimal, octal, hexadecimal, and mixed notations for the string passed in the *cp* parameter.

> **Note** The following notations are only used by Berkeley software, and nowhere else on the Internet. For compatibility with Berkeley software, the **inet_addr** function also supports the additional notations specified below.

When a three-part address is specified, the last part is interpreted as a 16-bit quantity and placed in the right-most 2 bytes of the network address. This makes the three-part address format convenient for specifying Class B network addresses as "128.net.host''

When a two-part address is specified, the last part is interpreted as a 24-bit quantity and placed in the right-most 3 bytes of the network address. This makes the two-part address format convenient for specifying Class A network addresses as "net.host''.

When only one part is given, the value is stored directly in the network address without any byte rearrangement.

Windows Phone 8: This function is supported for Windows Phone Store apps on Windows Phone 8 and later.

Windows 8.1 and **Windows Server 2012 R2**: This function is supported for Windows Store apps on Windows 8.1, Windows Server 2012 R2, and later.

Examples

The following code example shows how to use the **inet_addr** function.

```
#define WIN32_LEAN_AND_MEAN

#include <winsock2.h>
#include <stdio.h>
#include <windows.h>


// need link with Ws2_32.lib
#pragma comment(lib, "Ws2_32.lib")

int __cdecl main(int argc, char **argv)
{

    //-----------------------------------------
    // Declare and initialize variables
    WSADATA wsaData;
    int iResult;

    unsigned long ulAddr = INADDR_NONE;

    // Validate the parameters
    if (argc != 2) {
        printf("usage: %s <IPv4 address>\n", argv[0]);
        printf("  inetaddr converts a string containing an\n");
        printf("  IPv4 address in one of the supported formats\n");
        printf("  to a unsigned long representing an IN_ADDR\n");
        printf("      %s 192.168.16.34\n", argv[0]);
        return 1;
    }
    // Initialize Winsock
    iResult = WSAStartup(MAKEWORD(2, 2), &wsaData);
    if (iResult != 0) {
        printf("WSAStartup failed: %d\n", iResult);
        return 1;
    }

//--------------------------------
// Call inet_addr(). If the call succeeds,
// the result variable will hold a IN_ADDR
    ulAddr = inet_addr(argv[1]);
    if ( ulAddr == INADDR_NONE ) {
        printf("inet_addr failed and returned INADDR_NONE\n");
        WSACleanup();
        return 1;
    }

    if (ulAddr == INADDR_ANY) {
        printf("inet_addr failed and returned INADDR_ANY\n");
        WSACleanup();
        return 1;
    }

    printf("inet_addr returned success\n");

    // Here we could implement code to retrieve each address and
    // print out the hex bytes
    // for(i=0, ptr= (Char*) &ulAddr; i < 4; i++, ptr++) {

    WSACleanup();
    return 0;
}
```

Requirements

Minimum supported client	Windows 8.1, Windows Vista [desktop apps | UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps | UWP apps]
Target Platform	Windows
Header	winsock2.h (include Winsock2.h, Winsock.h)
Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

IN6_ADDR

IN_ADDR

InetNtop

RtlIpv4AddressToString

RtlIpv4AddressToStringEx

RtlIpv4StringToAddress

RtlIpv4StringToAddressEx

RtlIpv6AddressToString

RtlIpv6AddressToStringEx

RtlIpv6StringToAddress

RtlIpv6StringToAddressEx

Winsock Functions

Winsock Reference

inet_ntoa

inet_ntoa function (winsock2.h)

10/22/2022 • 2 minutes to read • Edit Online

The **inet_ntoa** function converts an (Ipv4) Internet network address into an ASCII string in Internet standard dotted-decimal format.

Syntax

```
char *WSAAPI inet_ntoa(
  in_addr in
);
```

Parameters

`in`

TBD

Return value

If no error occurs, **inet_ntoa** returns a character pointer to a static buffer containing the text address in standard "."' notation. Otherwise, it returns **NULL**.

Remarks

The **inet_ntoa** function takes an Internet address structure specified by the *in* parameter and returns a **NULL**-terminated ASCII string that represents the address in "." (dot) notation as in "192.168.16.0", an example of an IPv4 address in dotted-decimal notation. The string returned by **inet_ntoa** resides in memory that is allocated by Windows Sockets. The application should not make any assumptions about the way in which the memory is allocated. The string returned is guaranteed to be valid only until the next Windows Sockets function call is made within the same thread. Therefore, the data should be copied before another Windows Sockets call is made.

The WSAAddressToString function can be used to convert a sockaddr structure containing an IPv4 address to a string representation of an IPv4 address in Internet standard dotted-decimal notation. The advantage of the **WSAAddressToString** function is that it supports both IPv4 and IPv6 addresses. Another advantage of the **WSAAddressToString** function is that there are both ASCII and Unicode versions of this function.

On Windows Vista and later, the RtlIpv4AddressToString function can be used to convert an IPv4 address represented as an IN_ADDR structure to a string representation of an IPv4 address in Internet standard dotted-decimal notation. On Windows Vista and later, the RtlIpv6AddressToString function can be used to convert an IPv6 address represented as an **IN6_ADDR** structure to a string representation of an IPv6 address.

Windows Phone 8: This function is supported for Windows Phone Store apps on Windows Phone 8 and later.

Windows 8.1 and **Windows Server 2012 R2**: This function is supported for Windows Store apps on Windows 8.1, Windows Server 2012 R2, and later.

Requirements

Minimum supported client	Windows 8.1, Windows Vista [desktop apps | UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps | UWP apps]
Target Platform	Windows
Header	winsock2.h (include Winsock2.h, Winsock.h)
Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

IN6_ADDR

InetNtop

RtlIpv4AddressToString

RtlIpv4AddressToStringEx

RtlIpv4StringToAddress

RtlIpv4StringToAddressEx

RtlIpv6AddressToString

RtlIpv6AddressToStringEx

RtlIpv6StringToAddress

RtlIpv6StringToAddressEx

SOCKADDR

WSAAddressToString

Winsock Functions

Winsock Reference

in_addr

inet_addr

ioctlsocket function (winsock2.h)

10/22/2022 • 2 minutes to read • Edit Online

The **ioctlsocket** function controls the I/O mode of a socket.

Syntax

```
int WSAAPI ioctlsocket(
  [in]      SOCKET s,
  [in]      long   cmd,
  [in, out] u_long *argp
);
```

Parameters

`[in] s`

A descriptor identifying a socket.

`[in] cmd`

A command to perform on the socket *s*.

`[in, out] argp`

A pointer to a parameter for *cmd*.

Return value

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

ERROR CODE	MEANING
WSANOTINITIALISED	A successful WSAStartup call must occur before using this function.
WSAENETDOWN	The network subsystem has failed.
WSAEINPROGRESS	A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function.
WSAENOTSOCK	The descriptor *s* is not a socket.
WSAEFAULT	The *argp* parameter is not a valid part of the user address space.

Remarks

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

The WSAIoctl function is used to set or retrieve operating parameters associated with the socket, the transport protocol, or the communications subsystem.

The **WSAIoctl** function is more powerful than the **ioctlsocket** function and supports a large number of possible values for the operating parameters to set or retrieve.

Example Code

The following example demonstrates the use of the **ioctlsocket** function.

```
#include <winsock2.h>
#include <stdio.h>

#pragma comment(lib, "Ws2_32.lib")

void main()
{
//-------------------------
// Initialize Winsock
WSADATA wsaData;
int iResult;
u_long iMode = 0;

iResult = WSAStartup(MAKEWORD(2,2), &wsaData);
if (iResult != NO_ERROR)
  printf("Error at WSAStartup()\n");

//-------------------------
// Create a SOCKET object.
SOCKET m_socket;
m_socket = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
if (m_socket == INVALID_SOCKET) {
  printf("Error at socket(): %ld\n", WSAGetLastError());
  WSACleanup();
  return;
}

//-------------------------
// Set the socket I/O mode: In this case FIONBIO
// enables or disables the blocking mode for the
// socket based on the numerical value of iMode.
// If iMode = 0, blocking is enabled;
// If iMode != 0, non-blocking mode is enabled.

iResult = ioctlsocket(m_socket, FIONBIO, &iMode);
if (iResult != NO_ERROR)
  printf("ioctlsocket failed with error: %ld\n", iResult);


}
```

Compatibility

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

Windows Phone 8: This function is supported for Windows Phone Store apps on Windows Phone 8 and later.

Windows 8.1 and **Windows Server 2012 R2**: This function is supported for Windows Store apps on

Windows 8.1, Windows Server 2012 R2, and later.

Requirements

Minimum supported client	Windows 8.1, Windows Vista [desktop apps | UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps | UWP apps]
Target Platform	Windows
Header	winsock2.h (include Winsock2.h)
Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

WSAAsyncSelect

WSAEventSelect

WSAIoctl

Winsock Functions

Winsock Reference

getsockopt

setsockopt

socket

LINGER structure (winsock2.h)

10/22/2022 • 3 minutes to read • Edit Online

The **linger** structure maintains information about a specific socket that specifies how that socket should behave when data is queued to be sent and the closesocket function is called on the socket.

Syntax

```
typedef struct linger {
  u_short l_onoff;
  u_short l_linger;
} LINGER, *PLINGER, *LPLINGER;
```

Members

`l_onoff`

Type: **u_short**

Specifies whether a socket should remain open for a specified amount of time after a closesocket function call to enable queued data to be sent. This member can have one of the following values.

VALUE	MEANING
0	The socket will not remain open. This is the value set if the setsockopt function is called with the *optname* parameter set to **SO_DONTLINGER** and the *optval* parameter is zero. This value is also set if the setsockopt function is called with the *optname* parameter set to **SO_LINGER** and the **linger** structure passed in the *optval* parameter has the **l_onoff** member set to 0.
nonzero	The socket will remain open for a specified amount of time. This value is set if the setsockopt function is called with the *optname* parameter set to **SO_DONTLINGER** and the *optval* parameter is nonzero. This value is also set if the setsockopt function is called with the *optname* parameter set to **SO_LINGER** and the **linger** structure passed in the *optval* parameter has the **l_onoff** member set to a nonzero value.

`l_linger`

Type: **u_short**

The linger time in seconds. This member specifies how long to remain open after a closesocket function call to enable queued data to be sent. This member is only applicable if the **l_onoff** member of the **linger** structure is set to a nonzero value.

This value is set if the setsockopt function is called with the *optname* parameter set to **SO_LINGER**. The *optval* parameter passed to the **setsockopt** function must contain a **linger** structure that is copied to the internal **linger** structure maintained for the socket.

Remarks

The **l_onoff** member of the **linger** structure determines whether a socket should remain open for a specified amount of time after a closesocket function call to enable queued data to be sent. Somewhat confusing is that this member can be modified in two ways:

- Call the setsockopt function with the *optname* parameter set to **SO_DONTLINGER**. The *optval* parameter determines how the **l_onoff** member is modified.
- Call the setsockopt function with the *optname* parameter set to **SO_LINGER**. The *optval* parameter specifies how both the **l_onoff** and **l_linger** members are modified.

The **l_linger** member of the **linger** structure determines the amount of time, in seconds, a socket should remain open. This member is only applicable if the **l_onoff** member of the **linger** structure is nonzero.

To enable a socket to remain open, an application should set the **l_onoff** member to a nonzero value and set the **l_linger** member to the desired time-out in seconds. To disable a socket from remaining open, an application only needs to set the **l_onoff** member of the **linger** structure to zero.

If an application calls the setsockopt function with the *optname* parameter set to **SO_DONTLINGER** to set the **l_onoff** member to a nonzero value, the value for the **l_linger** member is not specified. In this case, the time-out used is implementation dependent. If a previous time-out has been established for a socket (by enabling SO_LINGER), this time-out value should be reinstated by the service provider.

Note that enabling a nonzero timeout on a nonblocking socket is not recommended.

The getsockopt function can be called with the *optname* parameter set to **SO_LINGER** to retrieve the current value of the **linger** structure associated with a socket.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	winsock2.h (include Winsock2.h)

See also

Graceful Shutdown, Linger Options, and Socket Closure

closesocket

getsockopt

setsockopt

listen function (winsock2.h)

10/22/2022 • 5 minutes to read • Edit Online

The **listen** function places a socket in a state in which it is listening for an incoming connection.

Syntax

```
int WSAAPI listen(
  [in] SOCKET s,
  [in] int    backlog
);
```

Parameters

`[in] s`

A descriptor identifying a bound, unconnected socket.

`[in] backlog`

The maximum length of the queue of pending connections. If set to **SOMAXCONN**, the underlying service provider responsible for socket *s* will set the backlog to a maximum reasonable value. If set to **SOMAXCONN_HINT(N)** (where N is a number), the backlog value will be N, adjusted to be within the range (200, 65535). Note that **SOMAXCONN_HINT** can be used to set the backlog to a larger value than possible with SOMAXCONN.

SOMAXCONN_HINT is only supported by the Microsoft TCP/IP service provider. There is no standard provision to obtain the actual backlog value.

Return value

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

ERROR CODE	MEANING
WSANOTINITIALISED	A successful WSAStartup call must occur before using this function.
WSAENETDOWN	The network subsystem has failed.
WSAEADDRINUSE	The socket's local address is already in use and the socket was not marked to allow address reuse with SO_REUSEADDR. This error usually occurs during execution of the bind function, but could be delayed until this function if the **bind** was to a partially wildcard address (involving ADDR_ANY) and if a specific address needs to be committed at the time of this function.

WSAEINPROGRESS	A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function.
WSAEINVAL	The socket has not been bound with bind.
WSAEISCONN	The socket is already connected.
WSAEMFILE	No more socket descriptors are available.
WSAENOBUFS	No buffer space is available.
WSAENOTSOCK	The descriptor is not a socket.
WSAEOPNOTSUPP	The referenced socket is not of a type that supports the listen operation.

Remarks

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

A value for the *backlog* of **SOMAXCONN** is a special constant that instructs the underlying service provider responsible for socket *s* to set the length of the queue of pending connections to a maximum reasonable value.

On Windows Sockets 2, this maximum value defaults to a large value (typically several hundred or more).

When calling the **listen** function in a Bluetooth application, it is strongly recommended that a much lower value be used for the *backlog* parameter (typically 2 to 4), since only a few client connections are accepted. This reduces the system resources that are allocated for use by the listening socket. This same recommendation applies to other network applications that expect only a few client connections.

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

If there are no available socket descriptors, **listen** attempts to continue to function. If descriptors become available, a later call to **listen** or accept will refill the queue to the current or most recent value specified for the *backlog* parameter, if possible, and resume listening for incoming connections.

If the **listen** function is called on an already listening socket, it will return success without changing the value for the *backlog* parameter. Setting the *backlog* parameter to 0 in a subsequent call to **listen** on a listening socket is not considered a proper reset, especially if there are connections on the socket.

> **Note** When issuing a blocking Winsock call such as **listen**, Winsock may need to wait for a network event before the call can complete. Winsock performs an alertable wait in this situation, which can be interrupted by an asynchronous procedure call (APC) scheduled on the same thread. Issuing another blocking Winsock call inside an APC that interrupted an ongoing blocking Winsock call on the same thread will lead to undefined behavior, and must never be attempted by Winsock clients.

Example Code

The following example demonstrates the use of the **listen** function.

```
#ifndef UNICODE
#define UNICODE
#endif

#define WIN32_LEAN_AND_MEAN

#include <winsock2.h>
#include <ws2tcpip.h>
#include <stdio.h>

// Need to link with Ws2_32.lib
#pragma comment(lib, "ws2_32.lib")

int wmain()
{

    //-----------------------
    // Initialize Winsock
    WSADATA wsaData;
    int iResult = 0;

    SOCKET ListenSocket = INVALID_SOCKET;
    sockaddr_in service;

    iResult = WSAStartup(MAKEWORD(2, 2), &wsaData);
    if (iResult != NO_ERROR) {
        wprintf(L"WSAStartup() failed with error: %d\n", iResult);
        return 1;
    }
    //-----------------------
    // Create a SOCKET for listening for incoming connection requests.
    ListenSocket = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
    if (ListenSocket == INVALID_SOCKET) {
        wprintf(L"socket function failed with error: %ld\n", WSAGetLastError());
        WSACleanup();
        return 1;
    }
    //-----------------------
    // The sockaddr_in structure specifies the address family,
    // IP address, and port for the socket that is being bound.
    service.sin_family = AF_INET;
    service.sin_addr.s_addr = inet_addr("127.0.0.1");
    service.sin_port = htons(27015);

    iResult = bind(ListenSocket, (SOCKADDR *) & service, sizeof (service));
    if (iResult == SOCKET_ERROR) {
        wprintf(L"bind function failed with error %d\n", WSAGetLastError());
        iResult = closesocket(ListenSocket);
        if (iResult == SOCKET_ERROR)
            wprintf(L"closesocket function failed with error %d\n", WSAGetLastError());
        WSACleanup();
        return 1;
    }
    //-----------------------
    // Listen for incoming connection requests
```

```
    // on the created socket
    if (listen(ListenSocket, SOMAXCONN) == SOCKET_ERROR)
        wprintf(L"listen function failed with error: %d\n", WSAGetLastError());

    wprintf(L"Listening on socket...\n");

    iResult = closesocket(ListenSocket);
    if (iResult == SOCKET_ERROR) {
        wprintf(L"closesocket function failed with error %d\n", WSAGetLastError());
        WSACleanup();
        return 1;
    }

    WSACleanup();
    return 0;
}
```

Example Code

For another example that uses the **listen** function, see Getting Started With Winsock.

Notes for IrDA Sockets

- The Af_irda.h header file must be explicitly included.

Compatibility

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

Windows Phone 8: This function is supported for Windows Phone Store apps on Windows Phone 8 and later.

Windows 8.1 and **Windows Server 2012 R2**: This function is supported for Windows Store apps on Windows 8.1, Windows Server 2012 R2, and later.

Requirements

Minimum supported client	Windows 8.1, Windows Vista [desktop apps | UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps | UWP apps]
Target Platform	Windows
Header	winsock2.h
Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

Winsock Functions

Winsock Reference

accept

connect

socket

LPWSAOVERLAPPED_COMPLETION_ROUTINE callback function (winsock2.h)

10/22/2022 • 2 minutes to read • Edit Online

LPWSAOVERLAPPED_COMPLETION_ROUTINE is a function pointer type. You implement a matching callback function in your app, and pass that to functions such as WSAIoctl, WSARecv, and WSASend, among others.

The system calls your callback function when the asynchronous input and output (I/O) operation is completed or canceled, and the calling thread is in an alertable state (by using the SleepEx, MsgWaitForMultipleObjectsEx, WaitForSingleObjectEx, or WaitForMultipleObjectsEx function with the *fAlertable* parameter set to **TRUE**).

Syntax

```
LPWSAOVERLAPPED_COMPLETION_ROUTINE LpwsaoverlappedCompletionRoutine;

void LpwsaoverlappedCompletionRoutine(
  DWORD dwError,
  DWORD cbTransferred,
  LPWSAOVERLAPPED lpOverlapped,
  DWORD dwFlags
)
{...}
```

Parameters

`dwError`

Type: IN **DWORD**

The I/O completion status. This parameter can be one of the system error codes.

`cbTransferred`

Type: IN **DWORD**

The number of bytes transferred. If an error occurs, this parameter is zero.

`lpOverlapped`

Type: IN **LPWSAOVERLAPPED**

A pointer to the **WSAOVERLAPPED** structure specified by the asynchronous I/O function.

The system doesn't use the **WSAOVERLAPPED** structure after the completion routine is called, so the completion routine can deallocate the memory used by the overlapped structure.

`dwFlags`

Type: IN **DWORD**

Flags associated with the call.

Return value

None

Remarks

See LPOVERLAPPED_COMPLETION_ROUTINE.

Requirements

Minimum supported client	Windows 10 Build 20348
Minimum supported server	Windows 10 Build 20348
Header	winsock2.h

ntohd function (winsock2.h)

10/22/2022 • 2 minutes to read • Edit Online

The **ntohd** inline function converts an **unsigned __int64** from TCP/IP network order to host byte order (which is little-endian on Intel processors) and returns a **double**.

Syntax

```
double ntohd(
  unsigned __int64 Value
);
```

Parameters

`Value`

An **unsigned __int64** number in TCP/IP network byte order.

Return value

The **ntohd** function returns the value supplied in the *value* parameter with the byte order reversed. If *value* is already in host byte order, then this function will reverse it. It is up to the application to determine if the byte order must be reversed.

Remarks

The **ntohd** inline function takes an **unsigned __int64** that contains number in TCP/IP network byte order (the AF_INET or AF_INET6 address family) and returns a **double** that contains a number in host byte order.

The **ntohd** function can be used to convert an IPv4 address in network byte order to the IPv4 address in host byte order. This function does not do any checking to determine if the *value* parameter is a valid IPv4 address.

The **ntohd** function does not require that the Winsock DLL has previously been loaded with a successful call to the WSAStartup function.

Windows Phone 8: This function is supported for Windows Phone Store apps on Windows Phone 8 and later.

Windows 8.1 and **Windows Server 2012 R2**: This function is supported for Windows Store apps on Windows 8.1, Windows Server 2012 R2, and later.

Requirements

Minimum supported client	Windows 8.1, Windows 8 [desktop apps | UWP apps]
Minimum supported server	Windows Server 2012 [desktop apps | UWP apps]
Target Platform	Windows

Header	winsock2.h

See also

InetNtop

InetPton

WSAHtonl

WSAHtons

WSANtohl

WSANtohs

htond

htonf

htonl

htonll

htons

inet_addr

inet_ntoa

ntohl

ntohll

ntohs

ntohf function (winsock2.h)

10/22/2022 • 2 minutes to read • Edit Online

The **ntohf** inline function converts an **unsigned __int32** from TCP/IP network order to host byte order (which is little-endian on Intel processors) and returns a **float**.

Syntax

```
float ntohf(
  unsigned __int32 Value
);
```

Parameters

`Value`

An **unsigned __int32** number in TCP/IP network byte order.

Return value

The **ntohf** function returns the value supplied in the *value* parameter with the byte order reversed. If *value* is already in host byte order, then this function will reverse it. It is up to the application to determine if the byte order must be reversed.

Remarks

The **ntohf** inline function takes an **unsigned __int32** in TCP/IP network byte order (the AF_INET or AF_INET6 address family) and returns a **float** that contains a number in host byte order.

The **ntohf** function can be used to convert an IPv4 address in network byte order to the IPv4 address in host byte order. This function does not do any checking to determine if the *value* parameter is a valid IPv4 address.

The **ntohf** function does not require that the Winsock DLL has previously been loaded with a successful call to the WSAStartup function.

Windows Phone 8: This function is supported for Windows Phone Store apps on Windows Phone 8 and later.

Windows 8.1 and **Windows Server 2012 R2**: This function is supported for Windows Store apps on Windows 8.1, Windows Server 2012 R2, and later.

Requirements

Minimum supported client	Windows 8.1, Windows 8 [desktop apps | UWP apps]
Minimum supported server	Windows Server 2012 [desktop apps | UWP apps]
Target Platform	Windows

Header	winsock2.h

See also

InetNtop

InetPton

WSAHtonl

WSAHtons

WSANtohl

WSANtohs

htond

htonf

htonl

htonll

htons

inet_addr

inet_ntoa

ntohd

ntohl

ntohll

ntohs

ntohl function (winsock2.h)

10/22/2022 • 2 minutes to read • Edit Online

The **ntohl** function converts a **u_long** from TCP/IP network order to host byte order (which is little-endian on Intel processors).

Syntax

```
u_long WSAAPI ntohl(
  [in] u_long netlong
);
```

Parameters

`[in] netlong`

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

Return value

The **ntohl** function returns the value supplied in the *netlong* parameter with the byte order reversed. If *netlong* is already in host byte order, then this function will reverse it. It is up to the application to determine if the byte order must be reversed.

Remarks

The **ntohl** function takes a 32-bit number in TCP/IP network byte order (the AF_INET or AF_INET6 address family) and returns a 32-bit number in host byte order.

The **ntohl** function can be used to convert an IPv4 address in network byte order to the IPv4 address in host byte order. This function does not do any checking to determine if the *netlong* parameter is a valid IPv4 address.

The **ntohl** function does not require that the Winsock DLL has previously been loaded with a successful call to the WSAStartup function.

Windows 8.1 and **Windows Server 2012 R2**: This function is supported for Windows Store apps on Windows 8.1, Windows Server 2012 R2, and later.

Requirements

Minimum supported client	Windows 8.1, Windows Vista [desktop apps | UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps | UWP apps]
Target Platform	Windows
Header	winsock2.h (include Winsock2.h)

Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

InetNtop

InetPton

WSAHtonl

WSAHtons

WSANtohl

WSANtohs

htond

htonf

htonl

htonll

htons

inet_addr

inet_ntoa

ntohd

ntohf

ntohs

ntohll function (winsock2.h)

10/22/2022 • 2 minutes to read • Edit Online

The **ntohll** inline function converts an **unsigned __int64** from TCP/IP network order to host byte order (which is little-endian on Intel processors).

Syntax

```
unsigned __int64 ntohll(
  unsigned __int64 Value
);
```

Parameters

`Value`

An **unsigned __int64** number in TCP/IP network byte order.

Return value

The **ntohll** function returns the value supplied in the *value* parameter with the byte order reversed. If *value* is already in host byte order, then this function will reverse it. It is up to the application to determine if the byte order must be reversed.

Remarks

The **ntohll** inline function takes an **unsigned __int64** number in TCP/IP network byte order (the AF_INET or AF_INET6 address family) and returns a 32-bit number in host byte order.

The **ntohll** function can be used to convert an IPv4 address in network byte order to the IPv4 address in host byte order. This function does not do any checking to determine if the *value* parameter is a valid IPv4 address.

The **ntohll** function does not require that the Winsock DLL has previously been loaded with a successful call to the WSAStartup function.

Windows Phone 8: This function is supported for Windows Phone Store apps on Windows Phone 8 and later.

Windows 8.1 and **Windows Server 2012 R2**: This function is supported for Windows Store apps on Windows 8.1, Windows Server 2012 R2, and later.

Requirements

Minimum supported client	Windows 8.1, Windows 8 [desktop apps | UWP apps]
Minimum supported server	Windows Server 2012 [desktop apps | UWP apps]
Target Platform	Windows

Header	winsock2.h

See also

InetNtop

InetPton

WSAHtonl

WSAHtons

WSANtohl

WSANtohs

htond

htonf

htonl

htonll

htons

inet_addr

inet_ntoa

ntohd

ntohf

ntohl

ntohs

ntohs function (winsock2.h)

10/22/2022 • 2 minutes to read • Edit Online

The **ntohs** function converts a **u_short** from TCP/IP network byte order to host byte order (which is little-endian on Intel processors).

Syntax

```
u_short WSAAPI ntohs(
  [in] u_short netshort
);
```

Parameters

`[in] netshort`

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

Return value

The **ntohs** function returns the value in host byte order. If the *netshort* parameter is already in host byte order, then this function will reverse it. It is up to the application to determine if the byte order must be reversed.

Remarks

The **ntohs** function takes a 16-bit number in TCP/IP network byte order (the AF_INET or AF_INET6 address family) and returns a 16-bit number in host byte order.

The **ntohs** function can be used to convert an IP port number in network byte order to the IP port number in host byte order.

The **ntohs** function does not require that the Winsock DLL has previously been loaded with a successful call to the WSAStartup function.

Windows 8.1 and **Windows Server 2012 R2**: This function is supported for Windows Store apps on Windows 8.1, Windows Server 2012 R2, and later.

Requirements

Minimum supported client	Windows 8.1, Windows Vista [desktop apps | UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps | UWP apps]
Target Platform	Windows
Header	winsock2.h (include Winsock2.h)
Library	Ws2_32.lib

DLL	Ws2_32.dll

See also

InetNtop

InetPton

WSAHtonl

WSAHtons

WSANtohl

WSANtohs

htond

htonf

htonl

htonll

htons

inet_addr

inet_ntoa

ntohd

ntohf

ntohl

ntohll

ProcessSocketNotifications function (winsock2.h)

10/22/2022 • 3 minutes to read • Edit Online

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. Only Microsoft Winsock provider sockets are supported.

To reduce system call overhead, you can register for notifications and retrieve them in a single call to **ProcessSocketNotifications**. Alternatively, you can retrieve them explicitly by calling the usual I/O completion port functions, such as GetQueuedCompletionStatus. Notifications retrieved using **ProcessSocketNotifications** are the same as those retrieved using GetQueuedCompletionStatusEx, which might include notification packets other than socket state changes.

The notification event flags are the integer value of the *dwNumberOfBytesTransferred* fields of the returned OVERLAPPED_ENTRY structures. This is similar to using JOBOBJECT_ASSOCIATE_COMPLETION_PORT, which also uses the *dwNumberOfBytesTransferred* field to send integer messages. Call the SocketNotificationRetrieveEvents function to obtain them.

A socket handle can be registered to only one IOCP at a time. Re-registering a previously-registered socket handle overwrites the existing registration. Before you close a handle used for registration, you should explicitly remove registration, and wait for the **SOCK_NOTIFY_EVENT_REMOVE** notification (see **Remarks** in this topic).

For more info, and code examples, see Winsock socket state notifications.

Syntax

```
DWORD WSAAPI ProcessSocketNotifications(
  HANDLE                   completionPort,
  UINT32                   registrationCount,
  SOCK_NOTIFY_REGISTRATION *registrationInfos,
  UINT32                   timeoutMs,
  ULONG                    completionCount,
  OVERLAPPED_ENTRY         *completionPortEntries,
  UINT32                   *receivedEntryCount
);
```

Parameters

`completionPort`

Type: _In_ **HANDLE**

A handle to an I/O completion port created using the CreateIoCompletionPort function. The port will be used in the *CompletionPort* parameter of the PostQueuedCompletionStatus function when messages are sent on behalf of the socket.

`registrationCount`

Type: _In_ **UINT32**

The number of registrations supplied by *registrationInfos*.

`registrationInfos`

Type: _Inout_updates_opt_(registrationCount) **SOCK_NOTIFY_REGISTRATION***

A pointer to an array of SOCK_NOTIFY_REGISTRATION structures that define the notification registration parameters. These include the socket of interest, the notification events of interest, and the operation flags. On success, you must inspect the elements for whether the registration was processed successfully. This argument must be **NULL** if *registrationCount* is 0.

`timeoutMs`

Type: _In_ **UINT32**

The time in milliseconds that you're willing to wait for a completion packet to appear at the completion port. If a completion packet doesn't appear within the specified time, then the function times out and returns **ERROR_TIMEOUT**.

If *timeoutMs* is **INFINITE** (0xFFFFFFFF), then the function will never time out. If *timeoutMs* is 0, and there is no I/O operation to dequeue, then the function will time out immediately.

The value of *timeoutMs* must be 0 if *completionCount* is 0.

`completionCount`

Type: _In_ **ULONG**

The maximum number of OVERLAPPED_ENTRY structures to remove. If 0 is specified, then only registration operations will be processed.

`completionPortEntries`

Type: _Out_writes_to_opt_(completionCount, *receivedEntryCount) **OVERLAPPED_ENTRY***

On input, points to a pre-allocated array of OVERLAPPED_ENTRY structures. The array mustn't overlap with the *registrationInfos* array. The value of *completionPortEntries* must be **NULL** if *completionCount* is 0.

On output, receives an array of OVERLAPPED_ENTRY structures that hold the entries. The number of array elements is provided by ReceivedEntryCount. The dwNumberOfBytesTransferred fields of the structures are integer masks of received events. The lpOverlapped fields are reserved and must not be used as pointers.

`receivedEntryCount`

Type: _Out_opt_ **UINT32***

A pointer to a variable that receives the number of entries removed. Must be **NULL** if *completionCount* is 0.

Return value

If successful, returns **ERROR_SUCCESS**. If the function succeeded and you supplied a non-0 *completionCount*, but no completion packets appeared within the specified time, returns **WAIT_TIMEOUT**. Otherwise, returns an appropriate **WSAE*** error code.

If **ERROR_SUCCESS** or **WAIT_TIMEOUT** is returned, then you must inspect the individual registration infos' registration results. Otherwise, the entire operation failed, and no changes occurred.

Remarks

See SocketNotificationRetrieveEvents for the events that are possible when a notification is received.

Requirements

Minimum supported client	Windows 10 Build 20348
Minimum supported server	Windows 10 Build 20348
Header	winsock2.h
Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

- CreateIoCompletionPort
- GetQueuedCompletionStatus
- GetQueuedCompletionStatusEx
- JOBOBJECT_ASSOCIATE_COMPLETION_PORT
- OVERLAPPED_ENTRY
- PostQueuedCompletionStatus
- SOCK_NOTIFY_REGISTRATION
- SocketNotificationRetrieveEvents
- Winsock
- Winsock socket state notifications

PROTOENT structure (winsock2.h)

10/22/2022 • 2 minutes to read • Edit Online

The **protoent** structure contains the name and protocol numbers that correspond to a given protocol name. Applications must never attempt to modify this structure or to free any of its components. Furthermore, only one copy of this structure is allocated per thread, and therefore, the application should copy any information it needs before issuing any other Windows Sockets function calls.

Syntax

```
typedef struct protoent {
  char  *p_name;
  char  **p_aliases;
  short p_proto;
} PROTOENT, *PPROTOENT, *LPPROTOENT;
```

Members

`p_name`

Official name of the protocol.

`p_aliases`

Null-terminated array of alternate names.

`p_proto`

Protocol number, in host byte order.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	winsock2.h (include Winsock2.h)

recv function (winsock2.h)

10/22/2022 • 7 minutes to read • Edit Online

The **recv** function receives data from a connected socket or a bound connectionless socket.

Syntax

```
int WSAAPI recv(
  [in]  SOCKET s,
  [out] char   *buf,
  [in]  int    len,
  [in]  int    flags
);
```

Parameters

`[in] s`

The descriptor that identifies a connected socket.

`[out] buf`

A pointer to the buffer to receive the incoming data.

`[in] len`

The length, in bytes, of the buffer pointed to by the *buf* parameter.

`[in] flags`

A set of flags that influences the behavior of this function. See remarks below. See the Remarks section for details on the possible value for this parameter.

Return value

If no error occurs, **recv** returns the number of bytes received and the buffer pointed to by the *buf* parameter will contain this data received. If the connection has been gracefully closed, the return value is zero.

Otherwise, a value of SOCKET_ERROR is returned, and a specific error code can be retrieved by calling WSAGetLastError.

ERROR CODE	MEANING
WSANOTINITIALISED	A successful WSAStartup call must occur before using this function.
WSAENETDOWN	The network subsystem has failed.
WSAEFAULT	The *buf* parameter is not completely contained in a valid part of the user address space.

WSAENOTCONN	The socket is not connected.
WSAEINTR	The (blocking) call was canceled through WSACancelBlockingCall.
WSAEINPROGRESS	A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function.
WSAENETRESET	For a connection-oriented socket, this error indicates that the connection has been broken due to *keep-alive* activity that detected a failure while the operation was in progress. For a datagram socket, this error indicates that the time to live has expired.
WSAENOTSOCK	The descriptor is not a socket.
WSAEOPNOTSUPP	MSG_OOB was specified, but the socket is not stream-style such as type SOCK_STREAM, OOB data is not supported in the communication domain associated with this socket, or the socket is unidirectional and supports only send operations.
WSAESHUTDOWN	The socket has been shut down; it is not possible to receive on a socket after shutdown has been invoked with **how** set to SD_RECEIVE or SD_BOTH.
WSAEWOULDBLOCK	The socket is marked as nonblocking and the receive operation would block.
WSAEMSGSIZE	The message was too large to fit into the specified buffer and was truncated.
WSAEINVAL	The socket has not been bound with bind, or an unknown flag was specified, or MSG_OOB was specified for a socket with SO_OOBINLINE enabled or (for byte stream sockets only) *len* was zero or negative.
WSAECONNABORTED	The virtual circuit was terminated due to a time-out or other failure. The application should close the socket as it is no longer usable.
WSAETIMEDOUT	The connection has been dropped because of a network failure or because the peer system failed to respond.
WSAECONNRESET	The virtual circuit was reset by the remote side executing a hard or abortive close. The application should close the socket as it is no longer usable. On a UDP-datagram socket, this error would indicate that a previous send operation resulted in an ICMP "Port Unreachable" message.

Remarks

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

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

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

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

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

If the datagram or message is larger than the buffer specified, the buffer is filled with the first part of the datagram, and **recv** generates the error WSAEMSGSIZE. For unreliable protocols (for example, UDP) the excess data is lost; for reliable protocols, the data is retained by the service provider until it is successfully read by calling **recv** with a large enough buffer.

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

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

The *flags* parameter can be used to influence the behavior of the function invocation beyond the options specified for the associated socket. The semantics of this function are determined by the socket options and the *flags* parameter. The possible value of *flags* parameter is constructed by using the bitwise OR operator with any of the following values.

VALUE	MEANING
MSG_PEEK	Peeks at the incoming data. The data is copied into the buffer, but is not removed from the input queue.
MSG_OOB	Processes Out Of Band (OOB) data.

MSG_WAITALL	The receive request will complete only when one of the following events occurs: • The buffer supplied by the caller is completely full. • The connection has been closed. • The request has been canceled or an error occurred. Note that if the underlying transport does not support MSG_WAITALL, or if the socket is in a non-blocking mode, then this call will fail with **WSAEOPNOTSUPP**. Also, if MSG_WAITALL is specified along with MSG_OOB, MSG_PEEK, or MSG_PARTIAL, then this call will fail with **WSAEOPNOTSUPP**. This flag is not supported on datagram sockets or message-oriented sockets.

> **Note** When issuing a blocking Winsock call such as **recv**, Winsock may need to wait for a network event before the call can complete. Winsock performs an alertable wait in this situation, which can be interrupted by an asynchronous procedure call (APC) scheduled on the same thread. Issuing another blocking Winsock call inside an APC that interrupted an ongoing blocking Winsock call on the same thread will lead to undefined behavior, and must never be attempted by Winsock clients.

Example Code

The following code example shows the use of the **recv** function.

```
#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")

#define DEFAULT_BUFLEN 512
#define DEFAULT_PORT "27015"

int __cdecl main() {

    //----------------------
    // Declare and initialize variables.
    WSADATA wsaData;
    int iResult;

    SOCKET ConnectSocket = INVALID_SOCKET;
    struct sockaddr_in clientService;

    char *sendbuf = "this is a test";
    char recvbuf[DEFAULT_BUFLEN];
    int recvbuflen = DEFAULT_BUFLEN;

    //----------------------
    // Initialize Winsock
    iResult = WSAStartup(MAKEWORD(2,2), &wsaData);
    if (iResult != NO_ERROR) {
      printf("WSAStartup failed: %d\n", iResult);
      return 1;
    }

    //----------------------
    // Create a SOCKET for connecting to server
    ConnectSocket = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
```

```
    if (ConnectSocket == INVALID_SOCKET) {
        printf("Error at socket(): %ld\n", WSAGetLastError() );
        WSACleanup();
        return 1;
    }

    //----------------------
    // The sockaddr_in structure specifies the address family,
    // IP address, and port of the server to be connected to.
    clientService.sin_family = AF_INET;
    clientService.sin_addr.s_addr = inet_addr( "127.0.0.1" );
    clientService.sin_port = htons( 27015 );

    //----------------------
    // Connect to server.
    iResult = connect( ConnectSocket, (SOCKADDR*) &clientService, sizeof(clientService) );
    if ( iResult == SOCKET_ERROR) {
        closesocket (ConnectSocket);
        printf("Unable to connect to server: %ld\n", WSAGetLastError());
        WSACleanup();
        return 1;
    }

    // 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 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;
    }

    // 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: %d\n", WSAGetLastError());

    } while( iResult > 0 );

    // cleanup
    closesocket(ConnectSocket);
    WSACleanup();

    return 0;
}
```

Example Code

For more information, and another example of the **recv** function, see Getting Started With Winsock.

Windows Phone 8: This function is supported for Windows Phone Store apps on Windows Phone 8 and later.

Windows 8.1 and **Windows Server 2012 R2**: This function is supported for Windows Store apps on Windows 8.1, Windows Server 2012 R2, and later.

Requirements

Minimum supported client	Windows 8.1, Windows Vista [desktop apps | UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps | UWP apps]
Target Platform	Windows
Header	winsock2.h (include Winsock2.h)
Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

WSAAsyncSelect

WSARecv

WSARecvEx

Winsock Functions

Winsock Reference

recvfrom

select

send

socket

recvfrom function (winsock2.h)

10/22/2022 • 7 minutes to read • Edit Online

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

Syntax

```
int WSAAPI recvfrom(
  [in]                SOCKET   s,
  [out]               char     *buf,
  [in]                int      len,
  [in]                int      flags,
  [out]               sockaddr *from,
  [in, out, optional] int      *fromlen
);
```

Parameters

`[in] s`

A descriptor identifying a bound socket.

`[out] buf`

A buffer for the incoming data.

`[in] len`

The length, in bytes, of the buffer pointed to by the *buf* parameter.

`[in] flags`

A set of options that modify the behavior of the function call beyond the options specified for the associated socket. See the Remarks below for more details.

`[out] from`

An optional pointer to a buffer in a sockaddr structure that will hold the source address upon return.

`[in, out, optional] fromlen`

An optional pointer to the size, in bytes, of the buffer pointed to by the *from* parameter.

Return value

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

ERROR CODE	MEANING
WSANOTINITIALISED	A successful WSAStartup call must occur before using this function.

WSAENETDOWN	The network subsystem has failed.
WSAEFAULT	The buffer pointed to by the *buf* or *from* parameters are not in the user address space, or the *fromlen* parameter is too small to accommodate the source address of the peer address.
WSAEINTR	The (blocking) call was canceled through WSACancelBlockingCall.
WSAEINPROGRESS	A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function.
WSAEINVAL	The socket has not been bound with bind, or an unknown flag was specified, or MSG_OOB was specified for a socket with SO_OOBINLINE enabled, or (for byte stream-style sockets only) *len* was zero or negative.
WSAEISCONN	The socket is connected. This function is not permitted with a connected socket, whether the socket is connection oriented or connectionless.
WSAENETRESET	For a datagram socket, this error indicates that the time to live has expired.
WSAENOTSOCK	The descriptor in the *s* parameter is not a socket.
WSAEOPNOTSUPP	MSG_OOB was specified, but the socket is not stream-style such as type SOCK_STREAM, OOB data is not supported in the communication domain associated with this socket, or the socket is unidirectional and supports only send operations.
WSAESHUTDOWN	The socket has been shut down; it is not possible to recvfrom on a socket after shutdown has been invoked with *how* set to SD_RECEIVE or SD_BOTH.
WSAEWOULDBLOCK	The socket is marked as nonblocking and the recvfrom operation would block.
WSAEMSGSIZE	The message was too large to fit into the buffer pointed to by the *buf* parameter and was truncated.
WSAETIMEDOUT	The connection has been dropped, because of a network failure or because the system on the other end went down without notice.
WSAECONNRESET	The virtual circuit was reset by the remote side executing a hard or abortive close. The application should close the socket; it is no longer usable. On a UDP-datagram socket this error indicates a previous send operation resulted in an ICMP *Port Unreachable* message.

Remarks

The **recvfrom** function reads incoming data on both connected and unconnected sockets and captures the address from which the data was sent. This function is typically used with connectionless sockets. The local address of the socket must be known. For server applications, this is usually done explicitly through bind. Explicit binding is discouraged for client applications. For client applications using this function, the socket can become bound implicitly to a local address through sendto, WSASendTo, or WSAJoinLeaf.

For stream-oriented sockets such as those of type SOCK_STREAM, a call to **recvfrom** returns as much information as is currently available—up to the size of the buffer specified. If the socket has been configured for inline reception of OOB data (socket option SO_OOBINLINE) and OOB data is yet unread, only OOB data will be returned. The application can use the ioctlsocket or WSAIoctl**SIOCATMARK** command to determine whether any more OOB data remains to be read. The *from* and *fromlen* parameters are ignored for connection-oriented sockets.

For message-oriented sockets, data is extracted from the first enqueued message, up to the size of the buffer specified. If the datagram or message is larger than the buffer specified, the buffer is filled with the first part of the datagram, and **recvfrom** generates the error WSAEMSGSIZE. For unreliable protocols (for example, UDP) the excess data is lost. For UDP if the packet received contains no data (empty), the return value from the **recvfrom** function function is zero.

If the *from* parameter is nonzero and the socket is not connection oriented, (type SOCK_DGRAM for example), the network address of the peer that sent the data is copied to the corresponding sockaddr structure. The value pointed to by *fromlen* is initialized to the size of this structure and is modified, on return, to indicate the actual size of the address stored in the **sockaddr** structure.

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

If the socket is connection oriented and the remote side has shut down the connection gracefully, the call to **recvfrom** will complete immediately with zero bytes received. If the connection has been reset **recvfrom** will fail with the error WSAECONNRESET.

The *flags* parameter can be used to influence the behavior of the function invocation beyond the options specified for the associated socket. The semantics of this function are determined by the socket options and the *flags* parameter. The latter is constructed by using the bitwise OR operator with any of the following values.

VALUE	MEANING
MSG_PEEK	Peeks at the incoming data. The data is copied into the buffer but is not removed from the input queue.
MSG_OOB	Processes Out Of Band (OOB) data.

> **Note** When issuing a blocking Winsock call such as **recvfrom**, Winsock may need to wait for a network event before the call can complete. Winsock performs an alertable wait in this situation, which can be interrupted by an asynchronous procedure call (APC) scheduled on the same thread. Issuing another blocking Winsock call inside an APC that interrupted an ongoing blocking Winsock call on the same thread will lead to undefined behavior, and must never be attempted by Winsock clients.

Example Code

The following example demonstrates the use of the **recvfrom** function.

```
#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 main()
{

    int iResult = 0;

    WSADATA wsaData;

    SOCKET RecvSocket;
    sockaddr_in RecvAddr;

    unsigned short Port = 27015;

    char RecvBuf[1024];
    int BufLen = 1024;

    sockaddr_in SenderAddr;
    int SenderAddrSize = sizeof (SenderAddr);

    //-----------------------------------------------
    // Initialize Winsock
    iResult = WSAStartup(MAKEWORD(2, 2), &wsaData);
    if (iResult != NO_ERROR) {
        wprintf(L"WSAStartup failed with error %d\n", iResult);
        return 1;
    }
    //-----------------------------------------------
    // Create a receiver socket to receive datagrams
    RecvSocket = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);
    if (RecvSocket == INVALID_SOCKET) {
        wprintf(L"socket failed with error %d\n", WSAGetLastError());
        return 1;
    }
    //-----------------------------------------------
    // Bind the socket to any address and the specified port.
    RecvAddr.sin_family = AF_INET;
    RecvAddr.sin_port = htons(Port);
    RecvAddr.sin_addr.s_addr = htonl(INADDR_ANY);

    iResult = bind(RecvSocket, (SOCKADDR *) & RecvAddr, sizeof (RecvAddr));
    if (iResult != 0) {
        wprintf(L"bind failed with error %d\n", WSAGetLastError());
        return 1;
    }
    //-----------------------------------------------
    // Call the recvfrom function to receive datagrams
    // on the bound socket.
    wprintf(L"Receiving datagrams...\n");
    iResult = recvfrom(RecvSocket,
                       RecvBuf, BufLen, 0, (SOCKADDR *) & SenderAddr, &SenderAddrSize);
    if (iResult == SOCKET_ERROR) {
        wprintf(L"recvfrom failed with error %d\n", WSAGetLastError());
    }
```

```
    //-----------------------------------------------
    // Close the socket when finished receiving datagrams
    wprintf(L"Finished receiving. Closing socket.\n");
    iResult = closesocket(RecvSocket);
    if (iResult == SOCKET_ERROR) {
        wprintf(L"closesocket failed with error %d\n", WSAGetLastError());
        return 1;
    }

    //-----------------------------------------------
    // Clean up and exit.
    wprintf(L"Exiting.\n");
    WSACleanup();
    return 0;
}
```

Windows Phone 8: This function is supported for Windows Phone Store apps on Windows Phone 8 and later.

Windows 8.1 and **Windows Server 2012 R2**: This function is supported for Windows Store apps on Windows 8.1, Windows Server 2012 R2, and later.

Requirements

Minimum supported client	Windows 8.1, Windows Vista [desktop apps | UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps | UWP apps]
Target Platform	Windows
Header	winsock2.h (include Winsock2.h)
Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

WSAAsyncSelect

WSAEventSelect

Winsock Functions

Winsock Reference

recv

send

sockaddr

socket

select function (winsock2.h)

10/22/2022 • 6 minutes to read • Edit Online

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

Syntax

```
int WSAAPI select(
  [in]      int           nfds,
  [in, out] fd_set        *readfds,
  [in, out] fd_set        *writefds,
  [in, out] fd_set        *exceptfds,
  [in]      const timeval *timeout
);
```

Parameters

`[in] nfds`

Ignored. The *nfds* parameter is included only for compatibility with Berkeley sockets.

`[in, out] readfds`

An optional pointer to a set of sockets to be checked for readability.

`[in, out] writefds`

An optional pointer to a set of sockets to be checked for writability.

`[in, out] exceptfds`

An optional pointer to a set of sockets to be checked for errors.

`[in] timeout`

The maximum time for **select** to wait, provided in the form of a TIMEVAL structure. Set the *timeout* parameter to **null** for blocking operations.

Return value

The **select** function returns the total number of socket handles that are ready and contained in the fd_set structures, zero if the time limit expired, or SOCKET_ERROR if an error occurred. If the return value is SOCKET_ERROR, WSAGetLastError can be used to retrieve a specific error code.

ERROR CODE	MEANING
WSANOTINITIALISED	A successful WSAStartup call must occur before using this function.

WSAEFAULT	The Windows Sockets implementation was unable to allocate needed resources for its internal operations, or the *readfds*, *writefds*, *exceptfds*, or *timeval* parameters are not part of the user address space.
WSAENETDOWN	The network subsystem has failed.
WSAEINVAL	The *time-out* value is not valid, or all three descriptor parameters were **null**.
WSAEINTR	A blocking Windows Socket 1.1 call was canceled through WSACancelBlockingCall.
WSAEINPROGRESS	A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function.
WSAENOTSOCK	One of the descriptor sets contains an entry that is not a socket.

Remarks

The **select** function is used to determine the status of one or more sockets. For each socket, the caller can request information on read, write, or error status. The set of sockets for which a given status is requested is indicated by an fd_set structure. The sockets contained within the **fd_set** structures must be associated with a single service provider. For the purpose of this restriction, sockets are considered to be from the same service provider if the WSAPROTOCOL_INFO structures describing their protocols have the same *providerId* value. Upon return, the structures are updated to reflect the subset of these sockets that meet the specified condition. The **select** function returns the number of sockets meeting the conditions. A set of macros is provided for manipulating an **fd_set** structure. These macros are compatible with those used in the Berkeley software, but the underlying representation is completely different.

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

For connection-oriented sockets, readability can also indicate that a request to close the socket has been received from the peer. If the virtual circuit was closed gracefully, and all data was received, then a recv will return immediately with zero bytes read. If the virtual circuit was reset, then a **recv** will complete immediately with an error code such as WSAECONNRESET. The presence of OOB data will be checked if the socket option SO_OOBINLINE has been enabled (see setsockopt).

The parameter *writefds* identifies the sockets that are to be checked for writability. If a socket is processing a connect call (nonblocking), a socket is writable if the connection establishment successfully completes. If the socket is not processing a **connect** call, writability means a send, sendto, or WSASendto are guaranteed to succeed. However, they can block on a blocking socket if the *len* parameter exceeds the amount of outgoing system buffer space available. It is not specified how long these guarantees can be assumed to be valid, particularly in a multithreaded environment.

The parameter *exceptfds* identifies the sockets that are to be checked for the presence of OOB data or any exceptional error conditions.

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

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

In summary, a socket will be identified in a particular set when **select** returns if:

readfds:

- If listen has been called and a connection is pending, accept will succeed.
- Data is available for reading (includes OOB data if SO_OOBINLINE is enabled).
- Connection has been closed/reset/terminated.

writefds:

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

exceptfds:

- If processing a **connect** call (nonblocking), connection attempt failed.
- OOB data is available for reading (only if SO_OOBINLINE is disabled).

Four macros are defined in the header file Winsock2.h for manipulating and checking the descriptor sets. The variable FD_SETSIZE determines the maximum number of descriptors in a set. (The default value of FD_SETSIZE is 64, which can be modified by defining FD_SETSIZE to another value before including Winsock2.h.) Internally, socket handles in an fd_set structure are not represented as bit flags as in Berkeley Unix. Their data representation is opaque. Use of these macros will maintain software portability between different socket environments. The macros to manipulate and check **fd_set** contents are:

- *FD_ZERO(*set)* - Initializes set to the empty set. A set should always be cleared before using.
- *FD_CLR(s, *set)* - Removes socket s from set.
- *FD_ISSET(s, *set)* - Checks to see if s is a member of set and returns TRUE if so.
- *FD_SET(s, *set)* - Adds socket s to set.

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

> **Note** The **select** function has no effect on the persistence of socket events registered with WSAAsyncSelect or WSAEventSelect.

> **Note** When issuing a blocking Winsock call such as **select** with the *timeout* parameter set to **NULL**, Winsock may need to wait for a network event before the call can complete. Winsock performs an alertable wait in this situation, which can be interrupted by an asynchronous procedure call (APC) scheduled on the same thread. Issuing another blocking Winsock call inside an APC that interrupted an ongoing blocking Winsock call on the same thread will lead to undefined behavior, and must never be attempted by Winsock clients.

Windows Phone 8: This function is supported for Windows Phone Store apps on Windows Phone 8 and later.

Windows 8.1 and **Windows Server 2012 R2**: This function is supported for Windows Store apps on Windows 8.1, Windows Server 2012 R2, and later.

Requirements

Minimum supported client	Windows 8.1, Windows Vista [desktop apps | UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps | UWP apps]
Target Platform	Windows
Header	winsock2.h
Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

TIMEVAL

WSAAsyncSelect

WSAEventSelect

Winsock Functions

Winsock Reference

accept

connect

recv

recvfrom

send

send function (winsock2.h)

10/22/2022 • 7 minutes to read • Edit Online

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

Syntax

```
int WSAAPI send(
  [in] SOCKET     s,
  [in] const char *buf,
  [in] int        len,
  [in] int        flags
);
```

Parameters

`[in] s`

A descriptor identifying a connected socket.

`[in] buf`

A pointer to a buffer containing the data to be transmitted.

`[in] len`

The length, in bytes, of the data in buffer pointed to by the *buf* parameter.

`[in] flags`

A set of flags that specify the way in which the call is made. This parameter is constructed by using the bitwise OR operator with any of the following values.

VALUE	MEANING
MSG_DONTROUTE	Specifies that the data should not be subject to routing. A Windows Sockets service provider can choose to ignore this flag.
MSG_OOB	Sends OOB data (stream-style socket such as SOCK_STREAM only.

Return value

If no error occurs, **send** returns the total number of bytes sent, which can be less than the number requested to be sent in the *len* parameter. Otherwise, a value of SOCKET_ERROR is returned, and a specific error code can be retrieved by calling WSAGetLastError.

ERROR CODE	MEANING
WSANOTINITIALISED	A successful WSAStartup call must occur before using this function.

WSAENETDOWN	The network subsystem has failed.
WSAEACCES	The requested address is a broadcast address, but the appropriate flag was not set. Call setsockopt with the SO_BROADCAST socket option to enable use of the broadcast address.
WSAEINTR	A blocking Windows Sockets 1.1 call was canceled through WSACancelBlockingCall.
WSAEINPROGRESS	A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function.
WSAEFAULT	The *buf* parameter is not completely contained in a valid part of the user address space.
WSAENETRESET	The connection has been broken due to the keep-alive activity detecting a failure while the operation was in progress.
WSAENOBUFS	No buffer space is available.
WSAENOTCONN	The socket is not connected.
WSAENOTSOCK	The descriptor is not a socket.
WSAEOPNOTSUPP	MSG_OOB was specified, but the socket is not stream-style such as type SOCK_STREAM, OOB data is not supported in the communication domain associated with this socket, or the socket is unidirectional and supports only receive operations.
WSAESHUTDOWN	The socket has been shut down; it is not possible to send on a socket after shutdown has been invoked with *how* set to SD_SEND or SD_BOTH.
WSAEWOULDBLOCK	The socket is marked as nonblocking and the requested operation would block.
WSAEMSGSIZE	The socket is message oriented, and the message is larger than the maximum supported by the underlying transport.
WSAEHOSTUNREACH	The remote host cannot be reached from this host at this time.
WSAEINVAL	The socket has not been bound with bind, or an unknown flag was specified, or MSG_OOB was specified for a socket with SO_OOBINLINE enabled.

WSAECONNABORTED	The virtual circuit was terminated due to a time-out or other failure. The application should close the socket as it is no longer usable.
WSAECONNRESET	The virtual circuit was reset by the remote side executing a hard or abortive close. For UDP sockets, the remote host was unable to deliver a previously sent UDP datagram and responded with a "Port Unreachable" ICMP packet. The application should close the socket as it is no longer usable.
WSAETIMEDOUT	The connection has been dropped, because of a network failure or because the system on the other end went down without notice.

Remarks

The **send** function is used to write outgoing data on a connected socket.

For message-oriented sockets (address family of **AF_INET** or **AF_INET6**, type of **SOCK_DGRAM**, and protocol of **IPPROTO_UDP**, for example), care must be taken not to exceed the maximum packet size of the underlying provider. The maximum message packet size for a provider can be obtained by calling getsockopt with the *optname* parameter set to **SO_MAX_MSG_SIZE** to retrieve the value of socket option. If the data is too long to pass atomically through the underlying protocol, the error WSAEMSGSIZE is returned, and no data is transmitted.

The successful completion of a **send** function does not indicate that the data was successfully delivered and received to the recipient. This function only indicates the data was successfully sent.

If no buffer space is available within the transport system to hold the data to be transmitted, **send** will block unless the socket has been placed in nonblocking mode. On nonblocking stream oriented sockets, the number of bytes written can be between 1 and the requested length, depending on buffer availability on both the client and server computers. The select, WSAAsyncSelect or WSAEventSelect functions can be used to determine when it is possible to send more data.

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

The *flags* parameter can be used to influence the behavior of the function beyond the options specified for the associated socket. The semantics of the **send** function are determined by any options previously set on the socket specified in the *s* parameter and the *flags* parameter passed to the **send** function.

The order of calls made to **send** is also the order in which the buffers are transmitted to the transport layer. **send** should not be called on the same stream-oriented socket concurrently from different threads, because some Winsock providers may split a large send request into multiple transmissions, and this may lead to unintended data interleaving from multiple concurrent send requests on the same stream-oriented socket.

> **Note** When issuing a blocking Winsock call such as **send**, Winsock may need to wait for a network event before the call can complete. Winsock performs an alertable wait in this situation, which can be interrupted by an asynchronous procedure call (APC) scheduled on the same thread. Issuing another blocking Winsock call inside an APC that interrupted an ongoing blocking Winsock call on the same thread will lead to undefined behavior, and must never be attempted by Winsock clients.

Example Code

The following example demonstrates the use of the **send** function.

```
#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")

#define DEFAULT_BUFLEN 512
#define DEFAULT_PORT 27015

int main() {

    //-----------------------
    // Declare and initialize variables.
    int iResult;
    WSADATA wsaData;

    SOCKET ConnectSocket = INVALID_SOCKET;
    struct sockaddr_in clientService;

    int recvbuflen = DEFAULT_BUFLEN;
    char *sendbuf = "Client: sending data test";
    char recvbuf[DEFAULT_BUFLEN] = "";

    //-----------------------
    // Initialize Winsock
    iResult = WSAStartup(MAKEWORD(2,2), &wsaData);
    if (iResult != NO_ERROR) {
        wprintf(L"WSAStartup failed with error: %d\n", iResult);
        return 1;
    }

    //-----------------------
    // Create a SOCKET for connecting to server
    ConnectSocket = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
    if (ConnectSocket == INVALID_SOCKET) {
        wprintf(L"socket failed with error: %ld\n", WSAGetLastError());
        WSACleanup();
        return 1;
    }

    //-----------------------
    // The sockaddr_in structure specifies the address family,
    // IP address, and port of the server to be connected to.
    clientService.sin_family = AF_INET;
    clientService.sin_addr.s_addr = inet_addr( "127.0.0.1" );
    clientService.sin_port = htons( DEFAULT_PORT );

    //-----------------------
    // Connect to server.
    iResult = connect( ConnectSocket, (SOCKADDR*) &clientService, sizeof(clientService) );
    if (iResult == SOCKET_ERROR) {
        wprintf(L"connect failed with error: %d\n", WSAGetLastError() );
        closesocket(ConnectSocket);
        WSACleanup();
        return 1;
  }

    //-----------------------
    // Send an initial buffer
    iResult = send( ConnectSocket, sendbuf, (int)strlen(sendbuf), 0 );
    if (iResult == SOCKET_ERROR) {
```

```
        wprintf(L"send failed with error: %d\n", WSAGetLastError());
        closesocket(ConnectSocket);
        WSACleanup();
        return 1;
    }

    printf("Bytes Sent: %d\n", iResult);

    // shutdown the connection since no more data will be sent
    iResult = shutdown(ConnectSocket, SD_SEND);
    if (iResult == SOCKET_ERROR) {
        wprintf(L"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 )
            wprintf(L"Bytes received: %d\n", iResult);
        else if ( iResult == 0 )
            wprintf(L"Connection closed\n");
        else
            wprintf(L"recv failed with error: %d\n", WSAGetLastError());

    } while( iResult > 0 );


    // close the socket
    iResult = closesocket(ConnectSocket);
    if (iResult == SOCKET_ERROR) {
        wprintf(L"close failed with error: %d\n", WSAGetLastError());
        WSACleanup();
        return 1;
    }

    WSACleanup();
    return 0;
}
```

Example Code

For a another example that uses the **send** function, see Getting Started With Winsock.

Notes for IrDA Sockets

- The Af_irda.h header file must be explicitly included.

Windows Phone 8: This function is supported for Windows Phone Store apps on Windows Phone 8 and later.

Windows 8.1 and **Windows Server 2012 R2**: This function is supported for Windows Store apps on Windows 8.1, Windows Server 2012 R2, and later.

Requirements

Minimum supported client	Windows 8.1, Windows Vista [desktop apps | UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps | UWP apps]

Target Platform	Windows
Header	winsock2.h
Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

Getting Started With Winsock

WSAAsyncSelect

WSAEventSelect

Winsock Functions

Winsock Reference

recv

recvfrom

select

sendto

socket

sendto function (winsock2.h)

10/22/2022 • 8 minutes to read • Edit Online

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

Syntax

```
int WSAAPI sendto(
  [in] SOCKET         s,
  [in] const char     *buf,
  [in] int            len,
  [in] int            flags,
  [in] const sockaddr *to,
  [in] int            tolen
);
```

Parameters

`[in] s`

A descriptor identifying a (possibly connected) socket.

`[in] buf`

A pointer to a buffer containing the data to be transmitted.

`[in] len`

The length, in bytes, of the data pointed to by the *buf* parameter.

`[in] flags`

A set of flags that specify the way in which the call is made.

`[in] to`

An optional pointer to a sockaddr structure that contains the address of the target socket.

`[in] tolen`

The size, in bytes, of the address pointed to by the *to* parameter.

Return value

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

ERROR CODE	MEANING
WSANOTINITIALISED	A successful WSAStartup call must occur before using this function.

WSAENETDOWN	The network subsystem has failed.
WSAEACCES	The requested address is a broadcast address, but the appropriate flag was not set. Call setsockopt with the SO_BROADCAST parameter to allow the use of the broadcast address.
WSAEINVAL	An unknown flag was specified, or MSG_OOB was specified for a socket with SO_OOBINLINE enabled.
WSAEINTR	A blocking Windows Sockets 1.1 call was canceled through WSACancelBlockingCall.
WSAEINPROGRESS	A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function.
WSAEFAULT	The *buf* or *to* parameters are not part of the user address space, or the *tolen* parameter is too small.
WSAENETRESET	The connection has been broken due to keep-alive activity detecting a failure while the operation was in progress.
WSAENOBUFS	No buffer space is available.
WSAENOTCONN	The socket is not connected (connection-oriented sockets only).
WSAENOTSOCK	The descriptor is not a socket.
WSAEOPNOTSUPP	MSG_OOB was specified, but the socket is not stream-style such as type SOCK_STREAM, OOB data is not supported in the communication domain associated with this socket, or the socket is unidirectional and supports only receive operations.
WSAESHUTDOWN	The socket has been shut down; it is not possible to sendto on a socket after shutdown has been invoked with *how* set to SD_SEND or SD_BOTH.
WSAEWOULDBLOCK	The socket is marked as nonblocking and the requested operation would block.
WSAEMSGSIZE	The socket is message oriented, and the message is larger than the maximum supported by the underlying transport.
WSAEHOSTUNREACH	The remote host cannot be reached from this host at this time.

WSAECONNABORTED	The virtual circuit was terminated due to a time-out or other failure. The application should close the socket as it is no longer usable.
WSAECONNRESET	The virtual circuit was reset by the remote side executing a hard or abortive close. For UPD sockets, the remote host was unable to deliver a previously sent UDP datagram and responded with a "Port Unreachable" ICMP packet. The application should close the socket as it is no longer usable.
WSAEADDRNOTAVAIL	The remote address is not a valid address, for example, ADDR_ANY.
WSAEAFNOSUPPORT	Addresses in the specified family cannot be used with this socket.
WSAEDESTADDRREQ	A destination address is required.
WSAENETUNREACH	The network cannot be reached from this host at this time.
WSAEHOSTUNREACH	A socket operation was attempted to an unreachable host.
WSAETIMEDOUT	The connection has been dropped, because of a network failure or because the system on the other end went down without notice.

Remarks

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

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

> **Note** If a socket is opened, a setsockopt call is made, and then a **sendto** call is made, Windows Sockets performs an implicit **bind** function call.

If the socket is unbound, unique values are assigned to the local association by the system, and the socket is then marked as bound. If the socket is connected, the getsockname function can be used to determine the local IP address and port associated with the socket.

If the socket is not connected, the getsockname function can be used to determine the local port number associated with the socket but the IP address returned is set to the wildcard address for the given protocol (for example, INADDR_ANY or "0.0.0.0" for IPv4 and IN6ADDR_ANY_INIT or "::" for IPv6).

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

The **sendto** function is normally used on a connectionless socket to send a datagram to a specific peer socket identified by the *to* parameter. Even if the connectionless socket has been previously connected to a specific address, the *to* parameter overrides the destination address for that particular datagram only. On a connection-oriented socket, the *to* and *tolen* parameters are ignored, making **sendto** equivalent to send.

> **Note** When issuing a blocking Winsock call such as **sendto**, Winsock may need to wait for a network event before the call can complete. Winsock performs an alertable wait in this situation, which can be interrupted by an asynchronous procedure call (APC) scheduled on the same thread. Issuing another blocking Winsock call inside an APC that interrupted an ongoing blocking Winsock call on the same thread will lead to undefined behavior, and must never be attempted by Winsock clients.

Example Code

The following example demonstrates the use of the **sendto** function.

```
#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 main()
{

    int iResult;
    WSADATA wsaData;

    SOCKET SendSocket = INVALID_SOCKET;
    sockaddr_in RecvAddr;

    unsigned short Port = 27015;

    char SendBuf[1024];
    int BufLen = 1024;

    //----------------------
    // Initialize Winsock
    iResult = WSAStartup(MAKEWORD(2, 2), &wsaData);
    if (iResult != NO_ERROR) {
        wprintf(L"WSAStartup failed with error: %d\n", iResult);
        return 1;
    }

    //-----------------------------------------------
    // Create a socket for sending data
    SendSocket = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);
    if (SendSocket == INVALID_SOCKET) {
        wprintf(L"socket failed with error: %ld\n", WSAGetLastError());
        WSACleanup();
        return 1;
    }
    //-----------------------------------------------
    // Set up the RecvAddr structure with the IP address of
    // the receiver (in this example case "192.168.1.1")
    // and the specified port number.
```

```
    RecvAddr.sin_family = AF_INET;
    RecvAddr.sin_port = htons(Port);
    RecvAddr.sin_addr.s_addr = inet_addr("192.168.1.1");

    //---------------------------------------------
    // Send a datagram to the receiver
    wprintf(L"Sending a datagram to the receiver...\n");
    iResult = sendto(SendSocket,
                     SendBuf, BufLen, 0, (SOCKADDR *) & RecvAddr, sizeof (RecvAddr));
    if (iResult == SOCKET_ERROR) {
        wprintf(L"sendto failed with error: %d\n", WSAGetLastError());
        closesocket(SendSocket);
        WSACleanup();
        return 1;
    }
    //---------------------------------------------
    // When the application is finished sending, close the socket.
    wprintf(L"Finished sending. Closing socket.\n");
    iResult = closesocket(SendSocket);
    if (iResult == SOCKET_ERROR) {
        wprintf(L"closesocket failed with error: %d\n", WSAGetLastError());
        WSACleanup();
        return 1;
    }
    //---------------------------------------------
    // Clean up and quit.
    wprintf(L"Exiting.\n");
    WSACleanup();
    return 0;
}
```

For Sockets Using IP (Version 4)

To send a broadcast (on a SOCK_DGRAM only), the address pointed to by the *to* parameter can be constructed to contain the special IPv4 address INADDR_BROADCAST (defined in Winsock2.h), together with the intended port number. If the address pointed to by the *to* parameter contains the INADDR_BROADCAST address and intended port, then the broadcast will be sent out on all interfaces to that port.

If the broadcast should be sent out only on a specific interface, then the address pointed to by the *to* parameter should contain the subnet broadcast address for the interface and the intended port. For example, an IPv4 network address of 192.168.1.0 with a subnet mask of 255.255.255.0 would use a subnet broadcast address of 192.168.1.255.

It is generally inadvisable for a broadcast datagram to exceed the size at which fragmentation can occur, which implies that the data portion of the datagram (excluding headers) should not exceed 512 bytes.

If no buffer space is available within the transport system to hold the data to be transmitted, **sendto** will block unless the socket has been placed in a nonblocking mode. On nonblocking, stream oriented sockets, the number of bytes written can be between 1 and the requested length, depending on buffer availability on both the client and server systems. The select, WSAAsyncSelect or WSAEventSelect function can be used to determine when it is possible to send more data.

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

The *flags* parameter can be used to influence the behavior of the function invocation beyond the options specified for the associated socket. The semantics of this function are determined by the socket options and the *flags* parameter. The latter is constructed by using the bitwise OR operator with any of the following values.

VALUE	MEANING

MSG_DONTROUTE	Specifies that the data should not be subject to routing. A Windows Sockets service provider can choose to ignore this flag.
MSG_OOB	Sends OOB data (stream-style socket such as SOCK_STREAM only).

Windows Phone 8: This function is supported for Windows Phone Store apps on Windows Phone 8 and later.

Windows 8.1 and **Windows Server 2012 R2**: This function is supported for Windows Store apps on Windows 8.1, Windows Server 2012 R2, and later.

Requirements

Minimum supported client	Windows 8.1, Windows Vista [desktop apps | UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps | UWP apps]
Target Platform	Windows
Header	winsock2.h (include Winsock2.h)
Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

WSAAsyncSelect

WSAEventSelect

Winsock Functions

Winsock Reference

recv

recvfrom

select

send

socket

SERVENT structure (winsock2.h)

10/22/2022 • 2 minutes to read • Edit Online

The **servent** structure is used to store or return the name and service number for a given service name.

Syntax

```
typedef struct servent {
  char  *s_name;
  char  **s_aliases;
#if ...
  char  *s_proto;
#if ...
  short s_port;
#else
  short s_port;
#endif
#else
  char  *s_proto;
#endif
} SERVENT, *PSERVENT, *LPSERVENT;
```

Members

`s_name`

The official name of the service.

`s_aliases`

A **NULL**-terminated array of alternate names.

`s_proto`

The name of the protocol to use when contacting the service.

`s_port`

The port number at which the service can be contacted. Port numbers are returned in network byte order.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	winsock2.h (include Winsock2.h)

See also

getservbyname

setsockopt function (winsock2.h)

10/22/2022 • 10 minutes to read • Edit Online

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

Syntax

```
int WSAAPI setsockopt(
  [in] SOCKET     s,
  [in] int        level,
  [in] int        optname,
  [in] const char *optval,
  [in] int        optlen
);
```

Parameters

`[in] s`

A descriptor that identifies a socket.

`[in] level`

The level at which the option is defined (for example, SOL_SOCKET).

`[in] optname`

The socket option for which the value is to be set (for example, SO_BROADCAST). The *optname* parameter must be a socket option defined within the specified *level*, or behavior is undefined.

`[in] optval`

A pointer to the buffer in which the value for the requested option is specified.

`[in] optlen`

The size, in bytes, of the buffer pointed to by the *optval* parameter.

Return value

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

ERROR CODE	MEANING
WSANOTINITIALISED	A successful WSAStartup call must occur before using this function.
WSAENETDOWN	The network subsystem has failed.

WSAEFAULT	The buffer pointed to by the *optval* parameter is not in a valid part of the process address space or the *optlen* parameter is too small.
WSAEINPROGRESS	A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function.
WSAEINVAL	The *level* parameter is not valid, or the information in the buffer pointed to by the *optval* parameter is not valid.
WSAENETRESET	The connection has timed out when SO_KEEPALIVE is set.
WSAENOPROTOOPT	The option is unknown or unsupported for the specified provider or socket (see SO_GROUP_PRIORITY limitations).
WSAENOTCONN	The connection has been reset when SO_KEEPALIVE is set.
WSAENOTSOCK	The descriptor is not a socket.

Remarks

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

> **Note** If the **setsockopt** function is called before the bind function, TCP/IP options will not be checked by using TCP/IP until the **bind** occurs. In this case, the **setsockopt** function call will always succeed, but the **bind** function call can fail because of an early **setsockopt** call failing.

> **Note** If a socket is opened, a **setsockopt** call is made, and then a sendto call is made, Windows Sockets performs an implicit bind function call.

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

The following tables list some of the common options supported by the **setsockopt** function. The Type column identifies the type of data addressed by *optval* parameter. The Description column provides some basic information about the socket option. For more complete lists of socket options and more detailed information (default values, for example), see the detailed topics under Socket Options.

level = SOL_SOCKET

VALUE	TYPE	DESCRIPTION
SO_BROADCAST	BOOL	Configures a socket for sending broadcast data.
SO_CONDITIONAL_ACCEPT	BOOL	Enables incoming connections are to be accepted or rejected by the application, not by the protocol stack.
SO_DEBUG	BOOL	Enables debug output. Microsoft providers currently do not output any debug information.
SO_DONTLINGER	BOOL	Does not block close waiting for unsent data to be sent. Setting this option is equivalent to setting SO_LINGER with **l_onoff** set to zero.
SO_DONTROUTE	BOOL	Sets whether outgoing data should be sent on interface the socket is bound to and not a routed on some other interface. This option is not supported on ATM sockets (results in an error).
SO_GROUP_PRIORITY	int	Reserved.
SO_KEEPALIVE	BOOL	Enables sending keep-alive packets for a socket connection. Not supported on ATM sockets (results in an error).
SO_LINGER	LINGER	Lingers on close if unsent data is present.
SO_OOBINLINE	BOOL	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. For a discussion of this topic, see Protocol Independent Out-Of-band Data.
SO_RCVBUF	int	Specifies the total per-socket buffer space reserved for receives.
SO_REUSEADDR	BOOL	Allows the socket to be bound to an address that is already in use. For more information, see bind. Not applicable on ATM sockets.
SO_EXCLUSIVEADDRUSE	BOOL	Enables a socket to be bound for exclusive access. Does not require administrative privilege.
SO_RCVTIMEO	DWORD	Sets the timeout, in milliseconds, for blocking receive calls.
SO_SNDBUF	int	Specifies the total per-socket buffer space reserved for sends.

SO_SNDTIMEO	DWORD	The timeout, in milliseconds, for blocking send calls.
SO_UPDATE_ACCEPT_CONTEXT	int	Updates the accepting socket with the context of the listening socket.
PVD_CONFIG	Service Provider Dependent	This object stores the configuration information for the service provider associated with socket *s*. The exact format of this data structure is service provider specific.

For more complete and detailed information about socket options for *level* = **SOL_SOCKET**, see SOL_SOCKET Socket Options.

level = **IPPROTO_TCP**

See **TCP_NODELAY** in IPPROTO_TCP socket options. Also see that topic for more complete and detailed information about socket options for *level* = **IPPROTO_TCP**.

level = **NSPROTO_IPX**

VALUE	TYPE	DESCRIPTION
IPX_PTYPE	int	Sets the IPX packet type.
IPX_FILTERPTYPE	int	Sets the receive filter packet type
IPX_STOPFILTERPTYPE	int	Stops filtering the filter type set with IPX_FILTERTYPE
IPX_DSTYPE	int	Sets the value of the data stream field in the SPX header on every packet sent.
IPX_EXTENDED_ADDRESS	BOOL	Sets whether extended addressing is enabled.
IPX_RECVHDR	BOOL	Sets whether the protocol header is sent up on all receive headers.
IPX_RECEIVE_BROADCAST	BOOL	Indicates broadcast packets are likely on the socket. Set to **TRUE** by default. Applications that do not use broadcasts should set this to **FALSE** for better system performance.
IPX_IMMEDIATESPXACK	BOOL	Directs SPX connections not to delay before sending an ACK. Applications without back-and-forth traffic should set this to **TRUE** to increase performance.

For more complete and detailed information about socket options for *level* = **NSPROTO_IPX**, see NSPROTO_IPX Socket Options.

BSD options not supported for **setsockopt** are shown in the following table.

VALUE	TYPE	DESCRIPTION
SO_ACCEPTCONN	BOOL	Returns whether a socket is in listening mode. This option is only Valid for connection-oriented protocols. This socket option is not supported for the setting.
SO_RCVLOWAT	int	A socket option from BSD UNIX included for backward compatibility. This option sets the minimum number of bytes to process for socket input operations.
SO_SNDLOWAT	int	A socket option from BSD UNIX included for backward compatibility. This option sets the minimum number of bytes to process for socket output operations.
SO_TYPE	int	Returns the socket type for the given socket (SOCK_STREAM or SOCK_DGRAM, for example This socket option is not supported for the setting the socket type.

Note When issuing a blocking Winsock call such as **setsockopt**, Winsock may need to wait for a network event before the call can complete. Winsock performs an alertable wait in this situation, which can be interrupted by an asynchronous procedure call (APC) scheduled on the same thread. Issuing another blocking Winsock call inside an APC that interrupted an ongoing blocking Winsock call on the same thread will lead to undefined behavior, and must never be attempted by Winsock clients.

Example Code

The following example demonstrates the **setsockopt** function.

```
#ifndef UNICODE
#define UNICODE
#endif

#ifndef WIN32_LEAN_AND_MEAN
#define WIN32_LEAN_AND_MEAN
#endif

#include <winsock2.h>
#include <Ws2tcpip.h>
#include <stdio.h>

// Link with ws2_32.lib
#pragma comment(lib, "Ws2_32.lib")

int main()
{

    //-----------------------------------------
    // Declare variables
```

```
    WSADATA wsaData;

    SOCKET ListenSocket;
    sockaddr_in service;

    int iResult = 0;

    BOOL bOptVal = FALSE;
    int bOptLen = sizeof (BOOL);

    int iOptVal = 0;
    int iOptLen = sizeof (int);

    //----------------------------------------
    // Initialize Winsock
    iResult = WSAStartup(MAKEWORD(2, 2), &wsaData);
    if (iResult != NO_ERROR) {
        wprintf(L"Error at WSAStartup()\n");
        return 1;
    }
    //----------------------------------------
    // Create a listening socket
    ListenSocket = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
    if (ListenSocket == INVALID_SOCKET) {
        wprintf(L"socket function failed with error: %u\n", WSAGetLastError());
        WSACleanup();
        return 1;
    }
    //----------------------------------------
    // Bind the socket to the local IP address
    // and port 27015
    hostent *thisHost;
    char *ip;
    u_short port;
    port = 27015;
    thisHost = gethostbyname("");
    ip = inet_ntoa(*(struct in_addr *) *thisHost->h_addr_list);

    service.sin_family = AF_INET;
    service.sin_addr.s_addr = inet_addr(ip);
    service.sin_port = htons(port);

    iResult = bind(ListenSocket, (SOCKADDR *) & service, sizeof (service));
    if (iResult == SOCKET_ERROR) {
        wprintf(L"bind failed with error %u\n", WSAGetLastError());
        closesocket(ListenSocket);
        WSACleanup();
        return 1;
    }
    //----------------------------------------
    // Initialize variables and call setsockopt.
    // The SO_KEEPALIVE parameter is a socket option
    // that makes the socket send keepalive messages
    // on the session. The SO_KEEPALIVE socket option
    // requires a boolean value to be passed to the
    // setsockopt function. If TRUE, the socket is
    // configured to send keepalive messages, if FALSE
    // the socket configured to NOT send keepalive messages.
    // This section of code tests the setsockopt function
    // by checking the status of SO_KEEPALIVE on the socket
    // using the getsockopt function.

    bOptVal = TRUE;

    iResult = getsockopt(ListenSocket, SOL_SOCKET, SO_KEEPALIVE, (char *) &iOptVal, &iOptLen);
    if (iResult == SOCKET_ERROR) {
        wprintf(L"getsockopt for SO_KEEPALIVE failed with error: %u\n", WSAGetLastError());
    } else
        wprintf(L"SO_KEEPALIVE Value: %ld\n", iOptVal);
```

```
    iResult = setsockopt(ListenSocket, SOL_SOCKET, SO_KEEPALIVE, (char *) &bOptVal, bOptLen);
    if (iResult == SOCKET_ERROR) {
        wprintf(L"setsockopt for SO_KEEPALIVE failed with error: %u\n", WSAGetLastError());
    } else
        wprintf(L"Set SO_KEEPALIVE: ON\n");

    iResult = getsockopt(ListenSocket, SOL_SOCKET, SO_KEEPALIVE, (char *) &iOptVal, &iOptLen);
    if (iResult == SOCKET_ERROR) {
        wprintf(L"getsockopt for SO_KEEPALIVE failed with error: %u\n", WSAGetLastError());
    } else
        wprintf(L"SO_KEEPALIVE Value: %ld\n", iOptVal);

    closesocket(ListenSocket);
    WSACleanup();
    return 0;
}
```

Notes for IrDA Sockets

When developing applications using Windows sockets for IrDA, note the following:

- The Af_irda.h header file must be explicitly included.
- IrDA provides the following socket option:

VALUE	TYPE	MEANING
IRLMP_IAS_SET	*IAS_SET	Sets IAS attributes

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

IrDA provides an IAS database that stores IrDA-based information. Limited access to the IAS database is available through the Windows Sockets 2 interface, but such access is not normally used by applications, and exists primarily to support connections to non-Windows devices that are not compliant with the Windows Sockets 2 IrDA conventions.

The following structure, **IAS_SET**, is used with the IRLMP_IAS_SET setsockopt option to manage the local IAS database:

```
// #include <Af_irda.h> for this struct

typedef struct _IAS_SET {
    u_char      irdaClassName[IAS_MAX_CLASSNAME];
    char      irdaAttribName[IAS_MAX_ATTRIBNAME];
    u_long    irdaAttribType;
    union
    {
              LONG irdaAttribInt;
              struct
              {
                   u_long   Len;
                   u_char    OctetSeq[IAS_MAX_OCTET_STRING];
              } irdaAttribOctetSeq;
              struct
              {
                   u_long    Len;
                   u_long    CharSet;
                   u_char    UsrStr[IAS_MAX_USER_STRING];
              } irdaAttribUsrStr;
    } irdaAttribute;
} IAS_SET, *PIAS_SET, FAR *LPIASSET;
```

The following structure, **IAS_QUERY**, is used with the IRLMP_IAS_QUERY setsockopt option to query a peer's IAS database:

```
// #include <Af_irda.h> for this struct

typedef struct _WINDOWS_IAS_QUERY {
        u_char   irdaDeviceID[4];
        char     irdaClassName[IAS_MAX_CLASSNAME];
        char     irdaAttribName[IAS_MAX_ATTRIBNAME];
        u_long   irdaAttribType;
        union
        {
                  LONG    irdaAttribInt;
                  struct
                  {
                          u_long  Len;
                          u_char  OctetSeq[IAS_MAX_OCTET_STRING];
                  } irdaAttribOctetSeq;
                  struct
                  {
                          u_long  Len;
                          u_long  CharSet;
                          u_char  UsrStr[IAS_MAX_USER_STRING];
                  } irdaAttribUsrStr;
        } irdaAttribute;
} IAS_QUERY, *PIAS_QUERY, FAR *LPIASQUERY;
```

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

Windows Phone 8: This function is supported for Windows Phone Store apps on Windows Phone 8 and later.

Windows 8.1 and **Windows Server 2012 R2**: This function is supported for Windows Store apps on Windows 8.1, Windows Server 2012 R2, and later.

Requirements

Minimum supported client	Windows 8.1, Windows Vista [desktop apps | UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps | UWP apps]
Target Platform	Windows
Header	winsock2.h (include Winsock2.h)
Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

IPPROTO_IP Socket Options

IPPROTO_IPV6 Socket Options

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

Socket Options

WSAAsyncSelect

WSAEventSelect

WSAIoctl

Winsock Functions

bind

getsockopt

ioctlsocket

socket

shutdown function (winsock2.h)

10/22/2022 • 5 minutes to read • Edit Online

The **shutdown** function disables sends or receives on a socket.

Syntax

```
int WSAAPI shutdown(
  [in] SOCKET s,
  [in] int    how
);
```

Parameters

`[in] s`

A descriptor identifying a socket.

`[in] how`

A flag that describes what types of operation will no longer be allowed. Possible values for this flag are listed in the *Winsock2.h* header file.

VALUE	MEANING
SD_RECEIVE 0	Shutdown receive operations.
SD_SEND 1	Shutdown send operations.
SD_BOTH 2	Shutdown both send and receive operations.

Return value

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

ERROR CODE	MEANING
WSAECONNABORTED	The virtual circuit was terminated due to a time-out or other failure. The application should close the socket as it is no longer usable. This error applies only to a connection-oriented socket.

WSAECONNRESET	The virtual circuit was reset by the remote side executing a hard or abortive close. The application should close the socket as it is no longer usable. This error applies only to a connection-oriented socket.
WSAEINPROGRESS	A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function.
WSAEINVAL	The *how* parameter is not valid, or is not consistent with the socket type. For example, SD_SEND is used with a UNI_RECV socket type.
WSAENETDOWN	The network subsystem has failed.
WSAENOTCONN	The socket is not connected. This error applies only to a connection-oriented socket.
WSAENOTSOCK	**Note** The descriptor is not a socket.
WSANOTINITIALISED	A successful WSAStartup call must occur before using this function.

Remarks

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

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

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

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

The **shutdown** function does not close the socket. Any resources attached to the socket will not be freed until closesocket is invoked.

To assure that all data is sent and received on a connected socket before it is closed, an application should use **shutdown** to close connection before calling closesocket. One method to wait for notification that the remote end has sent all its data and initiated a graceful disconnect uses the WSAEventSelect function as follows :

1. Call WSAEventSelect to register for FD_CLOSE notification.
2. Call **shutdown** with *how*=SD_SEND.
3. When FD_CLOSE received, call the recv or WSARecv until the function completes with success and indicates that zero bytes were received. If SOCKET_ERROR is returned, then the graceful disconnect is not possible.
4. Call closesocket.

Another method to wait for notification that the remote end has sent all its data and initiated a graceful disconnect uses overlapped receive calls follows :

1. Call **shutdown** with *how*=SD_SEND.
2. Call recv or WSARecv until the function completes with success and indicates zero bytes were received. If SOCKET_ERROR is returned, then the graceful disconnect is not possible.
3. Call closesocket.

> **Note** The **shutdown** function does not block regardless of the SO_LINGER setting on the socket.

For more information, see the section on Graceful Shutdown, Linger Options, and Socket Closure.

Once the **shutdown** function is called to disable send, receive, or both, there is no method to re-enable send or receive for the existing socket connection.

An application should not rely on being able to reuse a socket after it has been shut down. In particular, a Windows Sockets provider is not required to support the use of connect on a socket that has been shut down.

If an application wants to reuse a socket, then the DisconnectEx function should be called with the *dwFlags* parameter set to **TF_REUSE_SOCKET** to close a connection on a socket and prepare the socket handle to be reused. When the **DisconnectEx** request completes, the socket handle can be passed to the AcceptEx or ConnectEx function.

If an application wants to reuse a socket, the TransmitFile or TransmitPackets functions can be called with the *dwFlags* parameter set with **TF_DISCONNECT** and **TF_REUSE_SOCKET** to disconnect after all the data has been queued for transmission and prepare the socket handle to be reused. When the **TransmitFile** request completes, the socket handle can be passed to the function call previously used to establish the connection, such as AcceptEx or ConnectEx. When the **TransmitPackets** function completes, the socket handle can be passed to the **AcceptEx** function.

> **Note** The socket level disconnect is subject to the behavior of the underlying transport. For example, a TCP socket may be subject to the TCP TIME_WAIT state, causing the DisconnectEx, TransmitFile, or TransmitPackets call to be delayed.

> **Note** When issuing a blocking Winsock call such as **shutdown**, Winsock may need to wait for a network event before the call can complete. Winsock performs an alertable wait in this situation, which can be interrupted by an asynchronous procedure call (APC) scheduled on the same thread. Issuing another blocking Winsock call inside an APC that interrupted an ongoing blocking Winsock call on the same thread will lead to undefined behavior, and must never be attempted by Winsock clients.

Notes for ATM

There are important issues associated with connection teardown when using Asynchronous Transfer Mode (ATM) and Windows Sockets 2. For more information about these important considerations, see the section titled Notes for ATM in the Remarks section of the closesocket function reference.

Windows Phone 8: This function is supported for Windows Phone Store apps on Windows Phone 8 and later.

Windows 8.1 and **Windows Server 2012 R2**: This function is supported for Windows Store apps on Windows 8.1, Windows Server 2012 R2, and later.

Requirements

Minimum supported client	Windows 8.1, Windows Vista [desktop apps | UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps | UWP apps]
Target Platform	Windows
Header	winsock2.h (include Winsock2.h, Webhost.h)
Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

AcceptEx

ConnectEx

DisconnectEx

TransmitFile

TransmitPackets

WSAEventSelect

Winsock Functions

Winsock Reference

connect

socket

SOCK_NOTIFY_REGISTRATION structure (winsock2.h)

10/22/2022 • 2 minutes to read • Edit Online

Represents info supplied to the ProcessSocketNotifications function.

For more info, and code examples, see Winsock socket state notifications.

Syntax

```
typedef struct SOCK_NOTIFY_REGISTRATION {
  SOCKET socket;
  PVOID  completionKey;
  UINT16 eventFilter;
  UINT8  operation;
  UINT8  triggerFlags;
  DWORD  registrationResult;
} SOCK_NOTIFY_REGISTRATION;
```

Members

`socket`

Type: **SOCKET**

A handle to a Winsock socket opened by any of the WSASocket, socket, WSAAccept, accept, or WSADuplicateSocket functions. Only Microsoft Winsock provider sockets are supported.

`completionKey`

Type: **PVOID**

The value to use in the *dwCompletionKey* parameter of the PostQueuedCompletionStatus function when notifications are sent on behalf of the socket. This parameter is used upon registration creation. To change the completion key, remove the registration and re-register it.

`eventFilter`

Type: **UINT16**

A set of flags indicating the notifications being requested. This must be one or more of the following values (defined in `WinSock2.h`).

SOCK_NOTIFY_REGISTER_EVENT_NONE. Notifications should not be issued. **SOCK_NOTIFY_REGISTER_EVENT_IN**. A notification should be issued when data can be read without blocking. **SOCK_NOTIFY_REGISTER_EVENT_OUT**. A notification should be issued when data can be written without blocking. **SOCK_NOTIFY_REGISTER_EVENT_HANGUP**. A notification should be issued when a stream-oriented connection was either disconnected or aborted. **SOCK_NOTIFY_REGISTER_EVENTS_ALL**. Has the value `(SOCK_NOTIFY_REGISTER_EVENT_IN | SOCK_NOTIFY_REGISTER_EVENT_OUT | SOCK_NOTIFY_REGISTER_EVENT_HANGUP)`.

`operation`

Type: UINT8

Indicates the operation to perform on a registration. At most one operation may be performed at a time. These values are defined in `WinSock2.h`.

SOCK_NOTIFY_OP_NONE. No registration operations should take place. Use this if your application calls ProcessSocketNotifications and is only interested in receiving notifications. **SOCK_NOTIFY_OP_ENABLE**. Enables the registration. Notifications must not be re-enabled until the **SOCK_NOTIFY_EVENT_DISABLE** notification is received. **SOCK_NOTIFY_OP_DISABLE**. Disables the registration, but doesn't destroy the underlying data structures. Note that this doesn't remove the registration, it merely suppresses queuing of new notifications. Notifications that have already been queued might still be delivered until the **SOCK_NOTIFY_EVENT_DISABLE** event is received. **SOCK_NOTIFY_OP_REMOVE**. Removes a previously-registered notification. Both enabled and disabled notifications may be removed. The **SOCK_NOTIFY_EVENT_REMOVE** notification is issued, with the guarantee that no more notifications will be issued afterwards for that completion key unless it is re-registered.

`triggerFlags`

Type: UINT8

A set of flags indicating the trigger behavior (defined in `WinSock2.h`).

SOCK_NOTIFY_TRIGGER_ONESHOT. The registration will be disabled (not removed) upon delivery of the next notification. **SOCK_NOTIFY_TRIGGER_PERSISTENT**. The registration will remain active until it is explicitly disabled or removed. **SOCK_NOTIFY_TRIGGER_LEVEL**. The registration is for level-triggered notifications. Not compatible with edge-triggered. One of edge- or level-triggered must be supplied. **SOCK_NOTIFY_TRIGGER_EDGE**. The registration is for edge-triggered notifications. Not compatible with level-triggered. One of edge- or level-triggered must be supplied.

Notifications are only supplied when the registration is enabled. Notifications are not queued up while the registration is disabled. As notifications are queued up for a given socket, they are coalesced into a single notification. Therefore, multiple events may be described by a single event mask for the socket.

Given the registration is enabled, level-triggered notifications are supplied whenever the desired conditions hold.

Given the registration is enabled, edge-triggered notifications are supplied whenever a condition changes from not holding to holding. The condition must change while the registration is enabled for a notification to be queued. As such, after registering, the socket's receive buffer must be completely drained to ensure a notification is received.

`registrationResult`

Type: DWORD

After a successful call to ProcessSocketNotifications, *registrationResult* contains a code indicating the success or failure of the registration. A value of **ERROR_SUCCESS** indicates that registration was successful.

Requirements

Minimum supported client	Windows 10 Build 20348
Minimum supported server	Windows 10 Build 20348
Header	winsock2.h

See also

- accept
- PostQueuedCompletionStatus
- ProcessSocketNotifications
- socket
- Winsock
- Winsock socket state notifications
- WSAAccept
- WSADuplicateSocket
- WSASocket

socket function (winsock2.h)

10/22/2022 • 13 minutes to read • Edit Online

The **socket** function creates a socket that is bound to a specific transport service provider.

Syntax

```
SOCKET WSAAPI socket(
  [in] int af,
  [in] int type,
  [in] int protocol
);
```

Parameters

`[in] af`

The address family specification. Possible values for the address family are defined in the *Winsock2.h* header file.

On the Windows SDK released for Windows Vista and later, the organization of header files has changed and the 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. Note that the values for the AF_ address family and PF_ protocol family constants are identical (for example, **AF_INET** and **PF_INET**), so either constant can be used.

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 including Windows 7, Windows Server 2008, Windows Vista, Windows Server 2003, or Windows XP. The Windows Sockets provider for NetBIOS only supports sockets where the *type* parameter is set to **SOCK_DGRAM**. The Windows Sockets provider for NetBIOS is not directly related to the NetBIOS 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 supported on Windows XP with SP2 or later if the computer has a Bluetooth adapter and driver installed.

`[in] type`

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 IP_HDRINCL socket option must be set on the socket. To manipulate the IPv6 header, the IPV6_HDRINCL 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 reliable multicast programming. This *type* 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.

In Windows Sockets 2, new socket types were introduced. An application can dynamically discover the attributes of each available transport protocol through the WSAEnumProtocols function. So an application can determine the possible socket type and protocol options for an address family and use this information when specifying this parameter. Socket type definitions in the *Winsock2.h* and *Ws2def.h* header files will be periodically updated as new socket types, address families, and protocols are defined.

In Windows Sockets 1.1, the only possible socket types are **SOCK_DGRAM** and **SOCK_STREAM**.

`[in] 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 *Winsock2.h* and *Wsrm.h* header files.

On the Windows SDK released for Windows Vista and later, the organization of header files has changed and this parameter can be one of the values from 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.

When the *af* parameter is AF_INET or AF_INET6 and the *type* is **SOCK_RAW**, the value specified for the *protocol* is set in the protocol field of the IPv6 or IPv4 packet header.

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 *af* parameter is **AF_UNSPEC**, **AF_INET**, or **AF_INET6** and the *type* parameter is **SOCK_RAW** or unspecified. This *protocol* value is supported on Windows XP and later.
IPPROTO_IGMP 2	The Internet Group Management Protocol (IGMP). This is a possible value when the *af* parameter is **AF_UNSPEC**, **AF_INET**, or **AF_INET6** and the *type* parameter is **SOCK_RAW** or unspecified. This *protocol* value is supported on Windows XP and later.
BTHPROTO_RFCOMM 3	The Bluetooth Radio Frequency Communications (Bluetooth RFCOMM) protocol. This is a possible value when the *af* parameter is **AF_BTH** and the *type* parameter is **SOCK_STREAM**. This *protocol* value is supported on Windows XP with SP2 or later.
IPPROTO_TCP 6	The Transmission Control Protocol (TCP). This is a possible value when the *af* parameter is **AF_INET** or **AF_INET6** and the *type* parameter is **SOCK_STREAM**.
IPPROTO_UDP 17	The User Datagram Protocol (UDP). This is a possible value when the *af* parameter is **AF_INET** or **AF_INET6** and the *type* parameter is **SOCK_DGRAM**.
IPPROTO_ICMPV6 58	The Internet Control Message Protocol Version 6 (ICMPv6). This is a possible value when the *af* parameter is **AF_UNSPEC**, **AF_INET**, or **AF_INET6** and the *type* parameter is **SOCK_RAW** or unspecified. This *protocol* value is supported on Windows XP and later.
IPPROTO_RM 113	The PGM protocol for reliable multicast. This is a possible value when the *af* parameter is **AF_INET** and the *type* parameter is **SOCK_RDM**. On the Windows SDK released for Windows Vista and later, this protocol is also called **IPPROTO_PGM**. This *protocol* value is only supported if the Reliable Multicast Protocol is installed.

Return value

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

ERROR CODE	MEANING
WSANOTINITIALISED	A successful WSAStartup call must occur before using this function.

WSAENETDOWN	The network subsystem or the associated service provider has failed.
WSAEAFNOSUPPORT	The specified address family is not supported. For example, an application tried to create a socket for the **AF_IRDA** address family but an infrared adapter and device driver is not installed on the local computer.
WSAEINPROGRESS	A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function.
WSAEMFILE	No more socket descriptors are available.
WSAEINVAL	An invalid argument was supplied. This error is returned if the *af* parameter is set to **AF_UNSPEC** and the *type* and *protocol* parameter are unspecified.
WSAEINVALIDPROVIDER	The service provider returned a version other than 2.2.
WSAEINVALIDPROCTABLE	The service provider returned an invalid or incomplete procedure table to the WSPStartup.
WSAENOBUFS	No buffer space is available. The socket cannot be created.
WSAEPROTONOSUPPORT	The specified protocol is not supported.
WSAEPROTOTYPE	The specified protocol is the wrong type for this socket.
WSAEPROVIDERFAILEDINIT	The service provider failed to initialize. This error is returned if a layered service provider (LSP) or namespace provider was improperly installed or the provider fails to operate correctly.
WSAESOCKTNOSUPPORT	The specified socket type is not supported in this address family.

Remarks

The **socket** function causes a socket descriptor and any related resources to be allocated and bound to a specific transport-service provider. Winsock will utilize the first available service provider that supports the requested combination of address family, socket type and protocol parameters. The socket that is created will have the overlapped attribute as a default. For Windows, the Microsoft-specific socket option, SO_OPENTYPE, defined in Mswsock.h can affect this default. See Microsoft-specific documentation for a detailed description of SO_OPENTYPE.

Sockets without the overlapped attribute can be created by using WSASocket. All functions that allow overlapped operation (WSASend, WSARecv, WSASendTo, WSARecvFrom, and WSAIoctl) also support nonoverlapped usage on an overlapped socket if the values for parameters related to overlapped operation are

NULL.

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

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

Applications are encouraged to use **AF_INET6** for the *af* parameter and create a dual-mode socket that can be used with both IPv4 and IPv6.

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

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

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

IPv6 and IPv4 operate differently when receiving a socket with a *type* of **SOCK_RAW**. The IPv4 receive packet includes the packet payload, the next upper-level header (for example, the IP header for a TCP or UDP packet), and the IPv4 packet header. The IPv6 receive packet includes the packet payload and the next upper-level header. The IPv6 receive packet never includes the IPv6 packet header.

> **Note** On Windows NT, raw socket support requires administrative privileges.

A socket with a *type* parameter of **SOCK_SEQPACKET** is based on datagrams, but functions as a pseudo-stream protocol. For both send and receive packets, separate datagrams are used. However, Windows Sockets can coalesce multiple receive packets into a single packet. So an application can issue a receive call (for example, recv or WSARecvEx) and retrieve the data from several coalesced multiple packets in single call. The AF_NETBIOS address family supports a *type* parameter of **SOCK_SEQPACKET**.

When the *af* parameter is **AF_NETBIOS** for NetBIOS over TCP/IP, the *type* parameter can be **SOCK_DGRAM** or **SOCK_SEQPACKET**. For the **AF_NETBIOS** address family, the *protocol* parameter is the LAN adapter number represented as a negative number.

On Windows XP and later, the following command can be used to list the Windows Sockets catalog to determine the service providers installed and the address family, socket type, and protocols that are supported.

```
netsh winsock show catalog
```

Support for sockets with type **SOCK_RAW** is not required, but service providers are encouraged to support raw sockets as practicable.

Notes for IrDA Sockets

Keep the following in mind:

- The Af_irda.h header file must be explicitly included.
- Only **SOCK_STREAM** is supported; the **SOCK_DGRAM** type is not supported by IrDA.
- The *protocol* parameter is always set to 0 for IrDA.

A socket for use with the AF_IRDA address family can only be created if the local computer has an infrared port and driver installed. Otherwise, a call to the **socket** function with *af* parameter set to AF_IRDA will fail and WSAGetLastError returns WSAEPROTONOSUPPORT.

Example Code

The following example demonstrates the use of the **socket** function to create a socket that is bound to a specific transport service provider..

```
#ifndef UNICODE
#define UNICODE 1
#endif

// link with Ws2_32.lib
#pragma comment(lib,"Ws2_32.lib")

#include <winsock2.h>
#include <ws2tcpip.h>
#include <stdio.h>
#include <stdlib.h>   // Needed for _wtoi


int __cdecl wmain(int argc, wchar_t **argv)
{

    //-----------------------------------------
    // Declare and initialize variables
    WSADATA wsaData = {0};
    int iResult = 0;

//    int i = 1;

    SOCKET sock = INVALID_SOCKET;
    int iFamily = AF_UNSPEC;
    int iType = 0;
    int iProtocol = 0;

    // Validate the parameters
    if (argc != 4) {
        wprintf(L"usage: %s <addressfamily> <type> <protocol>\n", argv[0]);
        wprintf(L"socket opens a socket for the specified family, type, & protocol\n");
        wprintf(L"%ws example usage\n", argv[0]);
        wprintf(L"   %ws 0 2 17\n", argv[0]);
        wprintf(L"   where AF_UNSPEC=0 SOCK_DGRAM=2 IPPROTO_UDP=17\n", argv[0]);
        return 1;
    }

    iFamily = _wtoi(argv[1]);
    iType = _wtoi(argv[2]);
    iProtocol = _wtoi(argv[3]);

    // Initialize Winsock
    iResult = WSAStartup(MAKEWORD(2, 2), &wsaData);
    if (iResult != 0) {
        wprintf(L"WSAStartup failed: %d\n", iResult);
        return 1;
    }

    wprintf(L"Calling socket with following parameters:\n");
    wprintf(L"  Address Family = ");
    switch (iFamily) {
    case AF_UNSPEC:
```

```
        wprintf(L"Unspecified");
        break;
    case AF_INET:
        wprintf(L"AF_INET (IPv4)");
        break;
    case AF_INET6:
        wprintf(L"AF_INET6 (IPv6)");
        break;
    case AF_NETBIOS:
        wprintf(L"AF_NETBIOS (NetBIOS)");
        break;
    case AF_BTH:
        wprintf(L"AF_BTH (Bluetooth)");
        break;
    default:
        wprintf(L"Other");
        break;
    }
    wprintf(L" (%d)\n", iFamily);

    wprintf(L"  Socket type = ");
    switch (iType) {
    case 0:
        wprintf(L"Unspecified");
        break;
    case SOCK_STREAM:
        wprintf(L"SOCK_STREAM (stream)");
        break;
    case SOCK_DGRAM:
        wprintf(L"SOCK_DGRAM (datagram)");
        break;
    case SOCK_RAW:
        wprintf(L"SOCK_RAW (raw)");
        break;
    case SOCK_RDM:
        wprintf(L"SOCK_RDM (reliable message datagram)");
        break;
    case SOCK_SEQPACKET:
        wprintf(L"SOCK_SEQPACKET (pseudo-stream packet)");
        break;
    default:
        wprintf(L"Other");
        break;
    }
    wprintf(L" (%d)\n", iType);

    wprintf(L"  Protocol = %d = ", iProtocol);
    switch (iProtocol) {
    case 0:
        wprintf(L"Unspecified");
        break;
    case IPPROTO_ICMP:
        wprintf(L"IPPROTO_ICMP (ICMP)");
        break;
    case IPPROTO_IGMP:
        wprintf(L"IPPROTO_IGMP (IGMP)");
        break;
    case IPPROTO_TCP:
        wprintf(L"IPPROTO_TCP (TCP)");
        break;
    case IPPROTO_UDP:
        wprintf(L"IPPROTO_UDP (UDP)");
        break;
    case IPPROTO_ICMPV6:
        wprintf(L"IPPROTO_ICMPV6 (ICMP Version 6)");
        break;
    default:
        wprintf(L"Other");
        break;
```

```
    }
    wprintf(L" (%d)\n", iProtocol);

    sock = socket(iFamily, iType, iProtocol);
    if (sock == INVALID_SOCKET)
        wprintf(L"socket function failed with error = %d\n", WSAGetLastError() );
    else {
        wprintf(L"socket function succeeded\n");

        // Close the socket to release the resources associated
        // Normally an application calls shutdown() before closesocket
        //   to  disables sends or receives on a socket first
        // This isn't needed in this simple sample
        iResult = closesocket(sock);
        if (iResult == SOCKET_ERROR) {
            wprintf(L"closesocket failed with error = %d\n", WSAGetLastError() );
            WSACleanup();
            return 1;
        }
    }

    WSACleanup();

    return 0;
}
```

Windows Phone 8: This function is supported for Windows Phone Store apps on Windows Phone 8 and later.

Windows 8.1 and **Windows Server 2012 R2**: This function is supported for Windows Store apps on Windows 8.1, Windows Server 2012 R2, and later.

Requirements

Minimum supported client	Windows 8.1, Windows Vista [desktop apps | UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps | UWP apps]
Target Platform	Windows
Header	winsock2.h
Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

IPPROTO_IP Socket Options

IPPROTO_IPV6 Socket Options

Reliable Multicast Programming

WSASocket

Winsock Functions

Winsock Reference

accept

bind

closesocket

connect

getsockname

getsockopt

ioctlsocket

listen

recv

recvfrom

select

send

sendto

setsockopt

shutdown

SocketNotificationRetrieveEvents function (winsock2.h)

10/22/2022 • 2 minutes to read • Edit Online

This inline helper function is provided as a convenience to retrieve the events mask from an OVERLAPPED_ENTRY.

For more info, and code examples, see Winsock socket state notifications.

Syntax

```
UINT32 SocketNotificationRetrieveEvents(
  OVERLAPPED_ENTRY *notification
);
```

Parameters

`notification`

Type: _In_ **OVERLAPPED_ENTRY***

A pointer to an OVERLAPPED_ENTRY received for a socket state notification.

Return value

A UINT32 containing a bitmask of the notification events for the socket.

This table lists the socket notification events. These are the events possible when a notification is received.

EVENT	DESCRIPTION
SOCK_NOTIFY_EVENT_IN	Input is available from the socket without blocking.
SOCK_NOTIFY_EVENT_OUT	Output can be provided to the socket without blocking.
SOCK_NOTIFY_EVENT_HANGUP	The socket connection has terminated.
SOCK_NOTIFY_EVENT_ERR	The socket is in an error state.
SOCK_NOTIFY_EVENT_REMOVE	The notification has been deregistered.

Remarks

The **SOCK_NOTIFY_EVENT_ERR** and **SOCK_NOTIFY_EVENT_REMOVE** events might be indicated regardless of registered event filter.

If a **SOCK_NOTIFY_EVENT_REMOVE** event is indicated, then no more notifications will be provided.

Requirements

Minimum supported client	Windows 10 Build 20348
Minimum supported server	Windows 10 Build 20348
Header	winsock2.h
Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

- OVERLAPPED_ENTRY
- ProcessSocketNotifications
- Winsock socket state notifications

TIMEVAL structure (winsock2.h)

10/22/2022 • 2 minutes to read • Edit Online

The **timeval** structure is used to specify a time interval. It is associated with the Berkeley Software Distribution (BSD) *Time.h* header file.

Syntax

```
typedef struct timeval {
  long tv_sec;
  long tv_usec;
} TIMEVAL, *PTIMEVAL, *LPTIMEVAL;
```

Members

`tv_sec`

Time interval, in seconds.

`tv_usec`

Time interval, in microseconds. This value is used in combination with the **tv_sec** member to represent time interval values that are not a multiple of seconds.

Remarks

The **timeval** structure is used in Windows Sockets by the select function to specify the maximum time the function can take to complete. The time interval is a combination of the values in **tv_sec** and **tv_usec** members.

Several functions are added on Windows Vista and later that use the **timeval** structure. These functions include GetAddrInfoEx, SetAddrInfoEx, WSAConnectByList, and WSAConnectByName.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	winsock2.h (include Winsock2.h)

See also

GetAddrInfoEx

SetAddrInfoEx

WSAConnectByList

WSAConnectByName

linger

select

WSAAccept function (winsock2.h)

10/22/2022 • 13 minutes to read • Edit Online

The **WSAAccept** function 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.

Syntax

```
SOCKET WSAAPI WSAAccept(
  [in]      SOCKET          s,
  [out]     sockaddr        *addr,
  [in, out] LPINT           addrlen,
  [in]      LPCONDITIONPROC lpfnCondition,
  [in]      DWORD_PTR       dwCallbackData
);
```

Parameters

`[in] s`

A descriptor that identifies a socket that is listening for connections after a call to the listen function.

`[out] addr`

An optional pointer to an sockaddr structure that receives the address of the connecting entity, as known to the communications layer. The exact format of the *addr* parameter is determined by the address family established when the socket was created.

`[in, out] addrlen`

An optional pointer to an integer that contains the length of the sockaddr structure pointed to by the *addr* parameter, in bytes.

`[in] lpfnCondition`

The address of an optional, application-specified condition function that will make an accept/reject decision based on the caller information passed in as parameters, and optionally create or join a socket group by assigning an appropriate value to the result parameter *g* of this function. If this parameter is **NULL**, then no condition function is called.

`[in] dwCallbackData`

Callback data passed back to the application-specified condition function as the value of the *dwCallbackData* parameter passed to the condition function. This parameter is only applicable if the *lpfnCondition* parameter is not **NULL**. This parameter is not interpreted by Windows Sockets.

Return value

If no error occurs, **WSAAccept** returns a value of type SOCKET that is a descriptor for the accepted socket. Otherwise, a value of INVALID_SOCKET is returned, and a specific error code can be retrieved by calling WSAGetLastError.

The integer referred to by *addrlen* initially contains the amount of space pointed to by *addr*. On return it will

contain the actual length in bytes of the address returned.

ERROR CODE	MEANING
WSAEACCES	An attempt was made to access a socket in a way forbidden by its access permissions. This error is returned if the connection request that was offered has timed out or been withdrawn.
WSAECONNREFUSED	No connection could be made because the target machine actively refused it. This error is returned if the connection request was forcefully rejected as indicated in the return value of the condition function (CF_REJECT).
WSAECONNRESET	An existing connection was forcibly closed by the remote host. This error is returned of an incoming connection was indicated, but was subsequently terminated by the remote peer prior to accepting the call.
WSAEFAULT	The system detected an invalid pointer address in attempting to use a pointer argument in a call. This error is returned of the *addrlen* parameter is too small or the *addr* or *lpfnCondition* is not part of the user address space.
WSAEINTR	A blocking operation was interrupted by a call to WSACancelBlockingCall. This error is returned if a blocking Windows Sockets 1.1 call was canceled through **WSACancelBlockingCall**.
WSAEINPROGRESS	A blocking operation is currently executing. This error is returned if a blocking Windows Sockets 1.1 call is in progress.
WSAEINVAL	An invalid argument was supplied. This error is returned if listen was not invoked prior to WSAAccept, the return value of the condition function is not a valid one, or any case where the specified socket is in an invalid state.
WSAEMFILE	Too many open sockets. This error is returned if the queue is nonempty upon entry to WSAAccept and there are no socket descriptors available.
WSAENETDOWN	A socket operation encountered a dead network. This error is returned if the network subsystem has failed.
WSAENOBUFS	An operation on a socket could not be performed because the system lacked sufficient buffer space or because a queue was full. This error is returned if no buffer space is available.
WSAENOTSOCK	An operation was attempted on something that is not a socket. This error is returned if the socket descriptor passed in the *s* parameter is not a socket.
WSAEOPNOTSUPP	The protocol family has not been configured into the system or no implementation for it exists. This error is returned if the referenced socket is not a type that supports connection-oriented service.

WSAEWOULDBLOCK	A non-blocking socket operation could not be completed immediately. This error is returned if the socket is marked as nonblocking and no connections are present to be accepted.
WSANOTINITIALISED	Either the application has not called WSAStartup, or **WSAStartup** failed. This error is returned of a successful call to the **WSAStartup** function dit not occur before using this function.
WSATRY_AGAIN	This is usually a temporary error during hostname resolution and means that the local server did not receive a response from an authoritative server. This error is returned if the acceptance of the connection request was deferred as indicated in the return value of the condition function (CF_DEFER).

Remarks

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

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

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

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

A prototype of the condition function is defined in the `Winsock2.h` header file as the **LPCONDITIONPROC**, as follows.

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

The **ConditionFunc** is a placeholder for the application-specified callback function. The actual condition function must reside in a DLL or application module. It is exported in the module definition file.

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

The *lpSQOS* parameter references the FLOWSPEC structures for socket *s* specified by the caller, one for each direction, followed by any additional provider-specific parameters. The sending or receiving flow specification values will be ignored as appropriate for any unidirectional sockets. A **NULL** value indicates that there is no caller-supplied quality of service and that no negotiation is possible. A non-**NULL** *lpSQOS* pointer indicates that a quality of service negotiation is to occur or that the provider is prepared to accept the quality of service request without negotiation.

The *lpGQOS* parameter is reserved, and should be **NULL**. (reserved for future use with socket groups) references the FLOWSPEC structure for the socket group the caller is to create, one for each direction, followed by any additional provider-specific parameters. A **NULL** value for *lpGQOS* indicates no caller-specified group quality of service. Quality of service information can be returned if negotiation is to occur.

The *lpCalleeId* is a parameter that contains the local address of the connected entity. The *buf* portion of the WSABUF pointed to by *lpCalleeId* points to a sockaddr structure. The **sockaddr** structure is interpreted according to its address family (typically by casting the **sockaddr** to some type specific to the address family such as struct **sockaddr_in**).

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

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

- If *g* is an existing socket group identifier, add *s* to this group, provided all the requirements set by this group are met.
- If *g* = SG_UNCONSTRAINED_GROUP, create an unconstrained socket group and have *s* as the first member.
- If *g* = SG_CONSTRAINED_GROUP, create a constrained socket group and have *s* as the first member.
- If *g* = zero, no group operation is performed.

For unconstrained groups, any set of sockets can be grouped together as long as they are supported by a single service provider. A constrained socket group can consist only of connection-oriented sockets, and requires that connections on all grouped sockets be to the same address on the same host. For newly created socket groups, the new group identifier can be retrieved by using getsockopt function with *level* parameter set to SOL_SOCKET and the *optname* parameter set to **SO_GROUP_ID**. A socket group and its associated socket group ID remain valid until the last socket belonging to this socket group is closed. Socket group IDs are unique across all processes for a given service provider. A socket group and its associated identifier remain valid until the last socket belonging to this socket group is closed. Socket group identifiers are unique across all processes for a given service provider. For more information on socket groups, see the Remarks for the WSASocket functions.

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

> **Note** To protect use of the **WSAAccept** function from SYN attacks, applications must perform full TCP handshakes (SYN-SYNACK-ACK) before reporting the connection request. Protecting against SYN attacks in this manner results in the SO_CONDITIONAL_ACCEPT socket option becoming inoperative; the conditional function is still called, and the **WSAAccept** function operates properly, but server applications that rely on clients being unable to perform the handshake will not operate properly.

> **Note** When issuing a blocking Winsock call such as **WSAAccept**, Winsock may need to wait for a network event before the call can complete. Winsock performs an alertable wait in this situation, which can be interrupted by an asynchronous procedure call (APC) scheduled on the same thread. Issuing another blocking Winsock call inside an APC that interrupted an ongoing blocking Winsock call on the same thread will lead to undefined behavior, and must never be attempted by Winsock clients.

Example Code

The following example demonstrates the use of the **WSAAccept** function.

```
#include <winsock2.h>
#include <stdio.h>
#include <windows.h>

/* Define an example conditional function that depends on the pQos field */
int CALLBACK ConditionAcceptFunc(
    LPWSABUF lpCallerId,
    LPWSABUF lpCallerData,
    LPQOS pQos,
    LPQOS lpGQOS,
    LPWSABUF lpCalleeId,
    LPWSABUF lpCalleeData,
    GROUP FAR * g,
    DWORD_PTR dwCallbackData
    )
{

    if (pQos != NULL) {
        RtlZeroMemory(pQos, sizeof(QOS));
        return CF_ACCEPT;
    } else
        return CF_REJECT;
}
```

```
}

int main() {

    /* Declare and initialize variables */
    WSADATA wsaData;
    SOCKET ListenSocket, AcceptSocket;
    struct sockaddr_in saClient;
    int iClientSize = sizeof(saClient);
    u_short port = 27015;
    char* ip;
    sockaddr_in service;
    int error;

    /* Initialize Winsock */
    error = WSAStartup(MAKEWORD(2,2), &wsaData);
    if (error) {
        printf("WSAStartup() failed with error: %d\n", error);
        return 1;
    }

    /* Create a TCP listening socket */
    ListenSocket = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
    if (ListenSocket == INVALID_SOCKET) {
        printf("socket() failed with error: %d\n", WSAGetLastError() );
        WSACleanup();
        return 1;
    }

    /*----------------------------------------
     *  Set up the sock addr structure that the listening socket
     *  will be bound to. In this case, the structure holds the
     * local IP address and the port specified. */
    service.sin_family = AF_INET;
    service.sin_port = htons(port);
    hostent* thisHost;
    thisHost = gethostbyname("");
    ip = inet_ntoa (*(struct in_addr *)*thisHost->h_addr_list);
    service.sin_addr.s_addr = inet_addr(ip);

    /*----------------------------------------
     *  Bind the listening socket to the IP address.
     * and port number specified by the sockaddr structure. */
    error = bind(ListenSocket, (SOCKADDR *) &service, sizeof(SOCKADDR));
    if (error == SOCKET_ERROR) {
        printf("bind() failed with error: %d\n", WSAGetLastError() );
        closesocket(ListenSocket);
        WSACleanup();
        return 1;
    }

    /* Make the socket listen for incoming connection requests */
    error = listen(ListenSocket, 1);
    if (error == SOCKET_ERROR) {
        printf("listen() failed with error: %d\n", WSAGetLastError() );
        closesocket(ListenSocket);
        WSACleanup();
        return 1;
    }
    printf("Listening...\n");

    /*----------------------------------------
     *  Accept an incoming connection request on the
     *  listening socket and transfer control to the
     * accepting socket. */
    AcceptSocket = WSAAccept(ListenSocket, (SOCKADDR*) &saClient, &iClientSize,
        &ConditionAcceptFunc, NULL);

    /*  Now do some work with the AcceptSocket
     *  At this point, the application could
```

```
     * At this point, the application could
     *  handle data transfer on the socket, or other socket
     * functionality.*/

    /* Then clean up and quit */

    closesocket(AcceptSocket);
    closesocket(ListenSocket);
    WSACleanup();

    return 0;
}
```

Windows Phone 8: This function is supported for Windows Phone Store apps on Windows Phone 8 and later.

Windows 8.1 and **Windows Server 2012 R2**: This function is supported for Windows Store apps on Windows 8.1, Windows Server 2012 R2, and later.

Requirements

Minimum supported client	Windows 8.1, Windows Vista [desktop apps | UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps | UWP apps]
Target Platform	Windows
Header	winsock2.h
Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

WSAAsyncSelect

WSAConnect

WSASocket

Winsock Functions

Winsock Reference

accept

bind

connect

getsockopt

listen

select

sockaddr

socket

WSAAddressToStringA function (winsock2.h)

10/22/2022 • 3 minutes to read • Edit Online

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

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

Syntax

```
INT WSAAPI WSAAddressToStringA(
  [in]           LPSOCKADDR          lpsaAddress,
  [in]           DWORD               dwAddressLength,
  [in, optional] LPWSAPROTOCOL_INFOA lpProtocolInfo,
  [in, out]      LPSTR               lpszAddressString,
  [in, out]      LPDWORD             lpdwAddressStringLength
);
```

Parameters

`[in] lpsaAddress`

A pointer to the sockaddr structure to translate into a string.

`[in] dwAddressLength`

The length, in bytes, of the address in the sockaddr structure pointed to by the *lpsaAddress* parameter. The *dwAddressLength* parameter may vary in size with different protocols.

`[in, optional] lpProtocolInfo`

A pointer to the WSAPROTOCOL_INFO structure for a particular provider. If this is parameter is **NULL**, the call is routed to the provider of the first protocol supporting the address family indicated in the *lpsaAddress* parameter.

`[in, out] lpszAddressString`

A pointer to the buffer that receives the human-readable address string.

`[in, out] lpdwAddressStringLength`

On input, this parameter specifies the length of the buffer pointed to by the *lpszAddressString* parameter. The length is represented in bytes for ANSI strings, and in WCHARs for Unicode strings. On output, this parameter returns the length of the string including the **NULL** terminator actually copied into the buffer pointed to by the *lpszAddressString* parameter. If the specified buffer is not large enough, the function fails with a specific error of WSAEFAULT and this parameter is updated with the required size.

Return value

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

ERROR CODE	MEANING
WSAEFAULT	The specified *lpcsAddress*, *lpProtocolInfo*, and *lpszAddressString* parameters point to memory that is not all in the address space of the process, or the buffer pointed to by the *lpszAddressString* parameter is too small. Pass in a larger buffer.
WSAEINVAL	An invalid parameter was passed. This error is returned if the *lpsaAddress*, *dwAddressLength*, or *lpdwAddressStringLength* parameter are **NULL**. This error is also returned if the specified address is not a valid socket address, or no transport provider supports the indicated address family.
WSAENOBUFS	No buffer space is available.
WSANOTINITIALISED	The Winsock 2 DLL has not been initialized. The application must first call WSAStartup before calling any Windows Sockets functions.

Remarks

The **WSAAddressToString** function provides a protocol-independent address-to-string translation. The **WSAAddressToString** function takes a socket address structure pointed to by the *lpsaAddress* parameter and returns a pointer to **NULL**-terminated string that represents the socket address in the *lpszAddressString* parameter. While the inet_ntoa function works only with IPv4 addresses, the **WSAAddressToString** function works with any socket address supported by a Winsock provider on the local computer including IPv6 addresses.

If the *lpsaAddress* parameter points to an IPv4 socket address (the address family is **AF_INET**), then the address string returned in the buffer pointed to by the *lpszAddressString* parameter is in dotted-decimal notation as in "192.168.16.0", an example of an IPv4 address in dotted-decimal notation.

If the *lpsaAddress* parameter points to an IPv6 socket address (the address family is **AF_INET6**), then the address string returned in the buffer pointed to by the *lpszAddressString* parameter is in Internet standard format. The basic string representation consists of 8 hexadecimal numbers separated by colons. A string of consecutive zero numbers is replaced with a double-colon. There can only be one double-colon in the string representation of the IPv6 address.

If the length of the buffer pointed to by the *lpszAddressString* parameter is not large enough to receive the string representation of the socket address, **WSAAddressToString** returns WSAEFAULT.

Support for IPv6 addresses using the **WSAAddressToString** function was added on Windows XP with Service Pack 1 (SP1)and later. IPv6 must also be installed on the local computer for the **WSAAddressToString** function to support IPv6 addresses.

Windows Phone 8: The **WSAAddressToStringW** function is supported for Windows Phone Store apps on Windows Phone 8 and later.

Windows 8.1 and **Windows Server 2012 R2**: The **WSAAddressToStringW** function is supported for Windows Store apps on Windows 8.1, Windows Server 2012 R2, and later.

> **NOTE**
> The winsock2.h header defines WSAAddressToString as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see Conventions for Function Prototypes.

Requirements

Minimum supported client	Windows 8.1, Windows Vista [desktop apps | UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps | UWP apps]
Target Platform	Windows
Header	winsock2.h
Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

InetNtop

InetPton

RtlIpv4AddressToString

RtlIpv4AddressToStringEx

RtlIpv4StringToAddress

RtlIpv4StringToAddressEx

RtlIpv6AddressToString

RtlIpv6AddressToStringEx

RtlIpv6StringToAddress

RtlIpv6StringToAddressEx

WSAPROTOCOL_INFO

WSAStartup

WSAStringToAddress

inet_addr

inet_ntoa

sockaddr

WSAAddressToStringW function (winsock2.h)

10/22/2022 • 3 minutes to read • Edit Online

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

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

Syntax

```
INT WSAAPI WSAAddressToStringW(
  [in]           LPSOCKADDR          lpsaAddress,
  [in]           DWORD               dwAddressLength,
  [in, optional] LPWSAPROTOCOL_INFOW lpProtocolInfo,
  [in, out]      LPWSTR              lpszAddressString,
  [in, out]      LPDWORD             lpdwAddressStringLength
);
```

Parameters

`[in] lpsaAddress`

A pointer to the sockaddr structure to translate into a string.

`[in] dwAddressLength`

The length, in bytes, of the address in the sockaddr structure pointed to by the *lpsaAddress* parameter. The *dwAddressLength* parameter may vary in size with different protocols.

`[in, optional] lpProtocolInfo`

A pointer to the WSAPROTOCOL_INFO structure for a particular provider. If this is parameter is **NULL**, the call is routed to the provider of the first protocol supporting the address family indicated in the *lpsaAddress* parameter.

`[in, out] lpszAddressString`

A pointer to the buffer that receives the human-readable address string.

`[in, out] lpdwAddressStringLength`

On input, this parameter specifies the length of the buffer pointed to by the *lpszAddressString* parameter. The length is represented in bytes for ANSI strings, and in WCHARs for Unicode strings. On output, this parameter returns the length of the string including the **NULL** terminator actually copied into the buffer pointed to by the *lpszAddressString* parameter. If the specified buffer is not large enough, the function fails with a specific error of WSAEFAULT and this parameter is updated with the required size.

Return value

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

ERROR CODE	MEANING
WSAEFAULT	The specified *lpcsAddress*, *lpProtocolInfo*, and *lpszAddressString* parameters point to memory that is not all in the address space of the process, or the buffer pointed to by the *lpszAddressString* parameter is too small. Pass in a larger buffer.
WSAEINVAL	An invalid parameter was passed. This error is returned if the *lpsaAddress*, *dwAddressLength*, or *lpdwAddressStringLength* parameter are **NULL**. This error is also returned if the specified address is not a valid socket address, or no transport provider supports the indicated address family.
WSAENOBUFS	No buffer space is available.
WSANOTINITIALISED	The Winsock 2 DLL has not been initialized. The application must first call WSAStartup before calling any Windows Sockets functions.

Remarks

The **WSAAddressToString** function provides a protocol-independent address-to-string translation. The **WSAAddressToString** function takes a socket address structure pointed to by the *lpsaAddress* parameter and returns a pointer to **NULL**-terminated string that represents the socket address in the *lpszAddressString* parameter. While the inet_ntoa function works only with IPv4 addresses, the **WSAAddressToString** function works with any socket address supported by a Winsock provider on the local computer including IPv6 addresses.

If the *lpsaAddress* parameter points to an IPv4 socket address (the address family is **AF_INET**), then the address string returned in the buffer pointed to by the *lpszAddressString* parameter is in dotted-decimal notation as in "192.168.16.0", an example of an IPv4 address in dotted-decimal notation.

If the *lpsaAddress* parameter points to an IPv6 socket address (the address family is **AF_INET6**), then the address string returned in the buffer pointed to by the *lpszAddressString* parameter is in Internet standard format. The basic string representation consists of 8 hexadecimal numbers separated by colons. A string of consecutive zero numbers is replaced with a double-colon. There can only be one double-colon in the string representation of the IPv6 address.

If the length of the buffer pointed to by the *lpszAddressString* parameter is not large enough to receive the string representation of the socket address, **WSAAddressToString** returns WSAEFAULT.

Support for IPv6 addresses using the **WSAAddressToString** function was added on Windows XP with Service Pack 1 (SP1)and later. IPv6 must also be installed on the local computer for the **WSAAddressToString** function to support IPv6 addresses.

Windows Phone 8: The **WSAAddressToStringW** function is supported for Windows Phone Store apps on Windows Phone 8 and later.

Windows 8.1 and **Windows Server 2012 R2**: The **WSAAddressToStringW** function is supported for Windows Store apps on Windows 8.1, Windows Server 2012 R2, and later.

> **NOTE**
> The winsock2.h header defines WSAAddressToString as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see Conventions for Function Prototypes.

Requirements

Minimum supported client	Windows 8.1, Windows Vista [desktop apps | UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps | UWP apps]
Target Platform	Windows
Header	winsock2.h
Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

InetNtop

InetPton

RtlIpv4AddressToString

RtlIpv4AddressToStringEx

RtlIpv4StringToAddress

RtlIpv4StringToAddressEx

RtlIpv6AddressToString

RtlIpv6AddressToStringEx

RtlIpv6StringToAddress

RtlIpv6StringToAddressEx

WSAPROTOCOL_INFO

WSAStartup

WSAStringToAddress

inet_addr

inet_ntoa

sockaddr

WSAAsyncGetHostByAddr function (winsock2.h)

10/22/2022 • 4 minutes to read • Edit Online

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

> **Note** The **WSAAsyncGetHostByAddr** function is not designed to provide parallel resolution of several addresses. Therefore, applications that issue several requests should not expect them to be executed concurrently. Alternatively, applications can start another thread and use the getnameinfo function to resolve addresses in an IP-version agnostic manner. Developers creating Windows Sockets 2 applications are urged to use the **getnameinfo** function to enable smooth transition to IPv6 compatibility.

Syntax

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

Parameters

`hWnd`

TBD

`wMsg`

TBD

`addr`

TBD

`len`

TBD

`type`

TBD

`buf`

TBD

`buflen`

TBD

Return value

The return value specifies whether or not the asynchronous operation was successfully initiated. It does not imply success or failure of the operation itself.

If no error occurs, **WSAAsyncGetHostByAddr** returns a nonzero value of type HANDLE that is the asynchronous task handle (not to be confused with a Windows HTASK) for the request. This value can be used in two ways. It can be used to cancel the operation using WSACancelAsyncRequest, or it can be used to match up asynchronous operations and completion messages by examining the *wParam* message parameter.

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

The following error codes can be set when an application window receives a message. As described above, they can be extracted from the *lParam* in the reply message using the **WSAGETASYNCERROR** macro.

ERROR CODE	MEANING
WSAENETDOWN	The network subsystem has failed.
WSAENOBUFS	Insufficient buffer space is available.
WSAEFAULT	The *addr* or *buf* parameter is not in a valid part of the process address space.
WSAHOST_NOT_FOUND	Authoritative answer host not found.
WSATRY_AGAIN	Nonauthoritative host not found, or SERVERFAIL.
WSANO_RECOVERY	Nonrecoverable errors: FORMERR, REFUSED, NOTIMP.
WSANO_DATA	Valid name, no data record of requested type.

The following errors can occur at the time of the function call, and indicate that the asynchronous operation could not be initiated.

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

WSAEWOULDBLOCK	The asynchronous operation cannot be scheduled at this time due to resource or other constraints within the Windows Sockets implementation.

Remarks

The **WSAAsyncGetHostByAddr** function is an asynchronous version of gethostbyaddr. It is used to retrieve the host name and address information that corresponds to a network address. Windows Sockets initiates the operation and returns to the caller immediately, passing back an opaque, asynchronous task handle that the application can use to identify the operation. When the operation is completed, the results (if any) are copied into the buffer provided by the caller and a message is sent to the application's window.

When the asynchronous operation has completed, the application window indicated by the *hWnd* parameter receives message in the *wMsg* parameter. The *wParam* parameter contains the asynchronous task handle as returned by the original function call. The high 16 bits of *lParam* contain any error code. The error code can be any error as defined in Winsock2.h. An error code of zero indicates successful completion of the asynchronous operation.

On successful completion, the buffer specified to the original function call contains a hostent structure. To access the members of this structure, the original buffer address is cast to a **hostent** structure pointer and accessed as appropriate.

If the error code is WSAENOBUFS, the size of the buffer specified by *buflen* in the original call was too small to contain all the resulting information. In this case, the low 16 bits of *lParam* contain the size of buffer required to supply all the requisite information. If the application decides that the partial data is inadequate, it can reissue the **WSAAsyncGetHostByAddr** function call with a buffer large enough to receive all the desired information (that is, no smaller than the low 16 bits of *lParam*).

The buffer specified to this function is used by Windows Sockets to construct a structure together with the contents of data areas referenced by members of the same hostent structure. To avoid the WSAENOBUFS error, the application should provide a buffer of at least MAXGETHOSTSTRUCT bytes (as defined in Winsock2.h).

The error code and buffer length should be extracted from the *lParam* using the macros **WSAGETASYNCERROR** and **WSAGETASYNCBUFLEN**, defined in Winsock2.h as:

```
#include <windows.h>

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

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

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winsock2.h (include Winsock2.h, Winsock.h)

Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

WSACancelAsyncRequest

Winsock Functions

Winsock Reference

getaddrinfo

gethostbyaddr

getnameinfo

hostent

WSAAsyncGetHostByName function (winsock2.h)

10/22/2022 • 4 minutes to read • Edit Online

The **WSAAsyncGetHostByName** function asynchronously retrieves host information that corresponds to a host name.

> **Note** The **WSAAsyncGetHostByName** function is not designed to provide parallel resolution of several names. Therefore, applications that issue several requests should not expect them to be executed concurrently. Alternatively, applications can start another thread and use the getaddrinfo function to resolve names in an IP-version agnostic manner. Developers creating Windows Sockets 2 applications are urged to use the **getaddrinfo** function to enable smooth transition to IPv6 compatibility.

Syntax

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

Parameters

`hWnd`

TBD

`wMsg`

TBD

`name`

TBD

`buf`

TBD

`buflen`

TBD

Return value

The return value specifies whether or not the asynchronous operation was successfully initiated. It does not imply success or failure of the operation itself.

If no error occurs, **WSAAsyncGetHostByName** returns a nonzero value of type HANDLE that is the asynchronous task handle (not to be confused with a Windows HTASK) for the request. This value can be used in two ways. It can be used to cancel the operation using WSACancelAsyncRequest, or it can be used to match up

asynchronous operations and completion messages by examining the *wParam* message parameter.

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

The following error codes can be set when an application window receives a message. As described above, they can be extracted from the *lParam* in the reply message using the WSAGETASYNCERROR macro.

ERROR CODE	MEANING
WSAENETDOWN	The network subsystem has failed.
WSAENOBUFS	Insufficient buffer space is available.
WSAEFAULT	The *name* or *buf* parameter is not in a valid part of the process address space.
WSAHOST_NOT_FOUND	Authoritative answer host not found.
WSATRY_AGAIN	A nonauthoritative host not found, or SERVERFAIL.
WSANO_RECOVERY	Nonrecoverable errors: FORMERR, REFUSED, NOTIMP.
WSANO_DATA	Valid name, no data record of requested type.

The following errors can occur at the time of the function call, and indicate that the asynchronous operation could not be initiated.

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

Remarks

The **WSAAsyncGetHostByName** function is an asynchronous version of gethostbyname, and is used to retrieve host name and address information corresponding to a host name. Windows Sockets initiates the operation and returns to the caller immediately, passing back an opaque asynchronous task handle that which

the application can use to identify the operation. When the operation is completed, the results (if any) are copied into the buffer provided by the caller and a message is sent to the application's window.

When the asynchronous operation has completed, the application window indicated by the *hWnd* parameter receives message in the *wMsg* parameter. The *wParam* parameter contains the asynchronous task handle as returned by the original function call. The high 16 bits of *lParam* contain any error code. The error code can be any error as defined in Winsock2.h. An error code of zero indicates successful completion of the asynchronous operation.

On successful completion, the buffer specified to the original function call contains a hostent structure. To access the elements of this structure, the original buffer address should be cast to a **hostent** structure pointer and accessed as appropriate.

If the error code is WSAENOBUFS, the size of the buffer specified by *buflen* in the original call was too small to contain all the resulting information. In this case, the low 16 bits of *lParam* contain the size of buffer required to supply all the requisite information. If the application decides that the partial data is inadequate, it can reissue the **WSAAsyncGetHostByName** function call with a buffer large enough to receive all the desired information (that is, no smaller than the low 16 bits of *lParam*).

The buffer specified to this function is used by Windows Sockets to construct a hostent structure together with the contents of data areas referenced by members of the same **hostent** structure. To avoid the WSAENOBUFS error, the application should provide a buffer of at least MAXGETHOSTSTRUCT bytes (as defined in Winsock2.h).

The error code and buffer length should be extracted from the *lParam* using the macros **WSAGETASYNCERROR** and **WSAGETASYNCBUFLEN**, defined in Winsock2.h as:

```
#include <windows.h>

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

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

WSAAsyncGetHostByName is guaranteed to resolve the string returned by a successful call to gethostname.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winsock2.h (include Winsock2.h, Winsock.h)
Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

WSACancelAsyncRequest

Winsock Functions

Winsock Reference

getaddrinfo

gethostbyname

getnameinfo

hostent

WSAAsyncGetProtoByName function (winsock2.h)

10/22/2022 • 4 minutes to read • Edit Online

The **WSAAsyncGetProtoByName** function asynchronously retrieves protocol information that corresponds to a protocol name.

Syntax

```
HANDLE WSAAPI WSAAsyncGetProtoByName(
  [in]  HWND       hWnd,
  [in]  u_int      wMsg,
  [in]  const char *name,
  [out] char       *buf,
  [out] int        buflen
);
```

Parameters

`[in] hWnd`

Handle of the window that will receive a message when the asynchronous request completes.

`[in] wMsg`

Message to be received when the asynchronous request completes.

`[in] name`

Pointer to the null-terminated protocol name to be resolved.

`[out] buf`

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

`[out] buflen`

Size of data area for the *buf* parameter, in bytes.

Return value

The return value specifies whether or not the asynchronous operation was successfully initiated. It does not imply success or failure of the operation itself.

If no error occurs, **WSAAsyncGetProtoByName** returns a nonzero value of type HANDLE that is the asynchronous task handle for the request (not to be confused with a Windows HTASK). This value can be used in two ways. It can be used to cancel the operation using WSACancelAsyncRequest, or it can be used to match up asynchronous operations and completion messages, by examining the *wParam* message parameter.

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

The following error codes can be set when an application window receives a message. As described above, they can be extracted from the *lParam* in the reply message using the **WSAGETASYNCERROR** macro.

ERROR CODE	MEANING
WSAENETDOWN	The network subsystem has failed.
WSAENOBUFS	Insufficient buffer space is available.
WSAEFAULT	The *name* or *buf* parameter is not in a valid part of the process address space.
WSAHOST_NOT_FOUND	Authoritative answer protocol not found.
WSATRY_AGAIN	A nonauthoritative protocol not found, or server failure.
WSANO_RECOVERY	Nonrecoverable errors, the protocols database is not accessible.
WSANO_DATA	Valid name, no data record of requested type.

The following errors can occur at the time of the function call, and indicate that the asynchronous operation could not be initiated.

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

Remarks

The **WSAAsyncGetProtoByName** function is an asynchronous version of getprotobyname. It is used to retrieve the protocol name and number from the Windows Sockets database corresponding to a given protocol name. Windows Sockets initiates the operation and returns to the caller immediately, passing back an opaque, asynchronous task handle that the application can use to identify the operation. When the operation is completed, the results (if any) are copied into the buffer provided by the caller and a message is sent to the application's window.

When the asynchronous operation has completed, the application window indicated by the *hWnd* parameter receives message in the *wMsg* parameter. The *wParam* parameter contains the asynchronous task handle as returned by the original function call. The high 16 bits of *lParam* contain any error code. The error code can be any error as defined in Winsock2.h. An error code of zero indicates successful completion of the asynchronous operation.

On successful completion, the buffer specified to the original function call contains a protoent structure. To access the members of this structure, the original buffer address should be cast to a **protoent** structure pointer and accessed as appropriate.

If the error code is WSAENOBUFS, the size of the buffer specified by *buflen* in the original call was too small to contain all the resulting information. In this case, the low 16 bits of *lParam* contain the size of buffer required to supply all the requisite information. If the application decides that the partial data is inadequate, it can reissue the **WSAAsyncGetProtoByName** function call with a buffer large enough to receive all the desired information (that is, no smaller than the low 16 bits of *lParam*).

The buffer specified to this function is used by Windows Sockets to construct a protoent structure together with the contents of data areas referenced by members of the same **protoent** structure. To avoid the WSAENOBUFS error noted above, the application should provide a buffer of at least MAXGETHOSTSTRUCT bytes (as defined in Winsock2.h).

The error code and buffer length should be extracted from the *lParam* using the macros **WSAGETASYNCERROR** and **WSAGETASYNCBUFLEN**, defined in Winsock2.h as:

```
#include <windows.h>

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

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

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winsock2.h (include Winsock2.h)
Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

WSACancelAsyncRequest

Winsock Functions

Winsock Reference

getprotobyname

WSAAsyncGetProtoByNumber function (winsock2.h)

10/22/2022 • 4 minutes to read • Edit Online

The **WSAAsyncGetProtoByNumber** function asynchronously retrieves protocol information that corresponds to a protocol number.

Syntax

```
HANDLE WSAAPI WSAAsyncGetProtoByNumber(
  [in]  HWND  hWnd,
  [in]  u_int wMsg,
  [in]  int   number,
  [out] char  *buf,
  [in]  int   buflen
);
```

Parameters

`[in] hWnd`

Handle of the window that will receive a message when the asynchronous request completes.

`[in] wMsg`

Message to be received when the asynchronous request completes.

`[in] number`

Protocol number to be resolved, in host byte order.

`[out] buf`

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

`[in] buflen`

Size of data area for the *buf* parameter, in bytes.

Return value

The return value specifies whether or not the asynchronous operation was successfully initiated. It does not imply success or failure of the operation itself.

If no error occurs, **WSAAsyncGetProtoByNumber** returns a nonzero value of type **HANDLE** that is the asynchronous task handle for the request (not to be confused with a Windows HTASK). This value can be used in two ways. It can be used to cancel the operation using WSACancelAsyncRequest, or it can be used to match up asynchronous operations and completion messages, by examining the *wParam* message parameter.

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

The following error codes can be set when an application window receives a message. As described above, they can be extracted from the *lParam* in the reply message using the **WSAGETASYNCERROR** macro.

ERROR CODE	MEANING
WSAENETDOWN	The network subsystem has failed.
WSAENOBUFS	Insufficient buffer space is available.
WSAEFAULT	The *buf* parameter is not in a valid part of the process address space.
WSAHOST_NOT_FOUND	Authoritative answer protocol not found.
WSATRY_AGAIN	Nonauthoritative protocol not found, or server failure.
WSANO_RECOVERY	Nonrecoverable errors, the protocols database is not accessible.
WSANO_DATA	Valid name, no data record of requested type.

The following errors can occur at the time of the function call, and indicate that the asynchronous operation could not be initiated.

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

Remarks

The **WSAAsyncGetProtoByNumber** function is an asynchronous version of getprotobynumber, and is used to retrieve the protocol name and number corresponding to a protocol number. Windows Sockets initiates the operation and returns to the caller immediately, passing back an opaque, asynchronous task handle that the application can use to identify the operation. When the operation is completed, the results (if any) are copied into the buffer provided by the caller and a message is sent to the application's window.

When the asynchronous operation has completed, the application window indicated by the *hWnd* parameter

receives message in the *wMsg* parameter. The *wParam* parameter contains the asynchronous task handle as returned by the original function call. The high 16 bits of *lParam* contain any error code. The error code can be any error as defined in Winsock2.h. An error code of zero indicates successful completion of the asynchronous operation.

On successful completion, the buffer specified to the original function call contains a protoent structure. To access the members of this structure, the original buffer address is cast to a **protoent** structure pointer and accessed as appropriate.

If the error code is WSAENOBUFS, the size of the buffer specified by *buflen* in the original call was too small to contain all the resulting information. In this case, the low 16 bits of *lParam* contain the size of buffer required to supply all the requisite information. If the application decides that the partial data is inadequate, it can reissue the **WSAAsyncGetProtoByNumber** function call with a buffer large enough to receive all the desired information (that is, no smaller than the low 16 bits of *lParam*).

The buffer specified to this function is used by Windows Sockets to construct a protoent structure together with the contents of data areas referenced by members of the same **protoent** structure. To avoid the WSAENOBUFS error noted above, the application should provide a buffer of at least MAXGETHOSTSTRUCT bytes (as defined in Winsock2.h).

The error code and buffer length should be extracted from the *lParam* using the macros **WSAGETASYNCERROR** and **WSAGETASYNCBUFLEN**, defined in Winsock2.h as:

```
#include <windows.h>

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

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

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winsock2.h (include Winsock2.h)
Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

WSACancelAsyncRequest

Winsock Functions

Winsock Reference

WSAAsyncGetServByName function (winsock2.h)

10/22/2022 • 4 minutes to read • Edit Online

The **WSAAsyncGetServByName** function asynchronously retrieves service information that corresponds to a service name and port.

Syntax

```
HANDLE WSAAPI WSAAsyncGetServByName(
  [in]  HWND       hWnd,
  [in]  u_int      wMsg,
  [in]  const char *name,
  [in]  const char *proto,
  [out] char       *buf,
  [in]  int        buflen
);
```

Parameters

`[in] hWnd`

Handle of the window that should receive a message when the asynchronous request completes.

`[in] wMsg`

Message to be received when the asynchronous request completes.

`[in] name`

Pointer to a **null**-terminated service name.

`[in] proto`

Pointer to a protocol name. This can be **NULL**, in which case **WSAAsyncGetServByName** will search for the first service entry for which *s_name* or one of the *s_aliases* matches the given *name*. Otherwise, **WSAAsyncGetServByName** matches both *name* and *proto*.

`[out] buf`

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

`[in] buflen`

Size of data area for the *buf* parameter, in bytes.

Return value

The return value specifies whether or not the asynchronous operation was successfully initiated. It does not imply success or failure of the operation itself.

If no error occurs, **WSAAsyncGetServByName** returns a nonzero value of type **HANDLE** that is the

asynchronous task handle for the request (not to be confused with a Windows HTASK). This value can be used in two ways. It can be used to cancel the operation using WSACancelAsyncRequest, or it can be used to match up asynchronous operations and completion messages, by examining the *wParam* message parameter.

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

The following error codes can be set when an application window receives a message. As described above, they can be extracted from the *lParam* in the reply message using the **WSAGETASYNCERROR** macro.

ERROR CODE	MEANING
WSAENETDOWN	The network subsystem has failed.
WSAENOBUFS	Insufficient buffer space is available.
WSAEFAULT	The *buf* parameter is not in a valid part of the process address space.
WSAHOST_NOT_FOUND	Authoritative answer host not found.
WSATRY_AGAIN	Nonauthoritative service not found, or server failure.
WSANO_RECOVERY	Nonrecoverable errors, the services database is not accessible.
WSANO_DATA	Valid name, no data record of requested type.

The following errors can occur at the time of the function call, and indicate that the asynchronous operation could not be initiated.

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

Remarks

The **WSAAsyncGetServByName** function is an asynchronous version of getservbyname and is used to

retrieve service information corresponding to a service name. Windows Sockets initiates the operation and returns to the caller immediately, passing back an opaque, asynchronous task handle that the application can use to identify the operation. When the operation is completed, the results (if any) are copied into the buffer provided by the caller and a message is sent to the application's window.

When the asynchronous operation has completed, the application window indicated by the *hWnd* parameter receives message in the *wMsg* parameter. The *wParam* parameter contains the asynchronous task handle as returned by the original function call. The high 16 bits of *lParam* contain any error code. The error code can be any error as defined in Winsock2.h. An error code of zero indicates successful completion of the asynchronous operation.

On successful completion, the buffer specified to the original function call contains a servent structure. To access the members of this structure, the original buffer address should be cast to a **servent** structure pointer and accessed as appropriate.

If the error code is WSAENOBUFS, the size of the buffer specified by *buflen* in the original call was too small to contain all the resulting information. In this case, the low 16 bits of *lParam* contain the size of buffer required to supply all the requisite information. If the application decides that the partial data is inadequate, it can reissue the **WSAAsyncGetServByName** function call with a buffer large enough to receive all the desired information (that is, no smaller than the low 16 bits of *lParam*).

The buffer specified to this function is used by Windows Sockets to construct a servent structure together with the contents of data areas referenced by members of the same **servent** structure. To avoid the WSAENOBUFS error, the application should provide a buffer of at least MAXGETHOSTSTRUCT bytes (as defined in Winsock2.h).

The error code and buffer length should be extracted from the *lParam* using the macros **WSAGETASYNCERROR** and **WSAGETASYNCBUFLEN**, defined in Winsock2.h as:

```
#include <windows.h>

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

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

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winsock2.h (include Winsock2.h)
Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

WSACancelAsyncRequest

Winsock Functions

Winsock Reference

getservbyname

WSAAsyncGetServByPort function (winsock2.h)

10/22/2022 • 4 minutes to read • Edit Online

The **WSAAsyncGetServByPort** function asynchronously retrieves service information that corresponds to a port and protocol.

Syntax

```
HANDLE WSAAPI WSAAsyncGetServByPort(
  [in]  HWND       hWnd,
  [in]  u_int      wMsg,
  [in]  int        port,
  [in]  const char *proto,
  [out] char       *buf,
  [in]  int        buflen
);
```

Parameters

`[in] hWnd`

Handle of the window that should receive a message when the asynchronous request completes.

`[in] wMsg`

Message to be received when the asynchronous request completes.

`[in] port`

Port for the service, in network byte order.

`[in] proto`

Pointer to a protocol name. This can be **NULL**, in which case **WSAAsyncGetServByPort** will search for the first service entry for which *s_port* match the given *port*. Otherwise, **WSAAsyncGetServByPort** matches both *port* and *proto*.

`[out] buf`

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

`[in] buflen`

Size of data area for the *buf* parameter, in bytes.

Return value

The return value specifies whether or not the asynchronous operation was successfully initiated. It does not imply success or failure of the operation itself.

If no error occurs, **WSAAsyncGetServByPort** returns a nonzero value of type **HANDLE** that is the

asynchronous task handle for the request (not to be confused with a Windows HTASK). This value can be used in two ways. It can be used to cancel the operation using WSACancelAsyncRequest, or it can be used to match up asynchronous operations and completion messages, by examining the *wParam* message parameter.

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

The following error codes can be set when an application window receives a message. As described above, they can be extracted from the *lParam* in the reply message using the **WSAGETASYNCERROR** macro.

ERROR CODE	MEANING
WSAENETDOWN	The network subsystem has failed.
WSAENOBUFS	Insufficient buffer space is available.
WSAEFAULT	The *proto* or *buf* parameter is not in a valid part of the process address space.
WSAHOST_NOT_FOUND	Authoritative answer port not found.
WSATRY_AGAIN	Nonauthoritative port not found, or server failure.
WSANO_RECOVERY	Nonrecoverable errors, the services database is not accessible.
WSANO_DATA	Valid name, no data record of requested type.

The following errors can occur at the time of the function call, and indicate that the asynchronous operation could not be initiated.

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

Remarks

The **WSAAsyncGetServByPort** function is an asynchronous version of getservbyport, and is used to retrieve

service information corresponding to a port number. Windows Sockets initiates the operation and returns to the caller immediately, passing back an opaque, asynchronous task handle that the application can use to identify the operation. When the operation is completed, the results (if any) are copied into the buffer provided by the caller and a message is sent to the application's window.

When the asynchronous operation has completed, the application window indicated by the *hWnd* parameter receives message in the *wMsg* parameter. The *wParam* parameter contains the asynchronous task handle as returned by the original function call. The high 16 bits of *lParam* contain any error code. The error code can be any error as defined in Winsock2.h. An error code of zero indicates successful completion of the asynchronous operation.

On successful completion, the buffer specified to the original function call contains a servent structure. To access the members of this structure, the original buffer address should be cast to a **servent** structure pointer and accessed as appropriate.

If the error code is WSAENOBUFS, the size of the buffer specified by *buflen* in the original call was too small to contain all the resulting information. In this case, the low 16 bits of *lParam* contain the size of buffer required to supply all the requisite information. If the application decides that the partial data is inadequate, it can reissue the **WSAAsyncGetServByPort** function call with a buffer large enough to receive all the desired information (that is, no smaller than the low 16 bits of *lParam*).

The buffer specified to this function is used by Windows Sockets to construct a servent structure together with the contents of data areas referenced by members of the same **servent** structure. To avoid the WSAENOBUFS error, the application should provide a buffer of at least MAXGETHOSTSTRUCT bytes (as defined in Winsock2.h).

The error code and buffer length should be extracted from the *lParam* using the macros **WSAGETASYNCERROR** and **WSAGETASYNCBUFLEN**, defined in Winsock2.h as:

```
#include <windows.h>

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

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

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winsock2.h (include Winsock2.h)
Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

WSACancelAsyncRequest

Winsock Functions

Winsock Reference

getservbyport

WSAAsyncSelect function (winsock2.h)

10/22/2022 • 14 minutes to read • Edit Online

[The **WSAAsyncSelect** function is available for use in the operating systems specified in the Requirements section. It may be altered or unavailable in subsequent versions. Rather than use Select-style I/O, use Overlapped I/O and Event Objects with WinSock2.]

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

Syntax

```
int WSAAPI WSAAsyncSelect(
  [in] SOCKET s,
  [in] HWND   hWnd,
  [in] u_int  wMsg,
  [in] long   lEvent
);
```

Parameters

`[in] s`

A descriptor that identifies the socket for which event notification is required.

`[in] hWnd`

A handle that identifies the window that will receive a message when a network event occurs.

`[in] wMsg`

A message to be received when a network event occurs.

`[in] lEvent`

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

Return value

If the **WSAAsyncSelect** function succeeds, the return value is zero, provided that the application's declaration of interest in the network event set was successful. Otherwise, the value SOCKET_ERROR is returned, and a specific error number can be retrieved by calling WSAGetLastError.

ERROR CODE	MEANING
WSANOTINITIALISED	A successful WSAStartup call must occur before using this function.
WSAENETDOWN	The network subsystem failed.

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

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

Event: FD_CONNECT

ERROR CODE	MEANING
WSAEAFNOSUPPORT	Addresses in the specified family cannot be used with this socket.
WSAECONNREFUSED	The attempt to connect was rejected.
WSAENETUNREACH	The network cannot be reached from this host at this time.
WSAEFAULT	The *namelen* parameter is invalid.
WSAEINVAL	The socket is already bound to an address.
WSAEISCONN	The socket is already connected.
WSAEMFILE	No more file descriptors are available.
WSAENOBUFS	No buffer space is available. The socket cannot be connected.
WSAENOTCONN	The socket is not connected.
WSAETIMEDOUT	Attempt to connect timed out without establishing a connection.

Event: FD_CLOSE

ERROR CODE	MEANING
WSAENETDOWN	The network subsystem failed.
WSAECONNRESET	The connection was reset by the remote side.
WSAECONNABORTED	The connection was terminated due to a time-out or other failure.

ERROR CODE	MEANING
WSAENETDOWN	The network subsystem failed.

Event: FD_ROUTING_INTERFACE_CHANGE

ERROR CODE	MEANING
WSAENETUNREACH	The specified destination is no longer reachable.
WSAENETDOWN	The network subsystem failed.

Remarks

The **WSAAsyncSelect** function is used to request that WS2_32.DLL should send a message to the window *hWnd* when it detects any network event specified by the *lEvent* parameter. The message that should be sent is specified by the *wMsg* parameter. The socket for which notification is required is identified by the *s* parameter.

The **WSAAsyncSelect** function automatically sets socket *s* to nonblocking mode, regardless of the value of *lEvent*. To set socket *s* back to blocking mode, it is first necessary to clear the event record associated with socket *s* via a call to **WSAAsyncSelect** with *lEvent* set to zero. You can then call ioctlsocket or WSAIoctl to set the socket back to blocking mode. For more information about how to set the nonblocking socket back to blocking mode, see the **ioctlsocket** and **WSAIoctl** functions.

The *lEvent* parameter is constructed by using the bitwise OR operator with any value listed in the following table.

VALUE	MEANING
FD_READ	Set to receive notification of readiness for reading.
FD_WRITE	Wants to receive notification of readiness for writing.
FD_OOB	Wants to receive notification of the arrival of OOB data.
FD_ACCEPT	Wants to receive notification of incoming connections.
FD_CONNECT	Wants to receive notification of completed connection or multipoint join operation.
FD_CLOSE	Wants to receive notification of socket closure.
FD_QOS	Wants to receive notification of socket Quality of Service (QoS) changes.
FD_GROUP_QOS	Wants to receive notification of socket group Quality of Service (QoS) changes (reserved for future use with socket groups). Reserved.
FD_ROUTING_INTERFACE_CHANGE	Wants to receive notification of routing interface changes for the specified destination(s).

FD_ADDRESS_LIST_CHANGE	Wants to receive notification of local address list changes for the socket protocol family.

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

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

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

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

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

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

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

The socket created by the accept function has the same properties as the listening socket used to accept it. Consequently, **WSAAsyncSelect** events set for the listening socket also apply to the accepted socket. For example, if a listening socket has **WSAAsyncSelect** events **FD_ACCEPT**, **FD_READ**, and **FD_WRITE**, then any socket accepted on that listening socket will also have **FD_ACCEPT**, **FD_READ**, and **FD_WRITE** events with the same *wMsg* value used for messages. If a different *wMsg* or events are desired, the application should call **WSAAsyncSelect**, passing the accepted socket and the desired new data.

When one of the nominated network events occurs on the specified socket *s*, the application window *hWnd* receives message *wMsg*. The *wParam* parameter identifies the socket on which a network event has occurred. The low word of *lParam* specifies the network event that has occurred. The high word of *lParam* contains any error code. The error code be any error as defined in Winsock2.h.

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

The error and event codes can be extracted from the *lParam* using the macros **WSAGETSELECTERROR** and **WSAGETSELECTEVENT**, defined in Winsock2.h as:

```
#include <windows.h>

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

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

The possible network event codes that can be returned are listed in the following table.

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

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

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

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

6. Application issues `recv(s,...)`, which fails with the error WSAEWOULDBLOCK.

Other sequences are also possible.

The WS2_32.DLL will not continually flood an application with messages for a particular network event. Having successfully posted notification of a particular event to an application window, no further message(s) for that network event will be posted to the application window until the application makes the function call that implicitly reenables notification of that network event.

EVENT	REENABLING FUNCTION
FD_READ	recv, recvfrom, WSARecv, or WSARecvFrom.
FD_WRITE	send, sendto, WSASend, or WSASendTo.
FD_OOB	recv, recvfrom, WSARecv, or WSARecvFrom.
FD_ACCEPT	accept or WSAAccept unless the error code is WSATRY_AGAIN indicating that the condition function returned CF_DEFER.
FD_CONNECT	None.
FD_CLOSE	None.
FD_QOS	WSAIoctl with command SIO_GET_QOS.
FD_GROUP_QOS	Reserved. WSAIoctl with command SIO_GET_GROUP_QOS (reserved for future use with socket groups).
FD_ROUTING_INTERFACE_CHANGE	WSAIoctl with command SIO_ROUTING_INTERFACE_CHANGE.
FD_ADDRESS_LIST_CHANGE	WSAIoctl with command SIO_ADDRESS_LIST_CHANGE.

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

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

1. Network transport stack receives 100 bytes of data on socket *s* and causes Windows Sockets 2 to post an **FD_READ** message.
2. The application issues recv(*s*, *buffptr*, 50, 0) to read 50 bytes.
3. Another **FD_READ** message is posted because there is still data to be read.

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

The **FD_QOS** and **FD_GROUP_QOS** events are considered edge triggered. A message will be posted exactly

once when a quality of service change occurs. Further messages will not be forthcoming until either the provider detects a further change in quality of service or the application renegotiates the quality of service for the socket.

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

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

If any event has occurred when the application calls **WSAAsyncSelect** or when the reenabling function is called, then a message is posted as appropriate. For example, consider the following sequence:

1. An application calls listen.
2. A connect request is received, but not yet accepted.
3. The application calls **WSAAsyncSelect** specifying that it requires receiving **FD_ACCEPT** messages for the socket. Due to the persistence of events, Windows Sockets 2 posts an **FD_ACCEPT** message immediately.

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

The **FD_OOB** event is used only when a socket is configured to receive OOB data separately. If the socket is configured to receive OOB data inline, the OOB (expedited) data is treated as normal data and the application should register an interest in, and will receive, **FD_READ** events, not **FD_OOB** events. An application can set or inspect the way in which OOB data is to be handled by using setsockopt or getsockopt for the SO_OOBINLINE option.

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

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

Be aware that the application will only receive an **FD_CLOSE** message to indicate closure of a virtual circuit, and only when all the received data has been read if this is a graceful close. It will not receive an **FD_READ** message to indicate this condition.

The **FD_QOS** or **FD_GROUP_QOS** message is posted when any parameter in the flow specification associated with socket *s* or the socket group that *s* belongs to has changed, respectively. Applications should use WSAIoctl with command SIO_GET_QOS or SIO_GET_GROUP_QOS to get the current quality of service for socket *s* or for the socket group *s* belongs to, respectively.

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

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

- **FD_READ**:
  1. When **WSAAsyncSelect** is called, if there is data currently available to receive.
  2. When data arrives, if **FD_READ** is not already posted.
  3. After recv or recvfrom is called, with or without MSG_PEEK), if data is still available to receive.

  > **Note** When setsockopt SO_OOBINLINE is enabled, data includes both normal data and OOB data in the instances noted above.

- **FD_WRITE**:
  1. When **WSAAsyncSelect** called, if a send or sendto is possible.
  2. After connect or accept called, when connection established.
  3. After send or sendto fail with WSAEWOULDBLOCK, when **send** or **sendto** are likely to succeed.
  4. After bind on a connectionless socket. **FD_WRITE** may or may not occur at this time (implementation-dependent). In any case, a connectionless socket is always writable immediately after a **bind** operation.
- **FD_OOB**: Only valid when setsockopt SO_OOBINLINE is disabled (default).
  1. When **WSAAsyncSelect** called, if there is OOB data currently available to receive with the MSG_OOB flag.
  2. When OOB data arrives, if **FD_OOB** not already posted.
  3. After recv or recvfrom called with or without MSG_OOB flag, if OOB data is still available to receive.
- **FD_ACCEPT**:
  1. When **WSAAsyncSelect** called, if there is currently a connection request available to accept.
  2. When a connection request arrives, if **FD_ACCEPT** not already posted.
  3. After accept called, if there is another connection request available to accept.
- **FD_CONNECT**:
  1. When **WSAAsyncSelect** called, if there is currently a connection established.
  2. After connect called, when connection is established, even when **connect** succeeds immediately, as is typical with a datagram socket.
  3. After calling WSAJoinLeaf, when join operation completes.
  4. After connect, WSAConnect, or WSAJoinLeaf was called with a nonblocking, connection-oriented socket. The initial operation returned with a specific error of WSAEWOULDBLOCK, but the network operation went ahead. Whether the operation eventually succeeds or not, when the outcome has been determined, **FD_CONNECT** happens. The client should check the error code to determine whether the outcome was successful or failed.
- **FD_CLOSE**: Only valid on connection-oriented sockets (for example, SOCK_STREAM)
  1. When **WSAAsyncSelect** called, if socket connection has been closed.
  2. After remote system initiated graceful close, when no data currently available to receive (Be aware that, if data has been received and is waiting to be read when the remote system initiates a graceful close, the **FD_CLOSE** is not delivered until all pending data has been read).
  3. After local system initiates graceful close with shutdown and remote system has responded with "End of Data" notification (for example, TCP FIN), when no data currently available to receive.
  4. When remote system terminates connection (for example, sent TCP RST), and *lParam* will contain WSAECONNRESET error value.

  > **Note** **FD_CLOSE** is not posted after closesocket is called.

- **FD_QOS**:
  1. When **WSAAsyncSelect** called, if the quality of service associated with the socket has been changed.
  2. After WSAIoctl with SIO_GET_QOS called, when the quality of service is changed.
- **FD_GROUP_QOS**: Reserved.
- **FD_ROUTING_INTERFACE_CHANGE**:
  - After WSAIoctl with SIO_ROUTING_INTERFACE_CHANGE called, when the local interface that should be used to reach the destination specified in the IOCTL changes.
- **FD_ADDRESS_LIST_CHANGE**:
  - After WSAIoctl with SIO_ADDRESS_LIST_CHANGE called, when the list of local addresses to which the application can bind changes.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winsock2.h (include Winsock2.h)
Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

WSAEventSelect

Winsock Functions

Winsock Reference

select

WSACancelAsyncRequest function (winsock2.h)

10/22/2022 • 2 minutes to read • Edit Online

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

Syntax

```
int WSAAPI WSACancelAsyncRequest(
  [in] HANDLE hAsyncTaskHandle
);
```

Parameters

`[in] hAsyncTaskHandle`

Handle that specifies the asynchronous operation to be canceled.

Return value

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

ERROR CODE	MEANING
WSANOTINITIALISED	A successful WSAStartup call must occur before using this function.
WSAENETDOWN	The network subsystem has failed.
WSAEINVAL	Indicates that the specified asynchronous task handle was invalid.
WSAEINPROGRESS	A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function.
WSAEALREADY	The asynchronous routine being canceled has already completed.

Note It is unclear whether the application can usefully distinguish between WSAEINVAL and WSAEALREADY, since in both cases the error indicates that there is no asynchronous operation in progress with the indicated handle. (Trivial exception: zero is always an invalid asynchronous task handle.) The Windows Sockets specification does not prescribe how a conformant Windows Sockets provider should distinguish between the two cases. For maximum portability, a Windows Sockets application should treat the two errors as equivalent.

Remarks

The **WSACancelAsyncRequest** function is used to cancel an asynchronous operation that was initiated by one of the **WSAAsyncGetXByY** functions such as WSAAsyncGetHostByName. The operation to be canceled is identified by the *hAsyncTaskHandle* parameter, which should be set to the asynchronous task handle as returned by the initiating **WSAAsyncGetXByY** function.

An attempt to cancel an existing asynchronous **WSAAsyncGetXByY** operation can fail with an error code of WSAEALREADY for two reasons. First, the original operation has already completed and the application has dealt with the resultant message. Second, the original operation has already completed but the resultant message is still waiting in the application window queue.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winsock2.h (include Winsock2.h)
Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

WSAAsyncGetHostByAddr

WSAAsyncGetHostByName

WSAAsyncGetProtoByName

WSAAsyncGetProtoByNumber

WSAAsyncGetServByName

WSAAsyncGetServByPort

Winsock Functions

Winsock Reference

WSACancelBlockingCall function (winsock2.h)

10/22/2022 • 2 minutes to read • Edit Online

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

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

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

Syntax

```
int WSAAPI WSACancelBlockingCall();
```

Return value

None

Requirements

Target Platform	Windows
Header	winsock2.h

See also

Winsock Functions

Winsock Reference

WSACleanup function (winsock2.h)

10/22/2022 • 3 minutes to read • Edit Online

The **WSACleanup** function terminates use of the Winsock 2 DLL (*Ws2_32.dll*).

Syntax

```
int WSAAPI WSACleanup();
```

Return value

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

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

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

Remarks

An application or DLL is required to perform a successful WSAStartup call before it can use Windows Sockets services. When it has completed the use of Windows Sockets, the application or DLL must call **WSACleanup** to deregister itself from a Windows Sockets implementation and allow the implementation to free any resources allocated on behalf of the application or DLL.

When **WSACleanup** is called, any pending blocking or asynchronous Windows Sockets calls issued by any thread in this process are canceled without posting any notification messages or without signaling any event objects. Any pending overlapped send or receive operations (WSASend, WSASendTo, WSARecv, or WSARecvFrom with an overlapped socket, for example) issued by any thread in this process are also canceled without setting the event object or invoking the completion routine, if one was specified. In this case, the pending overlapped operations fail with the error status **WSA_OPERATION_ABORTED**.

Sockets that were open when **WSACleanup** was called are reset and automatically deallocated as if closesocket were called. Sockets that have been closed with **closesocket** but that still have pending data to be sent can be affected when **WSACleanup** is called. In this case, the pending data can be lost if the WS2_32.DLL is unloaded from memory as the application exits. To ensure that all pending data is sent, an application should use shutdown to close the connection, then wait until the close completes before calling **closesocket** and **WSACleanup**. All resources and internal state, such as queued unposted or posted messages, must be deallocated so as to be available to the next user.

There must be a call to **WSACleanup** for each successful call to WSAStartup. Only the final **WSACleanup** function call performs the actual cleanup. The preceding calls simply decrement an internal reference count in the WS2_32.DLL.

> **Note** **WSACleanup** does not unregister names (peer names, for example) that may have been registered with a Windows Sockets namespace provider such as Peer Name Resolution Protocol (PNRP) namespace provider.

In Windows Sockets 1.1, attempting to call **WSACleanup** from within a blocking hook and then failing to check the return code was a common programming error. If a Winsock 1.1 application needs to quit while a blocking call is outstanding, the application has to first cancel the blocking call with WSACancelBlockingCall then issue the **WSACleanup** call once control has been returned to the application. In Windows Sockets 2, this issue does not exist and the **WSACancelBlockingCall** function has been removed.

The **WSACleanup** function typically leads to protocol-specific helper DLLs being unloaded. As a result, the **WSACleanup** function should not be called from the DllMain function in a application DLL. This can potentially cause deadlocks. For more information, please see the DLL Main Function.

Windows Phone 8: This function is supported for Windows Phone Store apps on Windows Phone 8 and later.

Windows 8.1 and **Windows Server 2012 R2**: This function is supported for Windows Store apps on Windows 8.1, Windows Server 2012 R2, and later.

Requirements

Minimum supported client	Windows 8.1, Windows Vista [desktop apps | UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps | UWP apps]
Target Platform	Windows
Header	winsock2.h (include Winsock2.h)
Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

PNRP Namespace Provider API

WSAStartup

Winsock Functions

Winsock Reference

closesocket

shutdown

WSACloseEvent function (winsock2.h)

10/22/2022 • 2 minutes to read • Edit Online

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

Syntax

```
BOOL WSAAPI WSACloseEvent(
  [in] WSAEVENT hEvent
);
```

Parameters

`[in] hEvent`

Object handle identifying the open event.

Return value

If the function succeeds, the return value is **TRUE**.

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

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

Remarks

The **WSACloseEvent** function closes the handle to an event object and frees resources associated with the event object. This function is used to close a handle created by the WSACreateEvent function. Once the handle to the event object is closed, further references to this handle will fail with the error WSA_INVALID_HANDLE.

Windows Phone 8: This function is supported for Windows Phone Store apps on Windows Phone 8 and later.

Windows 8.1 and **Windows Server 2012 R2**: This function is supported for Windows Store apps on Windows 8.1, Windows Server 2012 R2, and later.

Requirements

Minimum supported client	Windows 8.1, Windows Vista [desktop apps | UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps | UWP apps]
Target Platform	Windows
Header	winsock2.h
Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

WSACreateEvent

WSAEnumNetworkEvents

WSAEventSelect

WSAGetOverlappedResult

WSARecv

WSARecvFrom

WSAResetEvent

WSASend

WSASendTo

WSASetEvent

WSAWaitForMultipleEvents

Winsock Functions

Winsock Reference

WSACOMPLETION structure (winsock2.h)

10/22/2022 • 2 minutes to read • Edit Online

The **WSACOMPLETION** structure specifies completion notification settings for I/O control calls made to a registered namespace.

Syntax

```
typedef struct _WSACOMPLETION {
  WSACOMPLETIONTYPE Type;
  union {
    struct {
      HWND   hWnd;
      UINT   uMsg;
      WPARAM context;
    } WindowMessage;
    struct {
      LPWSAOVERLAPPED lpOverlapped;
    } Event;
    struct {
      LPWSAOVERLAPPED                    lpOverlapped;
      LPWSAOVERLAPPED_COMPLETION_ROUTINE lpfnCompletionProc;
    } Apc;
    struct {
      LPWSAOVERLAPPED lpOverlapped;
      HANDLE          hPort;
      ULONG_PTR       Key;
    } Port;
  } Parameters;
} WSACOMPLETION, *PWSACOMPLETION, *LPWSACOMPLETION;
```

Members

`Type`

Type: **WSACOMPLETIONTYPE**

The type of completion notification required. See Remarks.

`Parameters`

The parameters required to complete the callback. The structures within the Parameters union specify information required for completing the callback of each given type. For example, the **WindowMessage** structure must be filled when **Type** is set to NSP_NOTIFY_HWND.

`Parameters.WindowMessage`

`Parameters.WindowMessage.hWnd`

Type: HWND Windows handle.

`Parameters.WindowMessage.uMsg`

Type: UINT Message handle.

`Parameters.WindowMessage.context`

Type: WPARAM Context of the message or handle.

`Parameters.Event`

`Parameters.Event.lpOverlapped`

Type: LPWSAOVERLAPPED A pointer to a WSAOVERLAPPED structure.

`Parameters.Apc`

`Parameters.Apc.lpOverlapped`

Type: LPWSAOVERLAPPED A pointer to a WSAOVERLAPPED structure.

`Parameters.Apc.lpfnCompletionProc`

Type: _In_opt_ **LPWSAOVERLAPPED_COMPLETION_ROUTINE**

A pointer to an application-provided completion routine.

`Parameters.Port`

`Parameters.Port.lpOverlapped`

Type: LPWSAOVERLAPPED A pointer to a WSAOVERLAPPED structure.

`Parameters.Port.hPort`

Type: HANDLE A handle to the port.

`Parameters.Port.Key`

Type: ULONG_PTR A pointer to the key.

Remarks

The **WSACOMPLETION** structure enables callbacks to be provided in any of the following formats, based on the value provided in **Type**:

CALLBACK FORMAT	TYPE VALUE
Polling	NSP_NOTIFY_IMMEDIATELY
Window Message	NSP_NOTIFY_HWND
Event	NSP_NOTIFY_EVENT
APC	NSP_NOTIFY_APC
Completion Port	NSP_NOTIFY_PORT

For a blocking function, set the **WSACOMPLETION** structure to null.

Requirements

Minimum supported client	Windows XP [desktop apps only]

Minimum supported server	Windows Server 2003 [desktop apps only]
Header	winsock2.h

See also

WSANSPIoctl

WSAConnect function (winsock2.h)

10/22/2022 • 9 minutes to read • Edit Online

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

Syntax

```
int WSAAPI WSAConnect(
  [in]  SOCKET         s,
  [in]  const sockaddr *name,
  [in]  int            namelen,
  [in]  LPWSABUF       lpCallerData,
  [out] LPWSABUF       lpCalleeData,
  [in]  LPQOS          lpSQOS,
  [in]  LPQOS          lpGQOS
);
```

Parameters

`[in] s`

A descriptor identifying an unconnected socket.

`[in] name`

A pointer to a sockaddr structure that specifies the address to which to connect. For IPv4, the **sockaddr** contains **AF_INET** for the address family, the destination IPv4 address, and the destination port. For IPv6, the **sockaddr** structure contains **AF_INET6** for the address family, the destination IPv6 address, the destination port, and may contain additional flow and scope-id information.

`[in] namelen`

The length, in bytes, of the sockaddr structure pointed to by the *name* parameter.

`[in] lpCallerData`

A pointer to the user data that is to be transferred to the other socket during connection establishment. See Remarks.

`[out] lpCalleeData`

A pointer to the user data that is to be transferred back from the other socket during connection establishment. See Remarks.

`[in] lpSQOS`

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

`[in] lpGQOS`

Reserved for future use with socket groups. A pointer to the FLOWSPEC structures for the socket group (if applicable). This parameter should be **NULL**.

Return value

If no error occurs, **WSAConnect** returns zero. Otherwise, it returns SOCKET_ERROR, and a specific error code can be retrieved by calling WSAGetLastError. On a blocking socket, the return value indicates success or failure of the connection attempt.

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

- Use select to determine the completion of the connection request by checking if the socket is writable.
- If your application is using WSAAsyncSelect to indicate interest in connection events, then your application will receive an FD_CONNECT notification when the connect operation is complete(successful or not).
- If your application is using WSAEventSelect to indicate interest in connection events, then the associated event object will be signaled when the connect operation is complete (successful or not).

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

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

ERROR CODE	MEANING
WSANOTINITIALISED	A successful WSAStartup call must occur before using this function.
WSAENETDOWN	The network subsystem has failed.
WSAEADDRINUSE	The local address of the socket is already in use and the socket was not marked to allow address reuse with SO_REUSEADDR. This error usually occurs during the execution of bind, but could be delayed until this function if the **bind** function operates on a partially wildcard address (involving ADDR_ANY) and if a specific address needs to be "committed" at the time of this function.
WSAEINTR	The (blocking) Windows Socket 1.1 call was canceled through WSACancelBlockingCall.
WSAEINPROGRESS	A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function.
WSAEALREADY	A nonblocking connect or WSAConnect call is in progress on the specified socket.
WSAEADDRNOTAVAIL	The remote address is not a valid address (such as ADDR_ANY).
WSAEAFNOSUPPORT	Addresses in the specified family cannot be used with this socket.

WSAECONNREFUSED	The attempt to connect was rejected.
WSAEFAULT	The *name* or the *namelen* parameter is not a valid part of the user address space, the *namelen* parameter is too small, the buffer length for *lpCalleeData*, *lpSQOS*, and *lpGQOS* are too small, or the buffer length for *lpCallerData* is too large.
WSAEINVAL	The parameter *s* is a listening socket, or the destination address specified is not consistent with that of the constrained group to which the socket belongs, or the *lpGQOS* parameter is not **NULL**.
WSAEISCONN	The socket is already connected (connection-oriented sockets only).
WSAENETUNREACH	The network cannot be reached from this host at this time.
WSAEHOSTUNREACH	A socket operation was attempted to an unreachable host.
WSAENOBUFS	No buffer space is available. The socket cannot be connected.
WSAENOTSOCK	The descriptor is not a socket.
WSAEOPNOTSUPP	The FLOWSPEC structures specified in *lpSQOS* and *lpGQOS* cannot be satisfied.
WSAEPROTONOSUPPORT	The *lpCallerData* parameter is not supported by the service provider.
WSAETIMEDOUT	Attempt to connect timed out without establishing a connection.
WSAEWOULDBLOCK	The socket is marked as nonblocking and the connection cannot be completed immediately.
WSAEACCES	Attempt to connect datagram socket to broadcast address failed because setsockopt SO_BROADCAST is not enabled.

Remarks

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

For applications targeted to Windows Vista and later, consider using the WSAConnectByList or WSAConnectByName function which greatly simplify client application design.

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

> **Note** If a socket is opened, a setsockopt call is made, and then a sendto call is made, Windows Sockets performs an implicit bind function call.

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

For a connectionless socket (for example, type SOCK_DGRAM), the operation performed by **WSAConnect** is merely to establish a default destination address so that the socket can be used on subsequent connection-oriented send and receive operations (send, WSASend, recv, and WSARecv). Any datagrams received from an address other than the destination address specified will be discarded. If the entire name structure is all zeros (not just the address parameter of the name structure), then the socket will be disconnected. Then, the default remote address will be indeterminate, so **send**, **WSASend**, **recv**, and **WSARecv** calls will return the error code WSAENOTCONN. However, sendto, WSASendTo, recvfrom, and WSARecvFrom can still be used. The default destination can be changed by simply calling **WSAConnect** again, even if the socket is already connected. Any datagrams queued for receipt are discarded if *name* is different from the previous **WSAConnect**.

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

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

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

The *lpCallerData* parameter contains a pointer to any user data that is to be sent along with the connection request (called connect data). This is 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.

> **Note** Connect data is not supported by the TCP/IP protocol in Windows. Connect data is supported only on ATM (RAWWAN) over a raw socket.

If *lpCallerData* is **NULL**, no user data will be passed to the peer. The *lpCalleeData* is a result parameter that will contain any user data passed back from the other socket as part of the connection establishment in a WSABUF structure. The **len** member of the **WSABUF** structure pointed to by the *lpCalleeData* parameter initially contains the length of the buffer allocated by the application for the **buf** member of the **WSABUF** structure. The **len** member of the **WSABUF** structure pointed to by the *lpCalleeData* parameter will be set to zero if no user data has been passed back. The *lpCalleeData* information will be valid when the connection operation is complete. For blocking sockets, the connection operation completes when the **WSAConnect** function returns. For

nonblocking sockets, completion will be after the FD_CONNECT notification has occurred. If *lpCalleeData* is **NULL**, no user data will be passed back. The exact format of the user data is specific to the address family to which the socket belongs.

At connect time, an application can use the *lpSQOS* and *lpGQOS* parameter to override any previous quality of service specification made for the socket through WSAIoctl with either the SIO_SET_QOS or SIO_SET_GROUP_QOS opcode.

The *lpSQOS* parameter specifies the FLOWSPEC structures for socket *s*, one for each direction, followed by any additional provider-specific parameters. If either the associated transport provider in general or the specific type of socket in particular cannot honor the quality of service request, an error will be returned as indicated in the following. The sending or receiving flow specification values will be ignored, respectively, for any unidirectional sockets. If no provider-specific parameters are specified, the **buf** and **len** members of the WSABUF structure pointed to by the *lpCalleeData* parameter should be set to **NULL** and zero, respectively. A **NULL** value for *lpSQOS* parameter indicates no application-supplied quality of service.

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

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

> **Note** When issuing a blocking Winsock call such as **WSAConnect**, Winsock may need to wait for a network event before the call can complete. Winsock performs an alertable wait in this situation, which can be interrupted by an asynchronous procedure call (APC) scheduled on the same thread. Issuing another blocking Winsock call inside an APC that interrupted an ongoing blocking Winsock call on the same thread will lead to undefined behavior, and must never be attempted by Winsock clients.

Windows Phone 8: This function is supported for Windows Phone Store apps on Windows Phone 8 and later.

Windows 8.1 and **Windows Server 2012 R2**: This function is supported for Windows Store apps on Windows 8.1, Windows Server 2012 R2, and later.

Requirements

Minimum supported client	Windows 8.1, Windows Vista [desktop apps | UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps | UWP apps]
Target Platform	Windows
Header	winsock2.h
Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

WSAAsyncSelect

WSABUF

WSAConnect

ConnectEx

WSAConnectByList

WSAEventSelect

Winsock Functions

Winsock Reference

accept

bind

connect

getsockname

getsockopt

select

socket

WSAConnectByList function (winsock2.h)

10/22/2022 • 8 minutes to read • Edit Online

The **WSAConnectByList** function establishes a connection to one out of a collection of possible endpoints represented by a set of destination addresses (host names and ports). This function takes all the destination addresses passed to it and all of the local computer's source addresses, and tries connecting using all possible address combinations before giving up.

This function supports both IPv4 and IPv6 addresses.

Syntax

```
BOOL WSAConnectByList(
  [in]      SOCKET               s,
  [in]      PSOCKET_ADDRESS_LIST SocketAddress,
  [in, out] LPDWORD              LocalAddressLength,
  [out]     LPSOCKADDR           LocalAddress,
  [in, out] LPDWORD              RemoteAddressLength,
  [out]     LPSOCKADDR           RemoteAddress,
  [in]      const timeval        *timeout,
  [in]      LPWSAOVERLAPPED      Reserved
);
```

Parameters

`[in] s`

A descriptor that identifies an unbound and unconnected socket. Note that unlike other Winsock calls to establish a connection (for example, WSAConnect), the **WSAConnectByList** function requires an unbound socket.

`[in] SocketAddress`

A pointer to a SOCKET_ADDRESS_LIST structure that represents the possible destination address and port pairs to connect to a peer. It is the application's responsibility to fill in the port number in the each SOCKET_ADDRESS structure in the **SOCKET_ADDRESS_LIST**.

`[in, out] LocalAddressLength`

On input, a pointer to the size, in bytes, of the *LocalAddress* buffer provided by the caller. On output, a pointer to the size, in bytes, of the **SOCKADDR** for the local address stored in the *LocalAddress* buffer filled in by the system upon successful completion of the call.

`[out] LocalAddress`

A pointer to the **SOCKADDR** structure that receives the local address of the connection. The size of the parameter is exactly the size returned in *LocalAddressLength*. This is the same information that would be returned by the getsockname function. This parameter can be **NULL**, in which case, the *LocalAddressLength* parameter is ignored.

`[in, out] RemoteAddressLength`

On input, a pointer to the size, in bytes, of the *RemoteAddress* buffer provided by the caller. On output, a pointer to the size, in bytes, of the **SOCKADDR** for the remote address stored in *RemoteAddress* buffer filled-in by the

system upon successful completion of the call.

`[out] RemoteAddress`

A pointer to the **SOCKADDR** structure that receives the remote address of the connection. This is the same information that would be returned by the **getpeername** function. This parameter can be **NULL**, in which case, the *RemoteAddressLength* is ignored.

`[in] timeout`

The time, in milliseconds, to wait for a response from the remote application before aborting the call. This parameter can be **NULL** in which case **WSAConnectByList** will complete after either the connection is successfully established or after a connection was attempted and failed on all possible local-remote address pairs.

`[in] Reserved`

Reserved for future implementation. This parameter must be set to **NULL**.

Return value

If a connection is established, **WSAConnectByList** returns **TRUE** and *LocalAddress* and *RemoteAddress* parameters are filled in if these buffers were supplied by the caller.

If the call fails, **FALSE** is returned. WSAGetLastError can then be called to get extended error information.

RETURN CODE	DESCRIPTION
WSAEHOSTUNREACH	The host passed as the *nodename* parameter was unreachable.
WSAEINVAL	An invalid parameter was passed to the function. The *Reserved* parameter must be **NULL**.
WSAENOBUFS	Sufficient memory could not be allocated.
WSAENOTSOCK	An invalid socket was passed to the function. The *s* parameter must not be **INVALID_SOCKET** or **NULL**.
WSAETIMEDOUT	A response from the remote application was not received before the *timeout* parameter was exceeded.

Remarks

WSAConnectByList is similar to the WSAConnectByName function. Instead of taking a single host name and service name (port), **WSAConnectByList** takes a list of addresses (host addresses and ports) and connects to one of the addresses. The **WSAConnectByList** function is designed to support peer-to-peer collaboration scenarios where an application needs to connect to any available node out of a list of potential nodes. **WSAConnectByList** is compatible with both IPv6 and IPv4 versions.

The set of possible destinations, represented by a list of addresses, is provided by the caller. **WSAConnectByList** does more than simply attempt to connect to one of possibly many destination addresses. Specifically, the function takes all remote addresses passed in by the caller, all local addresses, and then attempts a connection first using address pairs with the highest chance of success. As such, **WSAConnectByList** not only

ensures that connection will be established if a connection is at all possible, but also minimizes the time to establish the connection.

The caller can specify the *LocalAddress* and *RemoteAddress* buffers and lengths to determine the local and remote addresses for which the connection was successfully established.

The *timeout* parameter allows the caller to limit the time spent by the function in establishing a connection. Internally, **WSAConnectByList** performs multiple operations (connection attempts). In between each operation, the *timeout* parameter is checked to see if the *timeout* has been exceeded and, if so, the call is aborted. Note that an individual operation (connect) will not be interrupted once the *timeout* is exceeded, so the **WSAConnectByList** call can take longer to time out than the value specified in the *timeout* parameter.

WSAConnectByList has limitations: It works only for connection-oriented sockets, such as those of type SOCK_STREAM. The function does not support overlapped I/O or non-blocking behavior. **WSAConnectByList** will block even if the socket is in non-blocking mode. **WSAConnectByList** will try connecting (one-by-one) to the various addresses provided by the caller. Potentially, each of these connection attempts may fail with a different error code. Since only a single error code can be returned, the value returned is the error code from the last connection attempt.

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 0:0:0:0:0:FFFF. The IPv4-mapped IPv6 address format is specified in RFC 4291. For more information, see 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.

The arrays of pointers passed in the *SocketAddressList* parameter point to an array of SOCKET_ADDRESS structures, which are a generic data type. The *RemoteAddress* and the *LocalAddress* parameters also point to **SOCKADDR** structures. When **WSAConnectByList** is called, it is expected that a socket address type specific to the network protocol or address family being used will actually be passed in these parameters. So for IPv4 addresses, a pointer to a **sockaddr_in** structure would be cast to a pointer to **SOCKADDR** when passed as a parameter. For IPv6 addresses, a pointer to a **sockaddr_in6** structure would be cast to a pointer to **SOCKADDR** when passed as a parameter. The *SocketAddressList* parameter can contain pointers to a mixture of IPv4 and IPv6 addresses. So some **SOCKET_ADDRESS** pointers can be to **sockaddr_in** structures and others can be to **sockaddr_in6** structures. If it is expected that IPv6 addresses can be used, then the *RemoteAddress* and *LocalAddress* parameters should point to **sockaddr_in6** structures and be cast to **SOCKADDR** structures. The *RemoteAddressLength* and the *LocalAddressLength* parameters must represent the length of these larger structures.

When the WSAConnectByList function returns **TRUE**, the socket *s* is in the default state for a connected socket. The socket *s* does not enable previously set properties or options until SO_UPDATE_CONNECT_CONTEXT is set on the socket. Use the setsockopt function to set the SO_UPDATE_CONNECT_CONTEXT option.

For example:

```
//Need to #include <mswsock.h> for SO_UPDATE_CONNECT_CONTEXT

int iResult = 0;

iResult = setsockopt( s, SOL_SOCKET, SO_UPDATE_CONNECT_CONTEXT, NULL, 0 );
```

Note When issuing a blocking Winsock call such as WSAConnectByList with the *timeout* parameter set to **NULL**, Winsock may need to wait for a network event before the call can complete. Winsock performs an alertable wait in this situation, which can be interrupted by an asynchronous procedure call (APC) scheduled on the same thread. Issuing another blocking Winsock call inside an APC that interrupted an ongoing blocking Winsock call on the same thread will lead to undefined behavior, and must never be attempted by Winsock clients.

Windows Phone 8: This function is supported for Windows Phone Store apps on Windows Phone 8 and later.

Windows 8.1 and **Windows Server 2012 R2**: This function is supported for Windows Store apps on Windows 8.1, Windows Server 2012 R2, and later.

Examples

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 = INVALID_SOCKET;

    int 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 = WSAConnectByList(ConnSocket,
            AddressList,
            &dwLocalAddr,
            (SOCKADDR*)&LocalAddr,
            &dwRemoteAddr,
            (SOCKADDR*)&RemoteAddr,
            NULL,
            NULL);
    if (bSuccess){
        return ConnSocket;
    } else {
        return INVALID_SOCKET;
    }
}
```

Requirements

Minimum supported client	Windows 8.1, Windows Vista [desktop apps | UWP apps]

Minimum supported server	Windows Server 2003 [desktop apps | UWP apps]
Target Platform	Windows
Header	winsock2.h
Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

IPPROTO_IPV6 Socket Options

SOCKADDR

SOCKET_ADDRESS

SOCKET_ADDRESS_LIST

WSAConnect

WSAConnectByName

WSAGetLastError

getaddrinfo

getpeername

getsockname

setsockopt

WSAConnectByNameA function (winsock2.h)

10/22/2022 • 7 minutes to read • Edit Online

The **WSAConnectByName** function establishes a connection to a specified host and port. This function is provided to allow a quick connection to a network endpoint given a host name and port.

This function supports both IPv4 and IPv6 addresses.

Syntax

```
BOOL WSAConnectByNameA(
  [in]      SOCKET          s,
  [in]      LPCSTR          nodename,
  [in]      LPCSTR          servicename,
  [in, out] LPDWORD         LocalAddressLength,
  [out]     LPSOCKADDR      LocalAddress,
  [in, out] LPDWORD         RemoteAddressLength,
  [out]     LPSOCKADDR      RemoteAddress,
  [in]      const timeval   *timeout,
            LPWSAOVERLAPPED Reserved
);
```

Parameters

`[in] s`

A descriptor that identifies an unconnected socket.

> **Note** On Windows 7, Windows Server 2008 R2, and earlier, the **WSAConnectByName** function requires an unbound and unconnected socket. This differs from other Winsock calls to establish a connection (for example, WSAConnect).

`[in] nodename`

A **NULL**-terminated string that contains the name of the host or the IP address of the host on which to connect for IPv4 or IPv6.

`[in] servicename`

A **NULL**-terminated string that contains the service name or destination port of the host on which to connect for IPv4 or IPv6.

A service name is a string alias for a port number. For example, "http" is an alias for port 80 defined by the Internet Engineering Task Force (IETF) as the default port used by web servers for the HTTP protocol. Possible values for the *servicename* parameter when a port number is not specified are listed in the following file:

`%WINDIR%\system32\drivers\etc\services`

`[in, out] LocalAddressLength`

On input, a pointer to the size, in bytes, of the *LocalAddress* buffer provided by the caller. On output, a pointer to the size, in bytes, of the **SOCKADDR** for the local address stored in the *LocalAddress* buffer filled in by the system upon successful completion of the call.

`[out] LocalAddress`

A pointer to the **SOCKADDR** structure that receives the local address of the connection. The size of the parameter is exactly the size returned in *LocalAddressLength*. This is the same information that would be returned by the getsockname function. This parameter can be **NULL**, in which case, the *LocalAddressLength* parameter is ignored.

`[in, out] RemoteAddressLength`

On input, a pointer to the size, in bytes, of the *RemoteAddress* buffer provided by the caller. On output, a pointer to the size, in bytes, of the **SOCKADDR** for the remote address stored in *RemoteAddress* buffer filled-in by the system upon successful completion of the call.

`[out] RemoteAddress`

A pointer to the **SOCKADDR** structure that receives the remote address of the connection. This is the same information that would be returned by the **getpeername** function. This parameter can be **NULL**, in which case, the *RemoteAddressLength* is ignored.

`[in] timeout`

The time, in milliseconds, to wait for a response from the remote application before aborting the call.

`Reserved`

Reserved for future implementation. This parameter must be set to **NULL**.

Return value

If a connection is established, **WSAConnectByName** returns **TRUE** and *LocalAddress* and *RemoteAddress* parameters are filled in if these buffers were supplied by the caller.

If the call fails, **FALSE** is returned. WSAGetLastError can then be called to get extended error information.

RETURN CODE	DESCRIPTION
WSAEHOSTUNREACH	The host passed as the *nodename* parameter was unreachable.
WSAEINVAL	An invalid parameter was passed to the function. The *nodename* or the *servicename* parameter must not be **NULL**. The *Reserved* parameter must be **NULL**.
WSAENOBUFS	Sufficient memory could not be allocated.
WSAENOTSOCK	An invalid socket was passed to the function. The *s* parameter must not be **INVALID_SOCKET** or **NULL**.
WSAETIMEDOUT	A response from the remote application was not received before the *timeout* parameter was exceeded.

Remarks

WSAConnectByName is provided to enable quick and transparent connections to remote hosts on specific ports. It is compatible with both IPv6 and IPv4 versions.

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 .

WSAConnectByName has limitations: It works only for connection-oriented sockets, such as those of type SOCK_STREAM. The function does not support overlapped I/O or non-blocking behavior. **WSAConnectByName** will block even if the socket is in non-blocking mode.

WSAConnectByName does not support user-provided data during the establishment of a connection. This call does not support FLOWSPEC structures, either. In cases where these features are required, WSAConnect must be used instead.

In versions before Windows 10, 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.

This restriction was removed Windows 10.

The *RemoteAddress* and the *LocalAddress* parameters point to a **SOCKADDR** structure, which is a generic data type. When **WSAConnectByName** is called, it is expected that a socket address type specific to the network protocol or address family being used will actually be passed in these parameters. So for IPv4 addresses, a pointer to a **sockaddr_in** structure would be cast to a pointer to **SOCKADDR** as the *RemoteAddress* and *LocalAddress* parameters. For IPv6 addresses, a pointer to a **sockaddr_in6** structure would be cast to a pointer to **SOCKADDR** as the *RemoteAddress* and *LocalAddress* parameters.

When the **WSAConnectByName** function returns **TRUE**, the socket *s* is in the default state for a connected socket. The socket *s* does not enable previously set properties or options until SO_UPDATE_CONNECT_CONTEXT is set on the socket. Use the setsockopt function to set the SO_UPDATE_CONNECT_CONTEXT option.

For example:

```
//Need to #include <mswsock.h> for SO_UPDATE_CONNECT_CONTEXT

int iResult = 0;

iResult = setsockopt( s, SOL_SOCKET, SO_UPDATE_CONNECT_CONTEXT, NULL, 0 );
```

> **Note** When issuing a blocking Winsock call such as **WSAConnectByName** with the *timeout* parameter set to **NULL**, Winsock may need to wait for a network event before the call can complete. Winsock performs an alertable wait in this situation, which can be interrupted by an asynchronous procedure call (APC) scheduled on the same thread. Issuing another blocking Winsock call inside an APC that interrupted an ongoing blocking Winsock call on the same thread will lead to undefined behavior, and must never be attempted by Winsock clients.

Windows Phone 8: The **WSAConnectByNameW** function is supported for Windows Phone Store apps on Windows Phone 8 and later.

Windows 8.1 and **Windows Server 2012 R2**: The **WSAConnectByNameW** function is supported for Windows Store apps on Windows 8.1, Windows Server 2012 R2, and later.

Examples

Establish a connection using **WSAConnectByName**.

```
#ifndef UNICODE
#define UNICODE
#endif

#define WIN32_LEAN_AND_MEAN

#include <winsock2.h>
#include <Ws2tcpip.h>
#include <mswsock.h>   // Need for SO_UPDATE_CONNECT_CONTEXT
#include <stdio.h>

// Link with ws2_32.lib
#pragma comment(lib, "Ws2_32.lib")

SOCKET
OpenAndConnect(LPWSTR NodeName, LPWSTR PortName)
{
    SOCKET ConnSocket = INVALID_SOCKET;
    int 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){
        wprintf(L"socket failed with error: %d\n", WSAGetLastError());
        return INVALID_SOCKET;
    }

    iResult = setsockopt(ConnSocket, IPPROTO_IPV6,
        IPV6_V6ONLY, (char*)&ipv6only, sizeof(ipv6only) );
    if (iResult == SOCKET_ERROR){
        wprintf(L"setsockopt for IPV6_V6ONLY failed with error: %d\n",
            WSAGetLastError());
        closesocket(ConnSocket);
        return INVALID_SOCKET;
    }

    bSuccess = WSAConnectByName(ConnSocket, NodeName,
            PortName, &dwLocalAddr,
            (SOCKADDR*)&LocalAddr,
            &dwRemoteAddr,
            (SOCKADDR*)&RemoteAddr,
            NULL,
            NULL);
    if (!bSuccess){
        wprintf(L"WsaConnectByName failed with error: %d\n", WSAGetLastError());
        closesocket(ConnSocket);
        return INVALID_SOCKET;


    }

    iResult = setsockopt(ConnSocket, SOL_SOCKET,
        SO_UPDATE_CONNECT_CONTEXT, NULL, 0);
    if (iResult == SOCKET_ERROR){
        wprintf(L"setsockopt for SO_UPDATE_CONNECT_CONTEXT failed with error: %d\n",
            WSAGetLastError());
        closesocket(ConnSocket);
        return INVALID_SOCKET;
    }

    return ConnSocket;
```

```
}

int __cdecl wmain(int argc, wchar_t **argv)
{
   //-----------------------------------------
    // Declare and initialize variables
    WSADATA wsaData;
    int iResult;

    SOCKET s = INVALID_SOCKET;

    // Validate the parameters
    if (argc != 3) {
        wprintf(L"usage: %ws <Nodename> <Portname>\n", argv[0]);
        wprintf(L"wsaconnectbyname establishes a connection to a specified host and port.\n");
        wprintf(L"%ws www.contoso.com 8080\n", argv[0]);
        return 1;
    }

    // Initialize Winsock
    iResult = WSAStartup(MAKEWORD(2, 2), &wsaData);
    if (iResult != 0) {
        wprintf(L"WSAStartup failed: %d\n", iResult);
        return 1;
    }

    wprintf(L"WsaConnectByName with following parameters:\n");
    wprintf(L"\tNodename = %ws\n", argv[1]);
    wprintf(L"\tPortname (or port) = %ws\n\n", argv[2]);

    //--------------------------------
    // Call our function that uses the WsaConnectByName.

    s = OpenAndConnect(argv[1], argv[2]);
    if ( s == INVALID_SOCKET ) {
        wprintf(L"WsaConnectByName failed with error: %d\n", WSAGetLastError());
        WSACleanup();
        return 1;
    }
    else
    {
        wprintf(L"WsaConnectByName succeeded\n");

        closesocket(s);
        WSACleanup();
        return 0;
    }
}
```

NOTE

The winsock2.h header defines WSAConnectByName as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see Conventions for Function Prototypes.

Requirements

Minimum supported client	Windows 8.1, Windows Vista [desktop apps | UWP apps]

Minimum supported server	Windows Server 2008 [desktop apps | UWP apps]
Target Platform	Windows
Header	winsock2.h
Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

IPPROTO_IPV6 Socket Options

SOCKADDR

WSAConnect

WSAConnectByList

WSAGetLastError

getaddrinfo

getpeername

getsockname

setsockopt

WSAConnectByNameW function (winsock2.h)

10/22/2022 • 7 minutes to read • Edit Online

The **WSAConnectByName** function establishes a connection to a specified host and port. This function is provided to allow a quick connection to a network endpoint given a host name and port.

This function supports both IPv4 and IPv6 addresses.

Syntax

```
BOOL WSAConnectByNameW(
  [in]      SOCKET          s,
  [in]      LPWSTR          nodename,
  [in]      LPWSTR          servicename,
  [in, out] LPDWORD         LocalAddressLength,
  [out]     LPSOCKADDR      LocalAddress,
  [in, out] LPDWORD         RemoteAddressLength,
  [out]     LPSOCKADDR      RemoteAddress,
  [in]      const timeval   *timeout,
            LPWSAOVERLAPPED Reserved
);
```

Parameters

`[in] s`

A descriptor that identifies an unconnected socket.

> **Note** On Windows 7, Windows Server 2008 R2, and earlier, the **WSAConnectByName** function requires an unbound and unconnected socket. This differs from other Winsock calls to establish a connection (for example, WSAConnect).

`[in] nodename`

A **NULL**-terminated string that contains the name of the host or the IP address of the host on which to connect for IPv4 or IPv6.

`[in] servicename`

A **NULL**-terminated string that contains the service name or destination port of the host on which to connect for IPv4 or IPv6.

A service name is a string alias for a port number. For example, "http" is an alias for port 80 defined by the Internet Engineering Task Force (IETF) as the default port used by web servers for the HTTP protocol. Possible values for the *servicename* parameter when a port number is not specified are listed in the following file:

`%WINDIR%\system32\drivers\etc\services`

`[in, out] LocalAddressLength`

On input, a pointer to the size, in bytes, of the *LocalAddress* buffer provided by the caller. On output, a pointer to the size, in bytes, of the **SOCKADDR** for the local address stored in the *LocalAddress* buffer filled in by the system upon successful completion of the call.

`[out] LocalAddress`

A pointer to the **SOCKADDR** structure that receives the local address of the connection. The size of the parameter is exactly the size returned in *LocalAddressLength*. This is the same information that would be returned by the getsockname function. This parameter can be **NULL**, in which case, the *LocalAddressLength* parameter is ignored.

`[in, out] RemoteAddressLength`

On input, a pointer to the size, in bytes, of the *RemoteAddress* buffer provided by the caller. On output, a pointer to the size, in bytes, of the **SOCKADDR** for the remote address stored in *RemoteAddress* buffer filled-in by the system upon successful completion of the call.

`[out] RemoteAddress`

A pointer to the **SOCKADDR** structure that receives the remote address of the connection. This is the same information that would be returned by the **getpeername** function. This parameter can be **NULL**, in which case, the *RemoteAddressLength* is ignored.

`[in] timeout`

The time, in milliseconds, to wait for a response from the remote application before aborting the call.

`Reserved`

Reserved for future implementation. This parameter must be set to **NULL**.

Return value

If a connection is established, **WSAConnectByName** returns **TRUE** and *LocalAddress* and *RemoteAddress* parameters are filled in if these buffers were supplied by the caller.

If the call fails, **FALSE** is returned. WSAGetLastError can then be called to get extended error information.

RETURN CODE	DESCRIPTION
WSAEHOSTUNREACH	The host passed as the *nodename* parameter was unreachable.
WSAEINVAL	An invalid parameter was passed to the function. The *nodename* or the *servicename* parameter must not be **NULL**. The *Reserved* parameter must be **NULL**.
WSAENOBUFS	Sufficient memory could not be allocated.
WSAENOTSOCK	An invalid socket was passed to the function. The *s* parameter must not be **INVALID_SOCKET** or **NULL**.
WSAETIMEDOUT	A response from the remote application was not received before the *timeout* parameter was exceeded.

Remarks

WSAConnectByName is provided to enable quick and transparent connections to remote hosts on specific ports. It is compatible with both IPv6 and IPv4 versions.

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 .

WSAConnectByName has limitations: It works only for connection-oriented sockets, such as those of type SOCK_STREAM. The function does not support overlapped I/O or non-blocking behavior. **WSAConnectByName** will block even if the socket is in non-blocking mode.

WSAConnectByName does not support user-provided data during the establishment of a connection. This call does not support FLOWSPEC structures, either. In cases where these features are required, WSAConnect must be used instead.

In versions before Windows 10, 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.

This restriction was removed Windows 10.

The *RemoteAddress* and the *LocalAddress* parameters point to a **SOCKADDR** structure, which is a generic data type. When **WSAConnectByName** is called, it is expected that a socket address type specific to the network protocol or address family being used will actually be passed in these parameters. So for IPv4 addresses, a pointer to a **sockaddr_in** structure would be cast to a pointer to **SOCKADDR** as the *RemoteAddress* and *LocalAddress* parameters. For IPv6 addresses, a pointer to a **sockaddr_in6** structure would be cast to a pointer to **SOCKADDR** as the *RemoteAddress* and *LocalAddress* parameters.

When the **WSAConnectByName** function returns **TRUE**, the socket *s* is in the default state for a connected socket. The socket *s* does not enable previously set properties or options until SO_UPDATE_CONNECT_CONTEXT is set on the socket. Use the setsockopt function to set the SO_UPDATE_CONNECT_CONTEXT option.

For example:

```
//Need to #include <mswsock.h> for SO_UPDATE_CONNECT_CONTEXT

int iResult = 0;

iResult = setsockopt( s, SOL_SOCKET, SO_UPDATE_CONNECT_CONTEXT, NULL, 0 );
```

> **Note** When issuing a blocking Winsock call such as **WSAConnectByName** with the *timeout* parameter set to **NULL**, Winsock may need to wait for a network event before the call can complete. Winsock performs an alertable wait in this situation, which can be interrupted by an asynchronous procedure call (APC) scheduled on the same thread. Issuing another blocking Winsock call inside an APC that interrupted an ongoing blocking Winsock call on the same thread will lead to undefined behavior, and must never be attempted by Winsock clients.

Windows Phone 8: The **WSAConnectByNameW** function is supported for Windows Phone Store apps on Windows Phone 8 and later.

Windows 8.1 and **Windows Server 2012 R2**: The **WSAConnectByNameW** function is supported for Windows Store apps on Windows 8.1, Windows Server 2012 R2, and later.

Examples

Establish a connection using **WSAConnectByName**.

```
#ifndef UNICODE
#define UNICODE
#endif

#define WIN32_LEAN_AND_MEAN

#include <winsock2.h>
#include <Ws2tcpip.h>
#include <mswsock.h>   // Need for SO_UPDATE_CONNECT_CONTEXT
#include <stdio.h>

// Link with ws2_32.lib
#pragma comment(lib, "Ws2_32.lib")

SOCKET
OpenAndConnect(LPWSTR NodeName, LPWSTR PortName)
{
    SOCKET ConnSocket = INVALID_SOCKET;
    int 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){
        wprintf(L"socket failed with error: %d\n", WSAGetLastError());
        return INVALID_SOCKET;
    }

    iResult = setsockopt(ConnSocket, IPPROTO_IPV6,
        IPV6_V6ONLY, (char*)&ipv6only, sizeof(ipv6only) );
    if (iResult == SOCKET_ERROR){
        wprintf(L"setsockopt for IPV6_V6ONLY failed with error: %d\n",
            WSAGetLastError());
        closesocket(ConnSocket);
        return INVALID_SOCKET;
    }

    bSuccess = WSAConnectByName(ConnSocket, NodeName,
            PortName, &dwLocalAddr,
            (SOCKADDR*)&LocalAddr,
            &dwRemoteAddr,
            (SOCKADDR*)&RemoteAddr,
            NULL,
            NULL);
    if (!bSuccess){
        wprintf(L"WsaConnectByName failed with error: %d\n", WSAGetLastError());
        closesocket(ConnSocket);
        return INVALID_SOCKET;


    }

    iResult = setsockopt(ConnSocket, SOL_SOCKET,
        SO_UPDATE_CONNECT_CONTEXT, NULL, 0);
    if (iResult == SOCKET_ERROR){
        wprintf(L"setsockopt for SO_UPDATE_CONNECT_CONTEXT failed with error: %d\n",
            WSAGetLastError());
        closesocket(ConnSocket);
        return INVALID_SOCKET;
    }

    return ConnSocket;
```

```
}

int __cdecl wmain(int argc, wchar_t **argv)
{
   //-----------------------------------------
    // Declare and initialize variables
    WSADATA wsaData;
    int iResult;

    SOCKET s = INVALID_SOCKET;

    // Validate the parameters
    if (argc != 3) {
        wprintf(L"usage: %ws <Nodename> <Portname>\n", argv[0]);
        wprintf(L"wsaconnectbyname establishes a connection to a specified host and port.\n");
        wprintf(L"%ws www.contoso.com 8080\n", argv[0]);
        return 1;
    }

    // Initialize Winsock
    iResult = WSAStartup(MAKEWORD(2, 2), &wsaData);
    if (iResult != 0) {
        wprintf(L"WSAStartup failed: %d\n", iResult);
        return 1;
    }

    wprintf(L"WsaConnectByName with following parameters:\n");
    wprintf(L"\tNodename = %ws\n", argv[1]);
    wprintf(L"\tPortname (or port) = %ws\n\n", argv[2]);

    //--------------------------------
    // Call our function that uses the WsaConnectByName.

    s = OpenAndConnect(argv[1], argv[2]);
    if ( s == INVALID_SOCKET ) {
        wprintf(L"WsaConnectByName failed with error: %d\n", WSAGetLastError());
        WSACleanup();
        return 1;
    }
    else
    {
        wprintf(L"WsaConnectByName succeeded\n");

        closesocket(s);
        WSACleanup();
        return 0;
    }
}
```

NOTE

The winsock2.h header defines WSAConnectByName as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see Conventions for Function Prototypes.

Requirements

Minimum supported client	Windows 8.1, Windows Vista [desktop apps | UWP apps]

Minimum supported server	Windows Server 2008 [desktop apps | UWP apps]
Target Platform	Windows
Header	winsock2.h
Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

IPPROTO_IPV6 Socket Options

SOCKADDR

WSAConnect

WSAConnectByList

WSAGetLastError

getaddrinfo

getpeername

getsockname

setsockopt

WSACreateEvent function (winsock2.h)

10/22/2022 • 2 minutes to read • Edit Online

The **WSACreateEvent** function creates a new event object.

Syntax

```
WSAEVENT WSAAPI WSACreateEvent();
```

Return value

If no error occurs, **WSACreateEvent** returns the handle of the event object. Otherwise, the return value is WSA_INVALID_EVENT. To get extended error information, call WSAGetLastError.

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

Remarks

The **WSACreateEvent** function creates a manual-reset event object with an initial state of nonsignaled. The handle of the event object returned cannot be inherited by child processes. The event object is unnamed.

The WSASetEvent function can be called to set the state of the event object to signaled. The WSAResetEvent function can be called to set the state of the event object to nonsignaled. When an event object is no longer needed, the WSACloseEvent function should be called to free the resources associated with the event object.

Windows Sockets 2 event objects are system objects in Windows environments. Therefore, if a Windows application wants to use an auto-reset event rather than a manual-reset event, the application can call the CreateEvent function directly. The scope of an event object is limited to the process in which it is created.

Windows Phone 8: This function is supported for Windows Phone Store apps on Windows Phone 8 and later.

Windows 8.1 and **Windows Server 2012 R2**: This function is supported for Windows Store apps on Windows 8.1, Windows Server 2012 R2, and later.

Requirements

Minimum supported client	Windows 8.1, Windows Vista [desktop apps | UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps | UWP apps]
Target Platform	Windows
Header	winsock2.h
Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

CreateEvent

WSACloseEvent

WSAEnumNetworkEvents

WSAEventSelect

WSAGetOverlappedResult

WSARecv

WSARecvFrom

WSAResetEvent

WSASend

WSASendTo

WSASetEvent

WSAWaitForMultipleEvents

Winsock Functions

Winsock Reference

WSADATA structure (winsock2.h)

10/22/2022 • 4 minutes to read • Edit Online

The **WSADATA** structure contains information about the Windows Sockets implementation.

Syntax

```
typedef struct WSAData {
  WORD           wVersion;
  WORD           wHighVersion;
#if ...
  unsigned short iMaxSockets;
#if ...
  unsigned short iMaxUdpDg;
#if ...
  char           *lpVendorInfo;
#if ...
  char           szDescription[WSADESCRIPTION_LEN + 1];
#if ...
  char           szSystemStatus[WSASYS_STATUS_LEN + 1];
#else
  char           szDescription[WSADESCRIPTION_LEN + 1];
#endif
#else
  char           szSystemStatus[WSASYS_STATUS_LEN + 1];
#endif
#else
  unsigned short iMaxSockets;
#endif
#else
  unsigned short iMaxUdpDg;
#endif
#else
  char           *lpVendorInfo;
#endif
} WSADATA, *LPWSADATA;
```

Members

`wVersion`

Type: **WORD**

The version of the Windows Sockets specification that the *Ws2_32.dll* expects the caller to use. The high-order byte specifies the minor version number; the low-order byte specifies the major version number.

`wHighVersion`

Type: **WORD**

The highest version of the Windows Sockets specification that the *Ws2_32.dll* can support. The high-order byte specifies the minor version number; the low-order byte specifies the major version number.

This is the same value as the **wVersion** member when the version requested in the *wVersionRequested* parameter passed to the WSAStartup function is the highest version of the Windows Sockets specification that the *Ws2_32.dll* can support.

`iMaxSockets`

Type: **unsigned short**

The maximum number of sockets that may be opened. This member should be ignored for Windows Sockets version 2 and later.

The **iMaxSockets** member is retained for compatibility with Windows Sockets specification 1.1, but should not be used when developing new applications. No single value can be appropriate for all underlying service providers. The architecture of Windows Sockets changed in version 2 to support multiple providers, and the **WSADATA** structure no longer applies to a single vendor's stack.

`iMaxUdpDg`

Type: **unsigned short**

The maximum datagram message size. This member is ignored for Windows Sockets version 2 and later.

The **iMaxUdpDg** member is retained for compatibility with Windows Sockets specification 1.1, but should not be used when developing new applications. The architecture of Windows Sockets changed in version 2 to support multiple providers, and the **WSADATA** structure no longer applies to a single vendor's stack. For the actual maximum message size specific to a particular Windows Sockets service provider and socket type, applications should use getsockopt to retrieve the value of option SO_MAX_MSG_SIZE after a socket has been created.

`lpVendorInfo`

Type: **char FAR***

A pointer to vendor-specific information. This member should be ignored for Windows Sockets version 2 and later.

The **lpVendorInfo** member is retained for compatibility with Windows Sockets specification 1.1. The architecture of Windows Sockets changed in version 2 to support multiple providers, and the **WSADATA** structure no longer applies to a single vendor's stack. Applications needing to access vendor-specific configuration information should use getsockopt to retrieve the value of option PVD_CONFIG for vendor-specific information.

`szDescription[WSADESCRIPTION_LEN + 1]`

Type: **char[WSADESCRIPTION_LEN+1]**

A **NULL**-terminated ASCII string into which the *Ws2_32.dll* copies a description of the Windows Sockets implementation. The text (up to 256 characters in length) can contain any characters except control and formatting characters. The most likely use that an application would have for this member is to display it (possibly truncated) in a status message.

`szSystemStatus[WSASYS_STATUS_LEN + 1]`

Type: **char[WSASYS_STATUS_LEN+1]**

A **NULL**-terminated ASCII string into which the *Ws2_32.dll* copies relevant status or configuration information. The *Ws2_32.dll* should use this parameter only if the information might be useful to the user or support staff. This member should not be considered as an extension of the **szDescription** parameter.

Remarks

The WSAStartup function initiates the use of the Windows Sockets DLL by a process. The **WSAStartup** function returns a pointer to the

WSADATA structure in the *lpWSAData* parameter.

The current version of the Windows Sockets specification returned in the **wHighVersion** member of the **WSADATA** structure is version 2.2 encoded with the major version number in the low-byte and the minor version number in the high-byte. This version of the current Winsock DLL, *Ws2_32.dll*, supports applications that request any of the following versions of the Windows Sockets specification:

- 1.0
- 1.1
- 2.0
- 2.1
- 2.2

Depending on the version requested by the application, one of the above version numbers is the value encoded as the major version number in the low-byte and the minor version number in the high-byte that is returned in the **wVersion** member of the **WSADATA** structure.

> **Note** An application should ignore the **iMaxsockets**, **iMaxUdpDg**, and **lpVendorInfo** members in **WSADATA** if the value in **wVersion** after a successful call to WSAStartup is at least 2. This is because the architecture of Windows Sockets changed in version 2 to support multiple providers, and **WSADATA** no longer applies to a single vendor's stack. Two new socket options are introduced to supply provider-specific information: SO_MAX_MSG_SIZE (replaces the **iMaxUdpDg** member) and PVD_CONFIG (allows any other provider-specific configuration to occur).

Examples

The following example demonstrates the use of the **WSADATA** structure.

```
WORD wVersionRequested;
WSADATA wsaData;
int err;

wVersionRequested = MAKEWORD( 2, 2 );

err = WSAStartup( wVersionRequested, &wsaData );
if ( err != 0 ) {
    /* Tell the user that we could not find a usable */
    /* WinSock DLL.                                  */
    return;
}

/* Confirm that the WinSock DLL supports 2.2.*/
/* Note that if the DLL supports versions greater    */
/* than 2.2 in addition to 2.2, it will still return */
/* 2.2 in wVersion since that is the version we      */
/* requested.                                        */

if ( LOBYTE( wsaData.wVersion ) != 2 ||
        HIBYTE( wsaData.wVersion ) != 2 ) {
    /* Tell the user that we could not find a usable */
    /* WinSock DLL.                                  */
    WSACleanup( );
    return;
}

/* The WinSock DLL is acceptable. Proceed. */
```

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	winsock2.h (include Winsock2.h)

See also

SOL_SOCKET Socket Options

Socket Options and IOCTLs

WSAStartup

getsockopt

WSADuplicateSocketA function (winsock2.h)

10/22/2022 • 3 minutes to read • Edit Online

The **WSADuplicateSocket** function returns a WSAPROTOCOL_INFO structure that can be used to create a new socket descriptor for a shared socket. The **WSADuplicateSocket** function cannot be used on a QOS-enabled socket.

Syntax

```
int WSAAPI WSADuplicateSocketA(
  [in]  SOCKET              s,
  [in]  DWORD               dwProcessId,
  [out] LPWSAPROTOCOL_INFOA lpProtocolInfo
);
```

Parameters

`[in] s`

Descriptor identifying the local socket.

`[in] dwProcessId`

Process identifier of the target process in which the duplicated socket will be used.

`[out] lpProtocolInfo`

Pointer to a buffer, allocated by the client, that is large enough to contain a WSAPROTOCOL_INFO structure. The service provider copies the protocol information structure contents to this buffer.

Return value

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

ERROR CODE	MEANING
WSANOTINITIALISED	A successful WSAStartup call must occur before using this function.
WSAENETDOWN	The network subsystem has failed.
WSAEINVAL	Indicates that one of the specified parameters was invalid.
WSAEINPROGRESS	A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function.

WSAEMFILE	No more socket descriptors are available.
WSAENOBUFS	No buffer space is available. The socket cannot be created.
WSAENOTSOCK	The descriptor is not a socket.
WSAEFAULT	The *lpProtocolInfo* parameter is not a valid part of the user address space.

Remarks

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

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

One possible scenario for establishing and handing off a shared socket is illustrated in the following table.

SOURCE PROCESS	IPC	DESTINATION PROCESS
1) WSASocket, WSAConnect		
2) Request target process identifier	==>	
		3) Receive process identifier request and respond
4) Receive process identifier	<==	
5) Call **WSADuplicateSocket** to get a special WSAPROTOCOL_INFO structure		
6) Send WSAPROTOCOL_INFO structure to target		
	==>	7) Receive WSAPROTOCOL_INFO structure
		8) Call WSASocket to create shared socket descriptor.
		9) Use shared socket for data exchange
10) closesocket	<==	

The descriptors that reference a shared socket can be used independently for I/O. However, the Windows Sockets interface does not implement any type of access control, so it is up to the processes involved to coordinate their operations on a shared socket. Shared sockets are typically used to having one process that is responsible for creating sockets and establishing connections, and other processes that are responsible for information exchange.

All of the state information associated with a socket is held in common across all the descriptors because the socket descriptors are duplicated and not the actual socket. For example, a setsockopt operation performed using one descriptor is subsequently visible using a getsockopt from any or all descriptors. Both the source process and the destination process should pass the same flags to their respective WSASocket function calls. If the source process uses the socket function to create the socket, the destination process must pass the **WSA_FLAG_OVERLAPPED** flag to its **WSASocket** function call. A process can call closesocket on a duplicated socket and the descriptor will become deallocated. The underlying socket, however, will remain open until **closesocket** is called by the last remaining descriptor.

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

Windows 8.1 and **Windows Server 2012 R2**: The **WSADuplicateSocketW** function is supported for Windows Store apps on Windows 8.1, Windows Server 2012 R2, and later.

> **NOTE**
>
> The winsock2.h header defines WSADuplicateSocket as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see Conventions for Function Prototypes.

Requirements

Minimum supported client	Windows 8.1, Windows Vista [desktop apps | UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps | UWP apps]
Target Platform	Windows
Header	winsock2.h
Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

WSASocket

Winsock Functions

WSADuplicateSocketW function (winsock2.h)

10/22/2022 • 3 minutes to read • Edit Online

The **WSADuplicateSocket** function returns a WSAPROTOCOL_INFO structure that can be used to create a new socket descriptor for a shared socket. The **WSADuplicateSocket** function cannot be used on a QOS-enabled socket.

Syntax

```
int WSAAPI WSADuplicateSocketW(
  [in]  SOCKET              s,
  [in]  DWORD               dwProcessId,
  [out] LPWSAPROTOCOL_INFOW lpProtocolInfo
);
```

Parameters

`[in] s`

Descriptor identifying the local socket.

`[in] dwProcessId`

Process identifier of the target process in which the duplicated socket will be used.

`[out] lpProtocolInfo`

Pointer to a buffer, allocated by the client, that is large enough to contain a WSAPROTOCOL_INFO structure. The service provider copies the protocol information structure contents to this buffer.

Return value

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

ERROR CODE	MEANING
WSANOTINITIALISED	A successful WSAStartup call must occur before using this function.
WSAENETDOWN	The network subsystem has failed.
WSAEINVAL	Indicates that one of the specified parameters was invalid.
WSAEINPROGRESS	A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function.

WSAEMFILE	No more socket descriptors are available.
WSAENOBUFS	No buffer space is available. The socket cannot be created.
WSAENOTSOCK	The descriptor is not a socket.
WSAEFAULT	The *lpProtocolInfo* parameter is not a valid part of the user address space.

Remarks

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

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

One possible scenario for establishing and handing off a shared socket is illustrated in the following table.

SOURCE PROCESS	IPC	DESTINATION PROCESS
1) WSASocket, WSAConnect		
2) Request target process identifier	==>	
		3) Receive process identifier request and respond
4) Receive process identifier	<==	
5) Call **WSADuplicateSocket** to get a special WSAPROTOCOL_INFO structure		
6) Send WSAPROTOCOL_INFO structure to target		
	==>	7) Receive WSAPROTOCOL_INFO structure
		8) Call WSASocket to create shared socket descriptor.
		9) Use shared socket for data exchange
10) closesocket	<==	

The descriptors that reference a shared socket can be used independently for I/O. However, the Windows Sockets interface does not implement any type of access control, so it is up to the processes involved to coordinate their operations on a shared socket. Shared sockets are typically used to having one process that is responsible for creating sockets and establishing connections, and other processes that are responsible for information exchange.

All of the state information associated with a socket is held in common across all the descriptors because the socket descriptors are duplicated and not the actual socket. For example, a setsockopt operation performed using one descriptor is subsequently visible using a getsockopt from any or all descriptors. Both the source process and the destination process should pass the same flags to their respective WSASocket function calls. If the source process uses the socket function to create the socket, the destination process must pass the **WSA_FLAG_OVERLAPPED** flag to its **WSASocket** function call. A process can call closesocket on a duplicated socket and the descriptor will become deallocated. The underlying socket, however, will remain open until **closesocket** is called by the last remaining descriptor.

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

Windows 8.1 and **Windows Server 2012 R2**: The **WSADuplicateSocketW** function is supported for Windows Store apps on Windows 8.1, Windows Server 2012 R2, and later.

> **NOTE**
>
> The winsock2.h header defines WSADuplicateSocket as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see Conventions for Function Prototypes.

Requirements

Minimum supported client	Windows 8.1, Windows Vista [desktop apps | UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps | UWP apps]
Target Platform	Windows
Header	winsock2.h
Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

WSASocket

Winsock Functions

WSAECOMPARATOR enumeration (winsock2.h)

10/22/2022 • 2 minutes to read • Edit Online

The Windows Sockets **WSAECOMPARATOR** enumeration type is used for version-comparison semantics in Windows Sockets 2.

Syntax

```
typedef enum _WSAEcomparator {
  COMP_EQUAL = 0,
  COMP_NOTLESS
} WSAECOMPARATOR, *PWSAECOMPARATOR, *LPWSAECOMPARATOR;
```

Constants

`COMP_EQUAL` Value: *0* Used for determining whether version values are equal.
`COMP_NOTLESS` Used for determining whether a version value is no less than a specified value.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	winsock2.h

WSAEnumNameSpaceProvidersA function (winsock2.h)

10/22/2022 • 4 minutes to read • Edit Online

The **WSAEnumNameSpaceProviders** function retrieves information on available namespace providers.

Syntax

```
INT WSAAPI WSAEnumNameSpaceProvidersA(
  [in, out] LPDWORD              lpdwBufferLength,
  [out]     LPWSANAMESPACE_INFOA lpnspBuffer
);
```

Parameters

`[in, out] lpdwBufferLength`

On input, the number of bytes contained in the buffer pointed to by *lpnspBuffer*. On output (if the function fails, and the error is WSAEFAULT), the minimum number of bytes to pass for the *lpnspBuffer* to retrieve all the requested information. The buffer passed to **WSAEnumNameSpaceProviders** must be sufficient to hold all of the namespace information.

`[out] lpnspBuffer`

A buffer that is filled with WSANAMESPACE_INFO structures. The returned structures are located consecutively at the head of the buffer. Variable sized information referenced by pointers in the structures point to locations within the buffer located between the end of the fixed sized structures and the end of the buffer. The number of structures filled in is the return value of **WSAEnumNameSpaceProviders**.

Return value

The **WSAEnumNameSpaceProviders** function returns the number of WSANAMESPACE_INFO structures copied into *lpnspBuffer*. Otherwise, the value SOCKET_ERROR is returned, and a specific error number can be retrieved by calling WSAGetLastError.

ERROR CODE	MEANING
WSAEFAULT	The *lpnspBuffer* parameter was a **NULL** pointer or the buffer length, *lpdwBufferLength*, was too small to receive all the relevant WSANAMESPACE_INFO structures and associated information. When this error is returned, the buffer length required is returned in the *lpdwBufferLength* parameter.
WSANOTINITIALISED	The WS2_32.DLL has not been initialized. The application must first call WSAStartup before calling any Windows Sockets functions.
WSA_NOT_ENOUGH_MEMORY	There was insufficient memory to perform the operation.

Remarks

The **WSAEnumNameSpaceProviders** function returns information on available namespace providers in the buffer pointed to by the *lpnspBuffer* parameter. The returned buffer contains an array of WSANAMESPACE_INFO structures located consecutively at the head of the buffer. Variable sized information referenced by pointers in the **WSANAMESPACE_INFO** structures point to locations within the buffer located between the end of the fixed **WSANAMESPACE_INFO** structures and the end of the buffer. The number of **WSANAMESPACE_INFO** structures filled in is returned by the **WSAEnumNameSpaceProviders** function.

Each WSANAMESPACE_INFO structure entry contains the provider-specific information on the namespace entry passed to the WSCInstallNameSpace and WSCInstallNameSpace32 functions when the namespace provider was installed.

The WSAEnumNameSpaceProvidersEx function is an enhanced version of the **WSAEnumNameSpaceProviders** function. The WSCEnumNameSpaceProvidersEx32 function is an enhanced version of the **WSAEnumNameSpaceProviders** function that returns information on available 32-bit namespace providers for use on 64-bit platforms.

Example Code

The following example demonstrates the use of the **WSAEnumNameSpaceProviders** function to retrieve information on available namespace providers.

```
#ifndef UNICODE
#define UNICODE 1
#endif

#include <winsock2.h>
#include <ws2tcpip.h>
#include <objbase.h>
#include <stdio.h>

// Link with ws2_32.lib and ole32.lib
#pragma comment (lib, "Ws2_32.lib")
#pragma comment (lib, "ole32.lib")

#define MALLOC(x) HeapAlloc(GetProcessHeap(), 0, (x))
#define FREE(x) HeapFree(GetProcessHeap(), 0, (x))
// Note: could also use malloc() and free()

int wmain()
{

    //-----------------------------------------
    // Declare and initialize variables
    WSADATA wsaData;
    int iResult;
    int iError = 0;

    INT iNuminfo = 0;

    int i;


    // Allocate a 4K buffer to retrieve all the namespace providers
    DWORD dwInitialBufferLen = 4096;
    DWORD dwBufferLen;
    LPWSANAMESPACE_INFO lpProviderInfo;


    // variables needed for converting provider GUID to a string
    int iRet = 0;
    WCHAR GuidString[40] = {0};
```

```
    // Set dwBufferLen to the initial buffer length
    dwBufferLen = dwInitialBufferLen;

    // Initialize Winsock
    iResult = WSAStartup(MAKEWORD(2, 2), &wsaData);
    if (iResult != 0) {
        wprintf(L"WSAStartup failed: %d\n", iResult);
        return 1;
    }

    lpProviderInfo = (LPWSANAMESPACE_INFO) MALLOC(dwBufferLen);
    if (lpProviderInfo == NULL) {
        wprintf(L"Memory allocation for providers buffer failed\n");
        WSACleanup();
        return 1;
    }

    iNuminfo = WSAEnumNameSpaceProviders(&dwBufferLen, lpProviderInfo);
    if (iNuminfo == SOCKET_ERROR) {
        iError = WSAGetLastError();
        if (iError == WSAEFAULT && dwBufferLen != dwInitialBufferLen) {
            wprintf(L"WSAEnumNameSpaceProviders failed with too small a buffer\n");
            wprintf(L"  Increasing the buffer to %u\n\n", dwBufferLen);
            if (lpProviderInfo) {
               FREE(lpProviderInfo);
               lpProviderInfo = NULL;
            }

            lpProviderInfo = (LPWSANAMESPACE_INFO) MALLOC(dwBufferLen);
            if (lpProviderInfo == NULL) {
                wprintf(L"Memory allocation for providers buffer failed\n");
                WSACleanup();
               return 1;
            }

            iNuminfo = WSAEnumNameSpaceProviders(&dwBufferLen, lpProviderInfo);
            if (iNuminfo == SOCKET_ERROR) {
               wprintf(L"WSAEnumNameSpaceProviders failed with error: %d\n",
                  WSAGetLastError() );
               if (lpProviderInfo) {
                  FREE(lpProviderInfo);
                  lpProviderInfo = NULL;
               }
               WSACleanup();
               return 1;
            }
            else
                wprintf(L"\n");
        }
        else {
            wprintf(L"WSAEnumNameSpaceProviders failed with error: %d\n",
               WSAGetLastError() );
            if (lpProviderInfo) {
               FREE(lpProviderInfo);
               lpProviderInfo = NULL;
            }
           WSACleanup();
           return 1;
        }
     }

       wprintf(L"WSAEnumNameSpaceProviders succeeded with provider data count = %d\n\n",
           iNuminfo);
       for (i= 0; i < iNuminfo; i++) {
            iRet = StringFromGUID2(lpProviderInfo[i].NSProviderId, (LPOLESTR) &GuidString, 39);
            if (iRet == 0)
                wprintf(L"StringFromGUID2 failed\n");
            else
```

```
                wprintf(L"NameSpace ProviderId[%u] = %ws\n",i, GuidString);

            wprintf(L"NameSpace[%u] = ", i);
            switch (lpProviderInfo[i].dwNameSpace) {
            case NS_DNS:
                wprintf(L"Domain Name System (NS_DNS)\n");
                break;
            case NS_WINS:
                wprintf(L"Windows Internet Naming Service (NS_WINS)\n");
                break;
            case NS_NETBT:
                wprintf(L"NetBIOS (NS_NETBT)\n");
                break;
            case NS_NTDS:
                wprintf(L"Windows NT Directory Services (NS_NTDS)\n");
                break;
            case NS_NLA:
                wprintf(L"Network Location Awareness (NS_NLA)\n");
                break;
            // following values only defined on Vista and later
#if(_WIN32_WINNT >= 0x0600)
            case NS_BTH:
                wprintf(L"Bluetooth (NS_BTH)\n");
                break;
            case NS_EMAIL:
                wprintf(L"Email (NS_EMAIL)\n");
                break;
            case NS_PNRPNAME:
                wprintf(L"Peer-to-peer (NS_PNRPNAME)\n");
                break;
            case NS_PNRPCLOUD:
                wprintf(L"Peer-to-peer collection (NS_PNRPCLOUD)\n");
                break;
#endif
            default:
                wprintf(L"Other value (%u)\n", lpProviderInfo[i].dwNameSpace);
                break;
            }

            if (lpProviderInfo[i].fActive)
                wprintf(L"Namespace[%u] is active\n", i);
            else
                wprintf(L"Namespace[%u] is inactive\n", i);

            wprintf(L"NameSpace Version[%u] = %u\n", i, lpProviderInfo[i].dwVersion);

            wprintf(L"Namespace Identifier[%u] = %ws\n\n", i, lpProviderInfo[i].lpszIdentifier);
        }

    if (lpProviderInfo) {
        FREE(lpProviderInfo);
        lpProviderInfo = NULL;
    }
    WSACleanup();

    return 0;
}
```

Windows Phone 8: The **WSAEnumNameSpaceProvidersW** function is supported for Windows Phone Store apps on Windows Phone 8 and later.

Windows 8.1 and **Windows Server 2012 R2**: The **WSAEnumNameSpaceProvidersW** function is supported for Windows Store apps on Windows 8.1, Windows Server 2012 R2, and later.

> **NOTE**
> The winsock2.h header defines WSAEnumNameSpaceProviders as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see Conventions for Function Prototypes.

Requirements

Minimum supported client	Windows 8.1, Windows Vista [desktop apps | UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps | UWP apps]
Target Platform	Windows
Header	winsock2.h
Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

WSAEnumNameSpaceProvidersEx

WSAGetLastError

WSANAMESPACE_INFO

WSAStartup

WSCEnumNameSpaceProvidersEx32

WSCInstallNameSpace

WSCInstallNameSpaceEx

Winsock Functions

Winsock Reference

WSAEnumNameSpaceProvidersExA function (winsock2.h)

10/22/2022 • 2 minutes to read • Edit Online

The **WSAEnumNameSpaceProvidersEx** function retrieves information on available namespace providers.

Syntax

```
INT WSAAPI WSAEnumNameSpaceProvidersExA(
  [in, out] LPDWORD                lpdwBufferLength,
  [out]     LPWSANAMESPACE_INFOEXA lpnspBuffer
);
```

Parameters

`[in, out] lpdwBufferLength`

On input, the number of bytes contained in the buffer pointed to by *lpnspBuffer*. On output (if the function fails, and the error is WSAEFAULT), the minimum number of bytes to allocate for the *lpnspBuffer* buffer to allow it to retrieve all the requested information. The buffer passed to **WSAEnumNameSpaceProvidersEx** must be sufficient to hold all of the namespace information.

`[out] lpnspBuffer`

A buffer that is filled with WSANAMESPACE_INFOEX structures. The returned structures are located consecutively at the head of the buffer. Variable sized information referenced by pointers in the structures point to locations within the buffer located between the end of the fixed sized structures and the end of the buffer. The number of structures filled in is the return value of **WSAEnumNameSpaceProvidersEx**.

Return value

The **WSAEnumNameSpaceProvidersEx** function returns the number of WSANAMESPACE_INFOEX structures copied into *lpnspBuffer*. Otherwise, the value SOCKET_ERROR is returned, and a specific error number can be retrieved by calling WSAGetLastError.

ERROR CODE	MEANING
WSAEFAULT	The *lpnspBuffer* parameter was a **NULL** pointer or the buffer length, *lpdwBufferLength*, was too small to receive all the relevant WSANAMESPACE_INFOEX structures and associated information. When this error is returned, the buffer length required is returned in the *lpdwBufferLength* parameter.
WSANOTINITIALISED	The WS2_32.DLL has not been initialized. The application must first call WSAStartup before calling any Windows Sockets functions.
WSA_NOT_ENOUGH_MEMORY	There was insufficient memory to perform the operation.

Remarks

The **WSAEnumNameSpaceProvidersEx** function is an enhanced version of the WSAEnumNameSpaceProviders function. The provider-specific data blob associated with the namespace entry passed in the *lpProviderInfo* parameter to the WSCInstallNameSpaceEx function can be queried using **WSAEnumNameSpaceProvidersEx** function.

Currently, the only namespace provider included with Windows that sets information in the **ProviderSpecific** member of the WSANAMESPACE_INFOEX structure is the NS_EMAIL provider. The format of the **ProviderSpecific** member for an NS_EMAIL namespace provider is a NAPI_PROVIDER_INSTALLATION_BLOB structure.

When UNICODE or _UNICODE is defined, **WSAEnumNameSpaceProvidersEx** is defined to **WSAEnumNameSpaceProvidersExW**, the Unicode version of this function. The *lpnspBuffer* parameter is defined to the LPSAWSANAMESPACE_INFOEXW data type and **WSANAMESPACE_INFOEXW** structures are returned on success.

When UNICODE or _UNICODE is not defined, **WSAEnumNameSpaceProvidersEx** is defined to **WSAEnumNameSpaceProvidersExA**, the ANSI version of this function. The *lpnspBuffer* parameter is defined to the LPSAWSANAMESPACE_INFOEXA data type and **WSANAMESPACE_INFOEXA** structures are returned on success.

Windows 8.1 and **Windows Server 2012 R2**: The **WSAEnumNameSpaceProvidersExW** function is supported for Windows Store apps on Windows 8.1, Windows Server 2012 R2, and later.

> **NOTE**
>
> The winsock2.h header defines WSAEnumNameSpaceProvidersEx as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see Conventions for Function Prototypes.

Requirements

Minimum supported client	Windows 8.1, Windows Vista [desktop apps | UWP apps]
Minimum supported server	Windows Server 2008 [desktop apps | UWP apps]
Target Platform	Windows
Header	winsock2.h
Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

NAPI_PROVIDER_INSTALLATION_BLOB

WSAEnumNameSpaceProviders

WSANAMESPACE_INFOEX

WSCEnumNameSpaceProvidersEx32

WSCInstallNameSpaceEx32

WSAEnumNameSpaceProvidersExW function (winsock2.h)

10/22/2022 • 2 minutes to read • Edit Online

The **WSAEnumNameSpaceProvidersEx** function retrieves information on available namespace providers.

Syntax

```
INT WSAAPI WSAEnumNameSpaceProvidersExW(
  [in, out] LPDWORD                lpdwBufferLength,
  [out]     LPWSANAMESPACE_INFOEXW lpnspBuffer
);
```

Parameters

`[in, out] lpdwBufferLength`

On input, the number of bytes contained in the buffer pointed to by *lpnspBuffer*. On output (if the function fails, and the error is WSAEFAULT), the minimum number of bytes to allocate for the *lpnspBuffer* buffer to allow it to retrieve all the requested information. The buffer passed to **WSAEnumNameSpaceProvidersEx** must be sufficient to hold all of the namespace information.

`[out] lpnspBuffer`

A buffer that is filled with WSANAMESPACE_INFOEX structures. The returned structures are located consecutively at the head of the buffer. Variable sized information referenced by pointers in the structures point to locations within the buffer located between the end of the fixed sized structures and the end of the buffer. The number of structures filled in is the return value of **WSAEnumNameSpaceProvidersEx**.

Return value

The **WSAEnumNameSpaceProvidersEx** function returns the number of WSANAMESPACE_INFOEX structures copied into *lpnspBuffer*. Otherwise, the value SOCKET_ERROR is returned, and a specific error number can be retrieved by calling WSAGetLastError.

ERROR CODE	MEANING
WSAEFAULT	The *lpnspBuffer* parameter was a **NULL** pointer or the buffer length, *lpdwBufferLength*, was too small to receive all the relevant WSANAMESPACE_INFOEX structures and associated information. When this error is returned, the buffer length required is returned in the *lpdwBufferLength* parameter.
WSANOTINITIALISED	The WS2_32.DLL has not been initialized. The application must first call WSAStartup before calling any Windows Sockets functions.
WSA_NOT_ENOUGH_MEMORY	There was insufficient memory to perform the operation.

Remarks

The **WSAEnumNameSpaceProvidersEx** function is an enhanced version of the WSAEnumNameSpaceProviders function. The provider-specific data blob associated with the namespace entry passed in the *lpProviderInfo* parameter to the WSCInstallNameSpaceEx function can be queried using **WSAEnumNameSpaceProvidersEx** function.

Currently, the only namespace provider included with Windows that sets information in the **ProviderSpecific** member of the WSANAMESPACE_INFOEX structure is the NS_EMAIL provider. The format of the **ProviderSpecific** member for an NS_EMAIL namespace provider is a NAPI_PROVIDER_INSTALLATION_BLOB structure.

When UNICODE or _UNICODE is defined, **WSAEnumNameSpaceProvidersEx** is defined to **WSAEnumNameSpaceProvidersExW**, the Unicode version of this function. The *lpnspBuffer* parameter is defined to the LPSAWSANAMESPACE_INFOEXW data type and **WSANAMESPACE_INFOEXW** structures are returned on success.

When UNICODE or _UNICODE is not defined, **WSAEnumNameSpaceProvidersEx** is defined to **WSAEnumNameSpaceProvidersExA**, the ANSI version of this function. The *lpnspBuffer* parameter is defined to the LPSAWSANAMESPACE_INFOEXA data type and **WSANAMESPACE_INFOEXA** structures are returned on success.

Windows 8.1 and **Windows Server 2012 R2**: The **WSAEnumNameSpaceProvidersExW** function is supported for Windows Store apps on Windows 8.1, Windows Server 2012 R2, and later.

> **NOTE**
>
> The winsock2.h header defines WSAEnumNameSpaceProvidersEx as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see Conventions for Function Prototypes.

Requirements

Minimum supported client	Windows 8.1, Windows Vista [desktop apps | UWP apps]
Minimum supported server	Windows Server 2008 [desktop apps | UWP apps]
Target Platform	Windows
Header	winsock2.h
Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

NAPI_PROVIDER_INSTALLATION_BLOB

WSAEnumNameSpaceProviders

WSANAMESPACE_INFOEX

WSCEnumNameSpaceProvidersEx32

WSCInstallNameSpaceEx32

WSAEnumNameSpaceProvidersW function (winsock2.h)

10/22/2022 • 4 minutes to read • Edit Online

The **WSAEnumNameSpaceProviders** function retrieves information on available namespace providers.

Syntax

```
INT WSAAPI WSAEnumNameSpaceProvidersW(
  [in, out] LPDWORD              lpdwBufferLength,
  [out]     LPWSANAMESPACE_INFOW lpnspBuffer
);
```

Parameters

`[in, out] lpdwBufferLength`

On input, the number of bytes contained in the buffer pointed to by *lpnspBuffer*. On output (if the function fails, and the error is WSAEFAULT), the minimum number of bytes to pass for the *lpnspBuffer* to retrieve all the requested information. The buffer passed to **WSAEnumNameSpaceProviders** must be sufficient to hold all of the namespace information.

`[out] lpnspBuffer`

A buffer that is filled with WSANAMESPACE_INFO structures. The returned structures are located consecutively at the head of the buffer. Variable sized information referenced by pointers in the structures point to locations within the buffer located between the end of the fixed sized structures and the end of the buffer. The number of structures filled in is the return value of **WSAEnumNameSpaceProviders**.

Return value

The **WSAEnumNameSpaceProviders** function returns the number of WSANAMESPACE_INFO structures copied into *lpnspBuffer*. Otherwise, the value SOCKET_ERROR is returned, and a specific error number can be retrieved by calling WSAGetLastError.

ERROR CODE	MEANING
WSAEFAULT	The *lpnspBuffer* parameter was a **NULL** pointer or the buffer length, *lpdwBufferLength*, was too small to receive all the relevant WSANAMESPACE_INFO structures and associated information. When this error is returned, the buffer length required is returned in the *lpdwBufferLength* parameter.
WSANOTINITIALISED	The WS2_32.DLL has not been initialized. The application must first call WSAStartup before calling any Windows Sockets functions.
WSA_NOT_ENOUGH_MEMORY	There was insufficient memory to perform the operation.

Remarks

The **WSAEnumNameSpaceProviders** function returns information on available namespace providers in the buffer pointed to by the *lpnspBuffer* parameter. The returned buffer contains an array of WSANAMESPACE_INFO structures located consecutively at the head of the buffer. Variable sized information referenced by pointers in the **WSANAMESPACE_INFO** structures point to locations within the buffer located between the end of the fixed **WSANAMESPACE_INFO** structures and the end of the buffer. The number of **WSANAMESPACE_INFO** structures filled in is returned by the **WSAEnumNameSpaceProviders** function.

Each WSANAMESPACE_INFO structure entry contains the provider-specific information on the namespace entry passed to the WSCInstallNameSpace and WSCInstallNameSpace32 functions when the namespace provider was installed.

The WSAEnumNameSpaceProvidersEx function is an enhanced version of the **WSAEnumNameSpaceProviders** function. The WSCEnumNameSpaceProvidersEx32 function is an enhanced version of the **WSAEnumNameSpaceProviders** function that returns information on available 32-bit namespace providers for use on 64-bit platforms.

Example Code

The following example demonstrates the use of the **WSAEnumNameSpaceProviders** function to retrieve information on available namespace providers.

```
#ifndef UNICODE
#define UNICODE 1
#endif

#include <winsock2.h>
#include <ws2tcpip.h>
#include <objbase.h>
#include <stdio.h>

// Link with ws2_32.lib and ole32.lib
#pragma comment (lib, "Ws2_32.lib")
#pragma comment (lib, "ole32.lib")

#define MALLOC(x) HeapAlloc(GetProcessHeap(), 0, (x))
#define FREE(x) HeapFree(GetProcessHeap(), 0, (x))
// Note: could also use malloc() and free()

int wmain()
{

    //-----------------------------------------
    // Declare and initialize variables
    WSADATA wsaData;
    int iResult;
    int iError = 0;

    INT iNuminfo = 0;

    int i;


    // Allocate a 4K buffer to retrieve all the namespace providers
    DWORD dwInitialBufferLen = 4096;
    DWORD dwBufferLen;
    LPWSANAMESPACE_INFO lpProviderInfo;


    // variables needed for converting provider GUID to a string
    int iRet = 0;
    WCHAR GuidString[40] = {0};
```

```
    // Set dwBufferLen to the initial buffer length
    dwBufferLen = dwInitialBufferLen;

    // Initialize Winsock
    iResult = WSAStartup(MAKEWORD(2, 2), &wsaData);
    if (iResult != 0) {
        wprintf(L"WSAStartup failed: %d\n", iResult);
        return 1;
    }

    lpProviderInfo = (LPWSANAMESPACE_INFO) MALLOC(dwBufferLen);
    if (lpProviderInfo == NULL) {
        wprintf(L"Memory allocation for providers buffer failed\n");
        WSACleanup();
        return 1;
    }

    iNuminfo = WSAEnumNameSpaceProviders(&dwBufferLen, lpProviderInfo);
    if (iNuminfo == SOCKET_ERROR) {
        iError = WSAGetLastError();
        if (iError == WSAEFAULT && dwBufferLen != dwInitialBufferLen) {
            wprintf(L"WSAEnumNameSpaceProviders failed with too small a buffer\n");
            wprintf(L"  Increasing the buffer to %u\n\n", dwBufferLen);
            if (lpProviderInfo) {
               FREE(lpProviderInfo);
               lpProviderInfo = NULL;
            }

            lpProviderInfo = (LPWSANAMESPACE_INFO) MALLOC(dwBufferLen);
            if (lpProviderInfo == NULL) {
                wprintf(L"Memory allocation for providers buffer failed\n");
                WSACleanup();
               return 1;
            }

            iNuminfo = WSAEnumNameSpaceProviders(&dwBufferLen, lpProviderInfo);
            if (iNuminfo == SOCKET_ERROR) {
               wprintf(L"WSAEnumNameSpaceProviders failed with error: %d\n",
                  WSAGetLastError() );
               if (lpProviderInfo) {
                  FREE(lpProviderInfo);
                  lpProviderInfo = NULL;
               }
               WSACleanup();
               return 1;
            }
            else
                wprintf(L"\n");
        }
        else {
            wprintf(L"WSAEnumNameSpaceProviders failed with error: %d\n",
               WSAGetLastError() );
            if (lpProviderInfo) {
               FREE(lpProviderInfo);
               lpProviderInfo = NULL;
            }
           WSACleanup();
           return 1;
        }
     }

       wprintf(L"WSAEnumNameSpaceProviders succeeded with provider data count = %d\n\n",
           iNuminfo);
       for (i= 0; i < iNuminfo; i++) {
            iRet = StringFromGUID2(lpProviderInfo[i].NSProviderId, (LPOLESTR) &GuidString, 39);
            if (iRet == 0)
                wprintf(L"StringFromGUID2 failed\n");
            else
```

```
                wprintf(L"NameSpace ProviderId[%u] = %ws\n",i, GuidString);

            wprintf(L"NameSpace[%u] = ", i);
            switch (lpProviderInfo[i].dwNameSpace) {
            case NS_DNS:
                wprintf(L"Domain Name System (NS_DNS)\n");
                break;
            case NS_WINS:
                wprintf(L"Windows Internet Naming Service (NS_WINS)\n");
                break;
            case NS_NETBT:
                wprintf(L"NetBIOS (NS_NETBT)\n");
                break;
            case NS_NTDS:
                wprintf(L"Windows NT Directory Services (NS_NTDS)\n");
                break;
            case NS_NLA:
                wprintf(L"Network Location Awareness (NS_NLA)\n");
                break;
            // following values only defined on Vista and later
#if(_WIN32_WINNT >= 0x0600)
            case NS_BTH:
                wprintf(L"Bluetooth (NS_BTH)\n");
                break;
            case NS_EMAIL:
                wprintf(L"Email (NS_EMAIL)\n");
                break;
            case NS_PNRPNAME:
                wprintf(L"Peer-to-peer (NS_PNRPNAME)\n");
                break;
            case NS_PNRPCLOUD:
                wprintf(L"Peer-to-peer collection (NS_PNRPCLOUD)\n");
                break;
#endif
            default:
                wprintf(L"Other value (%u)\n", lpProviderInfo[i].dwNameSpace);
                break;
            }

            if (lpProviderInfo[i].fActive)
                wprintf(L"Namespace[%u] is active\n", i);
            else
                wprintf(L"Namespace[%u] is inactive\n", i);

            wprintf(L"NameSpace Version[%u] = %u\n", i, lpProviderInfo[i].dwVersion);

            wprintf(L"Namespace Identifier[%u] = %ws\n\n", i, lpProviderInfo[i].lpszIdentifier);
        }

    if (lpProviderInfo) {
        FREE(lpProviderInfo);
        lpProviderInfo = NULL;
    }
    WSACleanup();

    return 0;
}
```

Windows Phone 8: The **WSAEnumNameSpaceProvidersW** function is supported for Windows Phone Store apps on Windows Phone 8 and later.

Windows 8.1 and **Windows Server 2012 R2**: The **WSAEnumNameSpaceProvidersW** function is supported for Windows Store apps on Windows 8.1, Windows Server 2012 R2, and later.

> **NOTE**
> The winsock2.h header defines WSAEnumNameSpaceProviders as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see Conventions for Function Prototypes.

Requirements

Minimum supported client	Windows 8.1, Windows Vista [desktop apps | UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps | UWP apps]
Target Platform	Windows
Header	winsock2.h
Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

WSAEnumNameSpaceProvidersEx

WSAGetLastError

WSANAMESPACE_INFO

WSAStartup

WSCEnumNameSpaceProvidersEx32

WSCInstallNameSpace

WSCInstallNameSpaceEx

Winsock Functions

Winsock Reference

WSAEnumNetworkEvents function (winsock2.h)

10/22/2022 • 5 minutes to read • Edit Online

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

Syntax

```
int WSAAPI WSAEnumNetworkEvents(
  [in]  SOCKET             s,
  [in]  WSAEVENT           hEventObject,
  [out] LPWSANETWORKEVENTS lpNetworkEvents
);
```

Parameters

`[in] s`

A descriptor identifying the socket.

`[in] hEventObject`

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

`[out] lpNetworkEvents`

A pointer to a WSANETWORKEVENTS structure that is filled with a record of network events that occurred and any associated error codes.

Return value

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

ERROR CODE	MEANING
WSANOTINITIALISED	A successful WSAStartup call must occur before using this function.
WSAENETDOWN	The network subsystem has failed.
WSAEINVAL	One of the specified parameters was invalid.
WSAEINPROGRESS	A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function.
WSAENOTSOCK	The descriptor is not a socket.

WSAEFAULT	The *lpNetworkEvents* parameter is not a valid part of the user address space.

Remarks

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

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

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

The **lNetworkEvents** member of the WSANETWORKEVENTS structure indicates which of the FD_XXX network events have occurred. The **iErrorCode** array is used to contain any associated error codes with the array index corresponding to the position of event bits in **lNetworkEvents**. Identifiers such as FD_READ_BIT and FD_WRITE_BIT can be used to index the **iErrorCode** array. Note that only those elements of the **iErrorCode** array are set that correspond to the bits set in *lNetworkEvents* parameter. Other parameters are not modified (this is important for backward compatibility with the applications that are not aware of new FD_ROUTING_INTERFACE_CHANGE and FD_ADDRESS_LIST_CHANGE events).

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

Event: FD_CONNECT

ERROR CODE	MEANING
WSAEAFNOSUPPORT	Addresses in the specified family cannot be used with this socket.
WSAECONNREFUSED	The attempt to connect was forcefully rejected.
WSAENETUNREACH	The network cannot be reached from this host at this time.
WSAENOBUFS	No buffer space is available. The socket cannot be connected.
WSAETIMEDOUT	An attempt to connect timed out without establishing a connection

Event: FD_CLOSE

ERROR CODE	MEANING

WSAENETDOWN	The network subsystem has failed.
WSAECONNRESET	The connection was reset by the remote side.
WSAECONNABORTED	The connection was terminated due to a time-out or other failure.

Event: FD_ACCEPT

Event: FD_ADDRESS_LIST_CHANGE

Event: FD_GROUP_QOS

Event: FD_QOS

Event: FD_OOB

Event: FD_READ

Event: FD_WRITE

ERROR CODE	MEANING
WSAENETDOWN	The network subsystem has failed.

Event: FD_ROUTING_INTERFACE_CHANGE

ERROR CODE	MEANING
WSAENETUNREACH	The specified destination is no longer reachable.
WSAENETDOWN	The network subsystem has failed.

Example Code

The following example demonstrates the use of the WSAEnumNetworkEvents function.

```
#ifndef UNICODE
#define UNICODE
#endif

#define WIN32_LEAN_AND_MEAN

#include <windows.h>

#include <winsock2.h>
#include <Ws2tcpip.h>
#include <stdio.h>

// Link with ws2_32.lib
#pragma comment(lib, "Ws2_32.lib")

int main()
{
//-------------------------
// Declare and initialize variables
    WSADATA wsaData;
```

```
    int iResult;

    SOCKET SocketArray[WSA_MAXIMUM_WAIT_EVENTS], ListenSocket;
    WSAEVENT EventArray[WSA_MAXIMUM_WAIT_EVENTS];
    WSANETWORKEVENTS NetworkEvents;
    sockaddr_in InetAddr;
    DWORD EventTotal = 0;
    DWORD Index;
    DWORD i;

    HANDLE NewEvent = NULL;

    // Initialize Winsock
    iResult = WSAStartup(MAKEWORD(2, 2), &wsaData);
    if (iResult != 0) {
        wprintf(L"WSAStartup failed with error: %d\n", iResult);
        return 1;
    }

//-------------------------
// Create a listening socket
    ListenSocket = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
    if (ListenSocket == INVALID_SOCKET) {
        wprintf(L"socket function failed with error: %d\n", WSAGetLastError() );
        return 1;
    }

    InetAddr.sin_family = AF_INET;
    InetAddr.sin_addr.s_addr = htonl(INADDR_ANY);
    InetAddr.sin_port = htons(27015);

//-------------------------
// Bind the listening socket
    iResult = bind(ListenSocket, (SOCKADDR *) & InetAddr, sizeof (InetAddr));
    if (iResult != 0) {
        wprintf(L"bind failed with error: %d\n", WSAGetLastError() );
        return 1;
    }

//-------------------------
// Create a new event
    NewEvent = WSACreateEvent();
    if (NewEvent == NULL) {
        wprintf(L"WSACreateEvent failed with error: %d\n", GetLastError() );
        return 1;
    }

//-------------------------
// Associate event types FD_ACCEPT and FD_CLOSE
// with the listening socket and NewEvent
    iResult = WSAEventSelect(ListenSocket, NewEvent, FD_ACCEPT | FD_CLOSE);
    if (iResult != 0) {
        wprintf(L"WSAEventSelect failed with error: %d\n", WSAGetLastError() );
        return 1;
    }

//-------------------------
// Start listening on the socket
    iResult = listen(ListenSocket, 10);
    if (iResult != 0) {
        wprintf(L"listen failed with error: %d\n", WSAGetLastError() );
        return 1;
    }

//-------------------------
// Add the socket and event to the arrays, increment number of events
    SocketArray[EventTotal] = ListenSocket;
    EventArray[EventTotal] = NewEvent;
    EventTotal++;
```

```
//-------------------------
// Wait for network events on all sockets
    Index = WSAWaitForMultipleEvents(EventTotal, EventArray, FALSE, WSA_INFINITE, FALSE);
    Index = Index - WSA_WAIT_EVENT_0;

//-------------------------
// Iterate through all events and enumerate
// if the wait does not fail.
    for (i = Index; i < EventTotal; i++) {
        Index = WSAWaitForMultipleEvents(1, &EventArray[i], TRUE, 1000, FALSE);
        if ((Index != WSA_WAIT_FAILED) && (Index != WSA_WAIT_TIMEOUT)) {
            WSAEnumNetworkEvents(SocketArray[i], EventArray[i], &NetworkEvents);
        }
    }

//...
    return 0;
```

Windows Phone 8: This function is supported for Windows Phone Store apps on Windows Phone 8 and later.

Windows 8.1 and **Windows Server 2012 R2**: This function is supported for Windows Store apps on Windows 8.1, Windows Server 2012 R2, and later.

Requirements

Minimum supported client	Windows 8.1, Windows Vista [desktop apps | UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps | UWP apps]
Target Platform	Windows
Header	winsock2.h
Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

WSAEventSelect

Winsock Functions

Winsock Reference

WSAEnumProtocolsA function (winsock2.h)

10/22/2022 • 5 minutes to read • Edit Online

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

Syntax

```
int WSAAPI WSAEnumProtocolsA(
  [in]      LPINT               lpiProtocols,
  [out]     LPWSAPROTOCOL_INFOA lpProtocolBuffer,
  [in, out] LPDWORD             lpdwBufferLength
);
```

Parameters

`[in] lpiProtocols`

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

`[out] lpProtocolBuffer`

A pointer to a buffer that is filled with WSAPROTOCOL_INFO structures.

`[in, out] lpdwBufferLength`

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

Return value

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

ERROR CODE	MEANING
WSANOTINITIALISED	A successful WSAStartup call must occur before using this function.
WSAENETDOWN	The network subsystem has failed.
WSAEINPROGRESS	A blocking Windows Sockets 1.1 call is in progress.
WSAEINVAL	Indicates that one of the specified parameters was invalid.

WSAENOBUFS	The buffer length was too small to receive all the relevant WSAPROTOCOL_INFO structures and associated information. Pass in a buffer at least as large as the value returned in *lpdwBufferLength*.
WSAEFAULT	One or more of the *lpiProtocols*, *lpProtocolBuffer*, or *lpdwBufferLength* parameters are not a valid part of the user address space.

Remarks

The **WSAEnumProtocols** function is used to discover information about the collection of transport protocols installed on the local computer. Layered protocols are only usable by applications when installed in protocol chains. Information on layered protocols is not returned except for any dummy layered service providers (LSPs) installed with a chain length of zero in the *lpProtocolBuffer*.

> **Note** Layered Service Providers are deprecated. Starting with Windows 8 and Windows Server 2012, use Windows Filtering Platform.

The *lpiProtocols* parameter can be used as a filter to constrain the amount of information provided. Often, *lpiProtocols* will be specified as a **NULL** pointer that will cause the function to return information on all available transport protocols and protocol chains.

The **WSAEnumProtocols** function differs from the WSCEnumProtocols and WSCEnumProtocols32 functions in that the **WSAEnumProtocols** function doesn't return WSAPROTOCOL_INFO structures for all installed protocols. The **WSAEnumProtocols** function excludes protocols that the service provider has set with the **PFL_HIDDEN** flag in the **dwProviderFlags** member of the WSAPROTOCOL_INFO structure to indicate to the Ws2_32.dll that this protocol should not be returned in the result buffer generated by **WSAEnumProtocols** function. In addition, the **WSAEnumProtocols** function does not return data for WSAPROTOCOL_INFO structures that have a chain length of one or greater (an LSP provider). The **WSAEnumProtocols** only returns information on base protocols and protocol chains that lack the **PFL_HIDDEN** flag and don't have a protocol chain length of zero.

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

The order in which the WSAPROTOCOL_INFO structures appear in the buffer coincides with the order in which the protocol entries were registered by the service provider using the WS2_32.DLL, or with any subsequent reordering that occurred through the Windows Sockets application or DLL supplied for establishing default TCP/IP providers.

Windows Phone 8: The **WSAEnumProtocolsW** function is supported for Windows Phone Store apps on Windows Phone 8 and later.

Windows 8.1 and **Windows Server 2012 R2**: The **WSAEnumProtocolsW** function is supported for Windows Store apps on Windows 8.1, Windows Server 2012 R2, and later.

Examples

The following example demonstrates the use of the **WSAEnumProtocols** function to retrieve an array of WSAPROTOCOL_INFO structures for available transport protocols.

```
#ifndef UNICODE
#define UNICODE 1
```

```
#define UNICODE 1
#endif

#include <winsock2.h>
#include <ws2tcpip.h>
#include <objbase.h>
#include <stdio.h>

// Link with ws2_32.lib and ole32.lib
#pragma comment (lib, "Ws2_32.lib")
#pragma comment (lib, "ole32.lib")

#define MALLOC(x) HeapAlloc(GetProcessHeap(), 0, (x))
#define FREE(x) HeapFree(GetProcessHeap(), 0, (x))
// Note: could also use malloc() and free()

int wmain()
{

    //-----------------------------------------
    // Declare and initialize variables
    WSADATA wsaData;
    int iResult = 0;

    int iError = 0;
    INT iNuminfo = 0;

    int i;

    // Allocate a 16K buffer to retrieve all the protocol providers
    DWORD dwBufferLen = 16384;

    LPWSAPROTOCOL_INFO lpProtocolInfo = NULL;

    // variables needed for converting provider GUID to a string
    int iRet = 0;
    WCHAR GuidString[40] = { 0 };

    // Initialize Winsock
    iResult = WSAStartup(MAKEWORD(2, 2), &wsaData);
    if (iResult != 0) {
        wprintf(L"WSAStartup failed: %d\n", iResult);
        return 1;
    }

    lpProtocolInfo = (LPWSAPROTOCOL_INFO) MALLOC(dwBufferLen);
    if (lpProtocolInfo == NULL) {
        wprintf(L"Memory allocation for providers buffer failed\n");
        WSACleanup();
        return 1;
    }

    iNuminfo = WSAEnumProtocols(NULL, lpProtocolInfo, &dwBufferLen);
    if (iNuminfo == SOCKET_ERROR) {
        iError = WSAGetLastError();
        if (iError != WSAENOBUFS) {
            wprintf(L"WSAEnumProtocols failed with error: %d\n", iError);
            if (lpProtocolInfo) {
                FREE(lpProtocolInfo);
                lpProtocolInfo = NULL;
            }
            WSACleanup();
            return 1;
        } else {
            wprintf(L"WSAEnumProtocols failed with error: WSAENOBUFS (%d)\n",
                    iError);
            wprintf(L"  Increasing buffer size to %d\n\n", dwBufferLen);
            if (lpProtocolInfo) {
                FREE(lpProtocolInfo);
```

```
                lpProtocolInfo = NULL;
            }
            lpProtocolInfo = (LPWSAPROTOCOL_INFO) MALLOC(dwBufferLen);
            if (lpProtocolInfo == NULL) {
                wprintf(L"Memory allocation increase for buffer failed\n");
                WSACleanup();
                return 1;
            }
            iNuminfo = WSAEnumProtocols(NULL, lpProtocolInfo, &dwBufferLen);
            if (iNuminfo == SOCKET_ERROR) {
                iError = WSAGetLastError();
                wprintf(L"WSAEnumProtocols failed with error: %d\n", iError);
                if (lpProtocolInfo) {
                    FREE(lpProtocolInfo);
                    lpProtocolInfo = NULL;
                }
                WSACleanup();
                return 1;
            }

        }
    }

    wprintf(L"WSAEnumProtocols succeeded with protocol count = %d\n\n",
            iNuminfo);
    for (i = 0; i < iNuminfo; i++) {
        wprintf(L"Winsock Catalog Provider Entry #%d\n", i);
        wprintf
            (L"----------------------------------------------------------\n");
        wprintf(L"Entry type:\t\t\t ");
        if (lpProtocolInfo[i].ProtocolChain.ChainLen == 1)
            wprintf(L"Base Service Provider\n");
        else
            wprintf(L"Layered Chain Entry\n");

        wprintf(L"Protocol:\t\t\t %ws\n", lpProtocolInfo[i].szProtocol);

        iRet =
            StringFromGUID2(lpProtocolInfo[i].ProviderId,
                            (LPOLESTR) & GuidString, 39);
        if (iRet == 0)
            wprintf(L"StringFromGUID2 failed\n");
        else
            wprintf(L"Provider ID:\t\t\t %ws\n", GuidString);

        wprintf(L"Catalog Entry ID:\t\t %u\n",
                lpProtocolInfo[i].dwCatalogEntryId);

        wprintf(L"Version:\t\t\t %d\n", lpProtocolInfo[i].iVersion);

        wprintf(L"Address Family:\t\t\t %d\n",
                lpProtocolInfo[i].iAddressFamily);
        wprintf(L"Max Socket Address Length:\t %d\n",
                lpProtocolInfo[i].iMaxSockAddr);
        wprintf(L"Min Socket Address Length:\t %d\n",
                lpProtocolInfo[i].iMinSockAddr);

        wprintf(L"Socket Type:\t\t\t %d\n", lpProtocolInfo[i].iSocketType);
        wprintf(L"Socket Protocol:\t\t %d\n", lpProtocolInfo[i].iProtocol);
        wprintf(L"Socket Protocol Max Offset:\t %d\n",
                lpProtocolInfo[i].iProtocolMaxOffset);

        wprintf(L"Network Byte Order:\t\t %d\n",
                lpProtocolInfo[i].iNetworkByteOrder);
        wprintf(L"Security Scheme:\t\t %d\n",
                lpProtocolInfo[i].iSecurityScheme);
        wprintf(L"Max Message Size:\t\t %u\n", lpProtocolInfo[i].dwMessageSize);

        wprintf(L"ServiceFlags1:\t\t\t 0x%x\n",
```

```
                lpProtocolInfo[i].dwServiceFlags1);
        wprintf(L"ServiceFlags2:\t\t\t 0x%x\n",
                lpProtocolInfo[i].dwServiceFlags2);
        wprintf(L"ServiceFlags3:\t\t\t 0x%x\n",
                lpProtocolInfo[i].dwServiceFlags3);
        wprintf(L"ServiceFlags4:\t\t\t 0x%x\n",
                lpProtocolInfo[i].dwServiceFlags4);
        wprintf(L"ProviderFlags:\t\t\t 0x%x\n",
                lpProtocolInfo[i].dwProviderFlags);

        wprintf(L"Protocol Chain length:\t\t %d\n",
                lpProtocolInfo[i].ProtocolChain.ChainLen);

        wprintf(L"\n");
    }

    if (lpProtocolInfo) {
        FREE(lpProtocolInfo);
        lpProtocolInfo = NULL;
    }
    WSACleanup();

    return 0;
}
```

NOTE

The winsock2.h header defines WSAEnumProtocols as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see Conventions for Function Prototypes.

Requirements

Minimum supported client	Windows 8.1, Windows Vista [desktop apps | UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps | UWP apps]
Target Platform	Windows
Header	winsock2.h
Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

WSAPROTOCOL_INFO

WSCEnumProtocols

WSCEnumProtocols32

Winsock Functions

Winsock Reference

WSAEnumProtocolsW function (winsock2.h)

10/22/2022 • 5 minutes to read • Edit Online

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

Syntax

```
int WSAAPI WSAEnumProtocolsW(
  [in]      LPINT               lpiProtocols,
  [out]     LPWSAPROTOCOL_INFOW lpProtocolBuffer,
  [in, out] LPDWORD             lpdwBufferLength
);
```

Parameters

`[in] lpiProtocols`

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

`[out] lpProtocolBuffer`

A pointer to a buffer that is filled with WSAPROTOCOL_INFO structures.

`[in, out] lpdwBufferLength`

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

Return value

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

ERROR CODE	MEANING
WSANOTINITIALISED	A successful WSAStartup call must occur before using this function.
WSAENETDOWN	The network subsystem has failed.
WSAEINPROGRESS	A blocking Windows Sockets 1.1 call is in progress.
WSAEINVAL	Indicates that one of the specified parameters was invalid.

WSAENOBUFS	The buffer length was too small to receive all the relevant WSAPROTOCOL_INFO structures and associated information. Pass in a buffer at least as large as the value returned in *lpdwBufferLength*.
WSAEFAULT	One or more of the *lpiProtocols*, *lpProtocolBuffer*, or *lpdwBufferLength* parameters are not a valid part of the user address space.

Remarks

The **WSAEnumProtocols** function is used to discover information about the collection of transport protocols installed on the local computer. Layered protocols are only usable by applications when installed in protocol chains. Information on layered protocols is not returned except for any dummy layered service providers (LSPs) installed with a chain length of zero in the *lpProtocolBuffer*.

> **Note** Layered Service Providers are deprecated. Starting with Windows 8 and Windows Server 2012, use Windows Filtering Platform.

The *lpiProtocols* parameter can be used as a filter to constrain the amount of information provided. Often, *lpiProtocols* will be specified as a **NULL** pointer that will cause the function to return information on all available transport protocols and protocol chains.

The **WSAEnumProtocols** function differs from the WSCEnumProtocols and WSCEnumProtocols32 functions in that the **WSAEnumProtocols** function doesn't return WSAPROTOCOL_INFO structures for all installed protocols. The **WSAEnumProtocols** function excludes protocols that the service provider has set with the **PFL_HIDDEN** flag in the **dwProviderFlags** member of the WSAPROTOCOL_INFO structure to indicate to the Ws2_32.dll that this protocol should not be returned in the result buffer generated by **WSAEnumProtocols** function. In addition, the **WSAEnumProtocols** function does not return data for WSAPROTOCOL_INFO structures that have a chain length of one or greater (an LSP provider). The **WSAEnumProtocols** only returns information on base protocols and protocol chains that lack the **PFL_HIDDEN** flag and don't have a protocol chain length of zero.

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

The order in which the WSAPROTOCOL_INFO structures appear in the buffer coincides with the order in which the protocol entries were registered by the service provider using the WS2_32.DLL, or with any subsequent reordering that occurred through the Windows Sockets application or DLL supplied for establishing default TCP/IP providers.

Windows Phone 8: The **WSAEnumProtocolsW** function is supported for Windows Phone Store apps on Windows Phone 8 and later.

Windows 8.1 and **Windows Server 2012 R2**: The **WSAEnumProtocolsW** function is supported for Windows Store apps on Windows 8.1, Windows Server 2012 R2, and later.

Examples

The following example demonstrates the use of the **WSAEnumProtocols** function to retrieve an array of WSAPROTOCOL_INFO structures for available transport protocols.

```
#ifndef UNICODE
#define UNICODE 1
```

```
#define UNICODE 1
#endif

#include <winsock2.h>
#include <ws2tcpip.h>
#include <objbase.h>
#include <stdio.h>

// Link with ws2_32.lib and ole32.lib
#pragma comment (lib, "Ws2_32.lib")
#pragma comment (lib, "ole32.lib")

#define MALLOC(x) HeapAlloc(GetProcessHeap(), 0, (x))
#define FREE(x) HeapFree(GetProcessHeap(), 0, (x))
// Note: could also use malloc() and free()

int wmain()
{

    //-----------------------------------------
    // Declare and initialize variables
    WSADATA wsaData;
    int iResult = 0;

    int iError = 0;
    INT iNuminfo = 0;

    int i;

    // Allocate a 16K buffer to retrieve all the protocol providers
    DWORD dwBufferLen = 16384;

    LPWSAPROTOCOL_INFO lpProtocolInfo = NULL;

    // variables needed for converting provider GUID to a string
    int iRet = 0;
    WCHAR GuidString[40] = { 0 };

    // Initialize Winsock
    iResult = WSAStartup(MAKEWORD(2, 2), &wsaData);
    if (iResult != 0) {
        wprintf(L"WSAStartup failed: %d\n", iResult);
        return 1;
    }

    lpProtocolInfo = (LPWSAPROTOCOL_INFO) MALLOC(dwBufferLen);
    if (lpProtocolInfo == NULL) {
        wprintf(L"Memory allocation for providers buffer failed\n");
        WSACleanup();
        return 1;
    }

    iNuminfo = WSAEnumProtocols(NULL, lpProtocolInfo, &dwBufferLen);
    if (iNuminfo == SOCKET_ERROR) {
        iError = WSAGetLastError();
        if (iError != WSAENOBUFS) {
            wprintf(L"WSAEnumProtocols failed with error: %d\n", iError);
            if (lpProtocolInfo) {
                FREE(lpProtocolInfo);
                lpProtocolInfo = NULL;
            }
            WSACleanup();
            return 1;
        } else {
            wprintf(L"WSAEnumProtocols failed with error: WSAENOBUFS (%d)\n",
                    iError);
            wprintf(L"  Increasing buffer size to %d\n\n", dwBufferLen);
            if (lpProtocolInfo) {
                FREE(lpProtocolInfo);
```

```
                lpProtocolInfo = NULL;
            }
            lpProtocolInfo = (LPWSAPROTOCOL_INFO) MALLOC(dwBufferLen);
            if (lpProtocolInfo == NULL) {
                wprintf(L"Memory allocation increase for buffer failed\n");
                WSACleanup();
                return 1;
            }
            iNuminfo = WSAEnumProtocols(NULL, lpProtocolInfo, &dwBufferLen);
            if (iNuminfo == SOCKET_ERROR) {
                iError = WSAGetLastError();
                wprintf(L"WSAEnumProtocols failed with error: %d\n", iError);
                if (lpProtocolInfo) {
                    FREE(lpProtocolInfo);
                    lpProtocolInfo = NULL;
                }
                WSACleanup();
                return 1;
            }

        }
    }

    wprintf(L"WSAEnumProtocols succeeded with protocol count = %d\n\n",
            iNuminfo);
    for (i = 0; i < iNuminfo; i++) {
        wprintf(L"Winsock Catalog Provider Entry #%d\n", i);
        wprintf
            (L"----------------------------------------------------------\n");
        wprintf(L"Entry type:\t\t\t ");
        if (lpProtocolInfo[i].ProtocolChain.ChainLen == 1)
            wprintf(L"Base Service Provider\n");
        else
            wprintf(L"Layered Chain Entry\n");

        wprintf(L"Protocol:\t\t\t %ws\n", lpProtocolInfo[i].szProtocol);

        iRet =
            StringFromGUID2(lpProtocolInfo[i].ProviderId,
                            (LPOLESTR) & GuidString, 39);
        if (iRet == 0)
            wprintf(L"StringFromGUID2 failed\n");
        else
            wprintf(L"Provider ID:\t\t\t %ws\n", GuidString);

        wprintf(L"Catalog Entry ID:\t\t %u\n",
                lpProtocolInfo[i].dwCatalogEntryId);

        wprintf(L"Version:\t\t\t %d\n", lpProtocolInfo[i].iVersion);

        wprintf(L"Address Family:\t\t\t %d\n",
                lpProtocolInfo[i].iAddressFamily);
        wprintf(L"Max Socket Address Length:\t %d\n",
                lpProtocolInfo[i].iMaxSockAddr);
        wprintf(L"Min Socket Address Length:\t %d\n",
                lpProtocolInfo[i].iMinSockAddr);

        wprintf(L"Socket Type:\t\t\t %d\n", lpProtocolInfo[i].iSocketType);
        wprintf(L"Socket Protocol:\t\t %d\n", lpProtocolInfo[i].iProtocol);
        wprintf(L"Socket Protocol Max Offset:\t %d\n",
                lpProtocolInfo[i].iProtocolMaxOffset);

        wprintf(L"Network Byte Order:\t\t %d\n",
                lpProtocolInfo[i].iNetworkByteOrder);
        wprintf(L"Security Scheme:\t\t %d\n",
                lpProtocolInfo[i].iSecurityScheme);
        wprintf(L"Max Message Size:\t\t %u\n", lpProtocolInfo[i].dwMessageSize);

        wprintf(L"ServiceFlags1:\t\t\t 0x%x\n",
```

```
                lpProtocolInfo[i].dwServiceFlags1);
        wprintf(L"ServiceFlags2:\t\t\t 0x%x\n",
                lpProtocolInfo[i].dwServiceFlags2);
        wprintf(L"ServiceFlags3:\t\t\t 0x%x\n",
                lpProtocolInfo[i].dwServiceFlags3);
        wprintf(L"ServiceFlags4:\t\t\t 0x%x\n",
                lpProtocolInfo[i].dwServiceFlags4);
        wprintf(L"ProviderFlags:\t\t\t 0x%x\n",
                lpProtocolInfo[i].dwProviderFlags);

        wprintf(L"Protocol Chain length:\t\t %d\n",
                lpProtocolInfo[i].ProtocolChain.ChainLen);

        wprintf(L"\n");
    }

    if (lpProtocolInfo) {
        FREE(lpProtocolInfo);
        lpProtocolInfo = NULL;
    }
    WSACleanup();

    return 0;
}
```

NOTE

The winsock2.h header defines WSAEnumProtocols as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see Conventions for Function Prototypes.

Requirements

Minimum supported client	Windows 8.1, Windows Vista [desktop apps | UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps | UWP apps]
Target Platform	Windows
Header	winsock2.h
Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

WSAPROTOCOL_INFO

WSCEnumProtocols

WSCEnumProtocols32

Winsock Functions

Winsock Reference

WSAEventSelect function (winsock2.h)

10/22/2022 • 11 minutes to read • Edit Online

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

Syntax

```
int WSAAPI WSAEventSelect(
  [in] SOCKET   s,
  [in] WSAEVENT hEventObject,
  [in] long     lNetworkEvents
);
```

Parameters

`[in] s`

A descriptor identifying the socket.

`[in] hEventObject`

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

`[in] lNetworkEvents`

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

Return value

The return value is zero if the application's specification of the network events and the associated event object was successful. Otherwise, the value SOCKET_ERROR is returned, and a specific error number can be retrieved by calling WSAGetLastError.

As in the case of the select and WSAAsyncSelect functions, **WSAEventSelect** will frequently be used to determine when a data transfer operation (send or recv) can be issued with the expectation of immediate success. Nevertheless, a robust application must be prepared for the possibility that the event object is set and it issues a Windows Sockets call that returns WSAEWOULDBLOCK immediately. For example, the following sequence of operations is possible:

- Data arrives on socket *s*; Windows Sockets sets the **WSAEventSelect** event object.
- The application does some other processing.
- While processing, the application issues an ioctlsocket(*s*, FIONREAD...) and notices that there is data ready to be read.
- The application issues a recv(*s*,...) to read the data.
- The application eventually waits on the event object specified in **WSAEventSelect**, which returns immediately indicating that data is ready to read.
- The application issues recv(*s*,...), which fails with the error WSAEWOULDBLOCK.

Having successfully recorded the occurrence of the network event (by setting the corresponding bit in the internal network event record) and signaled the associated event object, no further actions are taken for that network event

until the application makes the function call that implicitly reenables the setting of that network event and signaling of the associated event object.

NETWORK EVENT	RE-ENABLING FUNCTION
FD_READ	The recv, recvfrom, WSARecv, WSARecvEx, or WSARecvFrom function.
FD_WRITE	The send, sendto, WSASend, or WSASendTo function.
FD_OOB	The recv, recvfrom, WSARecv, WSARecvEx, or WSARecvFrom function.
FD_ACCEPT	The accept, AcceptEx, or WSAAccept function unless the error code returned is WSATRY_AGAIN indicating that the condition function returned CF_DEFER.
FD_CONNECT	None.
FD_CLOSE	None.
FD_QOS	The WSAIoctl function with command SIO_GET_QOS.
FD_GROUP_QOS	Reserved.
FD_ROUTING_ INTERFACE_CHANGE	The WSAIoctl function with command SIO_ROUTING_INTERFACE_CHANGE.
FD_ADDRESS_ LIST_CHANGE	The WSAIoctl function with command SIO_ADDRESS_LIST_CHANGE.

Any call to the reenabling routine, even one that fails, results in reenabling of recording and signaling for the relevant network event and event object.

For FD_READ, FD_OOB, and FD_ACCEPT network events, network event recording and event object signaling are level-triggered. This means that if the reenabling routine is called and the relevant network condition is still valid after the call, the network event is recorded and the associated event object is set. This allows an application to be event-driven and not be concerned with the amount of data that arrives at any one time. Consider the following sequence:

1. The transport provider receives 100 bytes of data on socket *s* and causes WS2_32.DLL to record the FD_READ network event and set the associated event object.
2. The application issues recv(*s*, *buffptr*, 50, 0) to read 50 bytes.
3. The transport provider causes WS2_32.DLL to record the FD_READ network event and sets the associated event object again since there is still data to be read.

With these semantics, an application need not read all available data in response to an FD_READ network event—a

single recv in response to each FD_READ network event is appropriate.

The FD_QOS event is considered edge triggered. A message will be posted exactly once when a quality of service change occurs. Further messages will not be forthcoming until either the provider detects a further change in quality of service or the application renegotiates the quality of service for the socket.

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

If a network event has already happened when the application calls **WSAEventSelect** or when the reenabling function is called, then a network event is recorded and the associated event object is set as appropriate. For example, consider the following sequence:

1. An application calls listen.
2. A connect request is received but not yet accepted.
3. The application calls **WSAEventSelect** specifying that it is interested in the FD_ACCEPT network event for the socket. Due to the persistence of network events, Windows Sockets records the FD_ACCEPT network event and sets the associated event object immediately.

The FD_WRITE network event is handled slightly differently. An FD_WRITE network event is recorded when a socket is first connected with a call to the connect, ConnectEx, WSAConnect, WSAConnectByList, or WSAConnectByName function or when a socket is accepted with accept, AcceptEx, or WSAAccept function and then after a send fails with WSAEWOULDBLOCK and buffer space becomes available. Therefore, an application can assume that sends are possible starting from the first FD_WRITE network event setting and lasting until a send returns WSAEWOULDBLOCK. After such a failure the application will find out that sends are again possible when an FD_WRITE network event is recorded and the associated event object is set.

The FD_OOB network event is used only when a socket is configured to receive OOB data separately. If the socket is configured to receive OOB data inline, the OOB (expedited) data is treated as normal data and the application should register an interest in, and will get FD_READ network event, not FD_OOB network event. An application can set or inspect the way in which OOB data is to be handled by using setsockopt or getsockopt for the SO_OOBINLINE option.

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

The FD_CLOSE network event is recorded when a close indication is received for the virtual circuit corresponding to the socket. In TCP terms, this means that the FD_CLOSE is recorded when the connection goes into the TIME WAIT or CLOSE WAIT states. This results from the remote end performing a shutdown on the send side or a closesocket. FD_CLOSE being posted after all data is read from a socket. An application should check for remaining data upon receipt of FD_CLOSE to avoid any possibility of losing data. For more information, see the section on Graceful Shutdown, Linger Options, and Socket Closure and the **shutdown** function.

Note that Windows Sockets will record only an FD_CLOSE network event to indicate closure of a virtual circuit. It will not record an FD_READ network event to indicate this condition.

The FD_QOS or FD_GROUP_QOS network event is recorded when any parameter in the flow specification associated with socket *s*. Applications should use WSAIoctl with command **SIO_GET_QOS** to get the current quality of service for socket *s*.

The FD_ROUTING_INTERFACE_CHANGE network event is recorded when the local interface that should be used to reach the destination specified in WSAIoctl with **SIO_ROUTING_INTERFACE_CHANGE** changes after such IOCTL has been issued.

The FD_ADDRESS_LIST_CHANGE network event is recorded when the list of addresses of protocol family for the socket to which the application can bind changes after WSAIoctl with **SIO_ADDRESS_LIST_CHANGE** has been issued.

ERROR CODE	MEANING
WSANOTINITIALISED	A successful WSAStartup call must occur before using this function.
WSAENETDOWN	The network subsystem has failed.
WSAEINVAL	One of the specified parameters was invalid, or the specified socket is in an invalid state.
WSAEINPROGRESS	A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function.
WSAENOTSOCK	The descriptor is not a socket.

Remarks

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

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

The proper way to reset the state of an event object used with the **WSAEventSelect** function is to pass the handle of the event object to the WSAEnumNetworkEvents function in the *hEventObject* parameter. This will reset the event object and adjust the status of active FD events on the socket in an atomic fashion.

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

The **WSAEventSelect** function automatically sets socket *s* to nonblocking mode, regardless of the value of *lNetworkEvents*. To set socket *s* back to blocking mode, it is first necessary to clear the event record associated with socket *s* via a call to **WSAEventSelect** with *lNetworkEvents* set to zero and the *hEventObject* parameter set to **NULL**. You can then call ioctlsocket or WSAIoctl to set the socket back to blocking mode.

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

VALUE	MEANING
FD_READ	Wants to receive notification of readiness for reading.
FD_WRITE	Wants to receive notification of readiness for writing.
FD_OOB	Wants to receive notification of the arrival of OOB data.

FD_ACCEPT	Wants to receive notification of incoming connections.
FD_CONNECT	Wants to receive notification of completed connection or multipoint join operation.
FD_CLOSE	Wants to receive notification of socket closure.
FD_QOS	Wants to receive notification of socket (QoS changes.
FD_GROUP_QOS	Reserved for future use with socket groups. Want to receive notification of socket group QoS changes.
FD_ROUTING_ INTERFACE_CHANGE	Wants to receive notification of routing interface changes for the specified destination.
FD_ADDRESS_ LIST_CHANGE	Wants to receive notification of local address list changes for the address family of the socket.

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

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

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

```
rc = WSAEventSelect(s, hEventObject1, FD_READ);
rc = WSAEventSelect(s, hEventObject2, FD_WRITE); //bad
```

To cancel the association and selection of network events on a socket, *lNetworkEvents* should be set to zero, in which case the *hEventObject* parameter will be ignored.

```
rc = WSAEventSelect(s, hEventObject, 0);
```

Closing a socket with closesocket also cancels the association and selection of network events specified in **WSAEventSelect** for the socket. The application, however, still must call WSACloseEvent to explicitly close the event object and free any resources.

The socket created when the accept function is called has the same properties as the listening socket used to accept it. Any **WSAEventSelect** association and network events selection set for the listening socket apply to the accepted socket. For example, if a listening socket has **WSAEventSelect** association of *hEventObject* with FD_ACCEPT, FD_READ, and FD_WRITE, then any socket accepted on that listening socket will also have FD_ACCEPT, FD_READ, and FD_WRITE network events associated with the same *hEventObject*. If a different *hEventObject* or network events are desired, the application should call **WSAEventSelect**, passing the accepted socket and the desired new information.

Example Code

The following example demonstrates the use of the **WSAEventSelect** function.

```
//-------------------------
// Declare and initialize variables
SOCKET ListenSocket;
WSAEVENT NewEvent;
sockaddr_in InetAddr;

//-------------------------
// Initialize listening socket
ListenSocket = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);

//-------------------------
// Bind listening socket
InetAddr.sin_family = AF_INET;
InetAddr.sin_addr.s_addr = htonl(INADDR_ANY);
InetAddr.sin_port = htons(27015);

bind (ListenSocket, (SOCKADDR *) &InetAddr, sizeof(InetAddr));

//-------------------------
// Create new event
NewEvent = WSACreateEvent();

//-------------------------
// Associate event types FD_ACCEPT and FD_CLOSE
// with the listening socket and NewEvent
WSAEventSelect( ListenSocket, NewEvent, FD_ACCEPT | FD_CLOSE);

//----------------------
// Listen for incoming connection requests
// on the created socket
if (listen( ListenSocket, SOMAXCONN ) == SOCKET_ERROR)
    printf("Error listening on socket.\n");

printf("Listening on socket...\n");

// Need an event handler added to handle connection requests
```

Windows Phone 8: This function is supported for Windows Phone Store apps on Windows Phone 8 and later.

Windows 8.1 and **Windows Server 2012 R2**: This function is supported for Windows Store apps on Windows 8.1, Windows Server 2012 R2, and later.

Requirements

Minimum supported client	Windows 8.1, Windows Vista [desktop apps | UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps | UWP apps]
Target Platform	Windows
Header	winsock2.h
Library	Ws2_32.lib

DLL	Ws2_32.dll

See also

WSAAsyncSelect

WSACloseEvent

WSACreateEvent

WSAEnumNetworkEvents

WSAWaitForMultipleEvents

Winsock Functions

Winsock Reference

shutdown

WSAGetLastError function (winsock2.h)

10/22/2022 • 2 minutes to read • Edit Online

The **WSAGetLastError** function returns the error status for the last Windows Sockets operation that failed.

Syntax

```
int WSAAPI WSAGetLastError();
```

Return value

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

Remarks

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. This extended error code can be different from the error code obtained from getsockopt when called with an *optname* parameter of **SO_ERROR**, which is socket-specific since **WSAGetLastError** is for all thread-specific sockets.

If a function call's return value indicates that error or other relevant data was returned in the error code, **WSAGetLastError** should be called immediately. This is necessary because some functions may reset the last extended error code to 0 if they succeed, overwriting the extended error code returned by a previously failed function. To specifically reset the extended error code, use the WSASetLastError function call with the *iError* parameter set to zero. A getsockopt function when called with an *optname* parameter of **SO_ERROR** also resets the extended error code to zero.

The **WSAGetLastError** function should not be used to check for an extended error value on receipt of an asynchronous message. In this case, the extended error value is passed in the *lParam* parameter of the message, and this can differ from the value returned by **WSAGetLastError**.

> **Note** An application can call the **WSAGetLastError** function to determine the extended error code for other Windows sockets functions as is normally done in Windows Sockets even if the WSAStartup function fails or the **WSAStartup** function was not called to properly initialize Windows Sockets before calling a Windows Sockets function. The **WSAGetLastError** function is one of the only functions in the Winsock 2.2 DLL that can be called in the case of a **WSAStartup** failure.

The Windows Sockets extended error codes returned by this function and the text description of the error are listed under Windows Sockets Error Codes. 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.

Windows Phone 8: This function is supported for Windows Phone Store apps on Windows Phone 8 and later.

Windows 8.1 and **Windows Server 2012 R2**: This function is supported for Windows Store apps on Windows 8.1, Windows Server 2012 R2, and later.

Requirements

Minimum supported client	Windows 8.1, Windows Vista [desktop apps | UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps | UWP apps]
Target Platform	Windows
Header	winsock2.h (include Winsock2.h)
Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

Error Codes - errno, h_errno and WSAGetLastError

FormatMessage

WSASetLastError

WSAStartup

Windows Sockets Error Codes

Winsock Functions

Winsock Reference

getsockopt

WSAGetOverlappedResult function (winsock2.h)

10/22/2022 • 4 minutes to read • Edit Online

The **WSAGetOverlappedResult** function retrieves the results of an overlapped operation on the specified socket.

Syntax

```
BOOL WSAAPI WSAGetOverlappedResult(
  [in]  SOCKET          s,
  [in]  LPWSAOVERLAPPED lpOverlapped,
  [out] LPDWORD         lpcbTransfer,
  [in]  BOOL            fWait,
  [out] LPDWORD         lpdwFlags
);
```

Parameters

`[in] s`

A descriptor identifying the socket.

This is the same socket that was specified when the overlapped operation was started by a call to any of the Winsock functions that supports overlapped operations. These functions include AcceptEx, ConnectEx, DisconnectEx, TransmitFile, TransmitPackets, WSARecv, WSARecvFrom, LPFN_WSARECVMSG (WSARecvMsg), WSASend, WSASendMsg, WSASendTo, and WSAIoctl.

`[in] lpOverlapped`

A pointer to a WSAOVERLAPPED structure that was specified when the overlapped operation was started. This parameter must not be a **NULL** pointer.

`[out] lpcbTransfer`

A pointer to a 32-bit variable that receives the number of bytes that were actually transferred by a send or receive operation, or by the WSAIoctl function. This parameter must not be a **NULL** pointer.

`[in] fWait`

A flag that specifies whether the function should wait for the pending overlapped operation to complete. If **TRUE**, the function does not return until the operation has been completed. If **FALSE** and the operation is still pending, the function returns **FALSE** and the WSAGetLastError function returns WSA_IO_INCOMPLETE. The *fWait* parameter may be set to **TRUE** only if the overlapped operation selected the event-based completion notification.

`[out] lpdwFlags`

A pointer to a 32-bit variable that will receive one or more flags that supplement the completion status. If the overlapped operation was initiated through WSARecv or WSARecvFrom, this parameter will contain the results value for *lpFlags* parameter. This parameter must not be a **NULL** pointer.

Return value

If **WSAGetOverlappedResult** succeeds, the return value is **TRUE**. This means that the overlapped operation has completed successfully and that the value pointed to by *lpcbTransfer* has been updated.

If **WSAGetOverlappedResult** returns **FALSE**, this means that either the overlapped operation has not completed, the overlapped operation completed but with errors, or the overlapped operation's completion status could not be determined due to errors in one or more parameters to **WSAGetOverlappedResult**. On failure, the value pointed to by *lpcbTransfer* will not be updated. Use WSAGetLastError to determine the cause of the failure (either by the **WSAGetOverlappedResult** function or by the associated overlapped operation).

ERROR CODE	MEANING
WSANOTINITIALISED	A successful WSAStartup call must occur before using this function.
WSAENETDOWN	The network subsystem has failed.
WSAENOTSOCK	The descriptor is not a socket.
WSA_INVALID_HANDLE	The *hEvent* parameter of the WSAOVERLAPPED structure does not contain a valid event object handle.
WSA_INVALID_PARAMETER	One of the parameters is unacceptable.
WSA_IO_INCOMPLETE	The *fWait* parameter is **FALSE** and the I/O operation has not yet completed.
WSAEFAULT	One or more of the *lpOverlapped*, *lpcbTransfer*, or *lpdwFlags* parameters are not in a valid part of the user address space. This error is returned if the *lpOverlapped*, *lpcbTransfer*, or *lpdwFlags* parameter was a **NULL** pointer on Windows Server 2003 and earlier.

Remarks

The **WSAGetOverlappedResult** function reports the results of the overlapped operation specified in the *lpOverlapped* parameter for the socket specified in the *s* parameter. The **WSAGetOverlappedResult** function is passed the socket descriptor and the WSAOVERLAPPED structure that was specified when the overlapped function was called. A pending operation is indicated when the function that started the operation returns **FALSE** and the WSAGetLastError function returns WSA_IO_PENDING. When an I/O operation such as WSARecv is pending, the function that started the operation resets the *hEvent* member of the **WSAOVERLAPPED** structure to the nonsignaled state. Then, when the pending operation has completed, the system sets the event object to the signaled state.

If the *fWait* parameter is **TRUE**, **WSAGetOverlappedResult** determines whether the pending operation has been completed by waiting for the event object to be in the signaled state. A client may set the *fWait* parameter to **TRUE**, but only if it selected event-based completion notification when the I/O operation was requested. If another form of notification was selected, the usage of the *hEvent* parameter of the WSAOVERLAPPED structure is different, and setting *fWait* to **TRUE** causes unpredictable results.

If the **WSAGetOverlappedResult** function is called with the *lpOverlapped*, *lpcbTransfer*, or *lpdwFlags* parameter set to a **NULL** pointer on Windows Vista, this will result in an access violation. If the

WSAGetOverlappedResult function is called with the *lpOverlapped*, *lpcbTransfer*, or *lpdwFlags* parameter set to a **NULL** pointer on Windows Server 2003 and earlier, this will result in the WSAEFAULT error code being returned.

> **Note** All I/O is canceled when a thread exits. For overlapped sockets, pending asynchronous operations can fail if the thread is closed before the operations complete. See ExitThread for more information.

Windows Phone 8: This function is supported for Windows Phone Store apps on Windows Phone 8 and later.

Windows 8.1 and **Windows Server 2012 R2**: This function is supported for Windows Store apps on Windows 8.1, Windows Server 2012 R2, and later.

Requirements

Minimum supported client	Windows 8.1, Windows Vista [desktop apps | UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps | UWP apps]
Target Platform	Windows
Header	winsock2.h
Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

WSAAccept

WSAConnect

WSACreateEvent

WSAIoctl

WSARecv

WSARecvFrom

WSASend

WSASendTo

WSAWaitForMultipleEvents

Winsock Functions

Winsock Reference

WSAGetQOSByName function (winsock2.h)

10/22/2022 • 2 minutes to read • Edit Online

The **WSAGetQOSByName** function initializes a QOS structure based on a named template, or it supplies a buffer to retrieve an enumeration of the available template names.

Syntax

```
BOOL WSAAPI WSAGetQOSByName(
  [in]      SOCKET   s,
  [in, out] LPWSABUF lpQOSName,
  [out]     LPQOS    lpQOS
);
```

Parameters

`[in] s`

A descriptor identifying a socket.

`[in, out] lpQOSName`

A pointer to a specific quality of service template.

`[out] lpQOS`

A pointer to the QOS structure to be filled.

Return value

If **WSAGetQOSByName** succeeds, the return value is **TRUE**. If the function fails, the return value is **FALSE**. To get extended error information, call WSAGetLastError.

ERROR CODE	MEANING
WSANOTINITIALISED	A successful WSAStartup call must occur before using this function.
WSAENETDOWN	The network subsystem has failed.
WSAENOTSOCK	The descriptor is not a socket.
WSAEFAULT	The *lpQOSName* or *lpQOS* parameter are not a valid part of the user address space, or the buffer length for *lpQOS* is too small.

Remarks

The **WSAGetQOSByName** function is used by applications to initialize a QOS structure to a set of known

values appropriate for a particular service class or media type. These values are stored in a template that is referenced by a well-known name. The client may retrieve these values by setting the *buf* parameter of the WSABUF structure indicated by *lpQOSName*, which points to a string of nonzero length specifying a template name. In this case, the usage of *lpQOSName* is IN only, and results are returned through *lpQOS*.

Alternatively, the client may use this function to retrieve an enumeration of available template names. The client may do this by setting the *buf* parameter of the WSABUF indicated by *lpQOSName* to a zero-length null-terminated string. In this case the buffer indicated by *buf* is overwritten with a sequence of as many available, null-terminated template names up to the number of bytes available in *buf* as indicated by the *len* parameter of the **WSABUF** indicated by *lpQOSName*. The list of names itself is terminated by a zero-length name. When the **WSAGetQOSByName** function is used to retrieve template names, the *lpQOS* parameter is ignored.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winsock2.h
Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

QOS

WSAAccept

WSAConnect

Winsock Functions

Winsock Reference

getsockopt

WSAGetServiceClassInfoA function (winsock2.h)

10/22/2022 • 2 minutes to read • Edit Online

The **WSAGetServiceClassInfo** function retrieves the class information (schema) pertaining to a specified service class from a specified namespace provider.

Syntax

```
INT WSAAPI WSAGetServiceClassInfoA(
  [in]      LPGUID                 lpProviderId,
  [in]      LPGUID                 lpServiceClassId,
  [in, out] LPDWORD                lpdwBufSize,
  [out]     LPWSASERVICECLASSINFOA lpServiceClassInfo
);
```

Parameters

`[in] lpProviderId`

A pointer to a GUID that identifies a specific namespace provider.

`[in] lpServiceClassId`

A pointer to a GUID identifying the service class.

`[in, out] lpdwBufSize`

On input, the number of bytes contained in the buffer pointed to by the *lpServiceClassInfo* parameter.

On output, if the function fails and the error is WSAEFAULT, this parameter specifies the minimum size, in bytes, of the buffer pointed to *lpServiceClassInfo* needed to retrieve the record.

`[out] lpServiceClassInfo`

A pointer to a WSASERVICECLASSINFO structure that contains the service class information from the indicated namespace provider for the specified service class.

Return value

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

ERROR CODE	MEANING
WSA_NOT_ENOUGH_MEMORY	There was insufficient memory to perform the operation.
WSAEACCES	The calling routine does not have sufficient privileges to access the information.

WSAEFAULT	The buffer pointed to by the *lpServiceClassInfo* parameter is too small to contain a WSASERVICECLASSINFOW. The application needs to pass in a larger buffer.
WSAEINVAL	The specified service class identifier or namespace provider identifier is not valid. This error is returned if the *lpProviderId* , *lpServiceClassId*, *lpdwBufSize*, or *lpServiceClassInfo* parameters are **NULL**.
WSAEOPNOTSUPP	The operation is not supported for the type of object referenced. This error is returned by some namespace providers that do not support getting service class information.
WSANO_DATA	The requested name is valid, but no data of the requested type was found.
WSANOTINITIALISED	The WS2_32.DLL has not been initialized. The application must first call WSAStartup before calling any Windows Sockets functions.
WSATYPE_NOT_FOUND	The specified class was not found.

Remarks

The **WSAGetServiceClassInfo** function retrieves service class information from a namespace provider. The service class information retrieved from a particular namespace provider might not be the complete set of class information that was specified when the service class was installed. Individual namespace providers are only required to retain service class information that is applicable to the namespaces that they support. See the section Service Class Data Structures for more information.

> **NOTE**
> The winsock2.h header defines WSAGetServiceClassInfo as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see Conventions for Function Prototypes.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps | UWP apps]
Minimum supported server	Windows 2000 Server [desktop apps | UWP apps]
Target Platform	Windows
Header	winsock2.h
Library	Ws2_32.lib

DLL	Ws2_32.dll

See also

Service Class Data Structures

WSAInstallServiceClass

WSASERVICECLASSINFOW

WSAStartup

Winsock Functions

Winsock Reference

WSAGetServiceClassInfoW function (winsock2.h)

10/22/2022 • 2 minutes to read • Edit Online

The **WSAGetServiceClassInfo** function retrieves the class information (schema) pertaining to a specified service class from a specified namespace provider.

Syntax

```
INT WSAAPI WSAGetServiceClassInfoW(
  [in]      LPGUID                 lpProviderId,
  [in]      LPGUID                 lpServiceClassId,
  [in, out] LPDWORD                lpdwBufSize,
  [out]     LPWSASERVICECLASSINFOW lpServiceClassInfo
);
```

Parameters

`[in] lpProviderId`

A pointer to a GUID that identifies a specific namespace provider.

`[in] lpServiceClassId`

A pointer to a GUID identifying the service class.

`[in, out] lpdwBufSize`

On input, the number of bytes contained in the buffer pointed to by the *lpServiceClassInfo* parameter.

On output, if the function fails and the error is WSAEFAULT, this parameter specifies the minimum size, in bytes, of the buffer pointed to *lpServiceClassInfo* needed to retrieve the record.

`[out] lpServiceClassInfo`

A pointer to a WSASERVICECLASSINFO structure that contains the service class information from the indicated namespace provider for the specified service class.

Return value

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

ERROR CODE	MEANING
WSA_NOT_ENOUGH_MEMORY	There was insufficient memory to perform the operation.
WSAEACCES	The calling routine does not have sufficient privileges to access the information.

WSAEFAULT	The buffer pointed to by the *lpServiceClassInfo* parameter is too small to contain a WSASERVICECLASSINFOW. The application needs to pass in a larger buffer.
WSAEINVAL	The specified service class identifier or namespace provider identifier is not valid. This error is returned if the *lpProviderId* , *lpServiceClassId*, *lpdwBufSize*, or *lpServiceClassInfo* parameters are **NULL**.
WSAEOPNOTSUPP	The operation is not supported for the type of object referenced. This error is returned by some namespace providers that do not support getting service class information.
WSANO_DATA	The requested name is valid, but no data of the requested type was found.
WSANOTINITIALISED	The WS2_32.DLL has not been initialized. The application must first call WSAStartup before calling any Windows Sockets functions.
WSATYPE_NOT_FOUND	The specified class was not found.

Remarks

The **WSAGetServiceClassInfo** function retrieves service class information from a namespace provider. The service class information retrieved from a particular namespace provider might not be the complete set of class information that was specified when the service class was installed. Individual namespace providers are only required to retain service class information that is applicable to the namespaces that they support. See the section Service Class Data Structures for more information.

> **NOTE**
> The winsock2.h header defines WSAGetServiceClassInfo as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see Conventions for Function Prototypes.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps | UWP apps]
Minimum supported server	Windows 2000 Server [desktop apps | UWP apps]
Target Platform	Windows
Header	winsock2.h
Library	Ws2_32.lib

DLL	Ws2_32.dll

See also

Service Class Data Structures

WSAInstallServiceClass

WSASERVICECLASSINFOW

WSAStartup

Winsock Functions

Winsock Reference

WSAGetServiceClassNameByClassIdA function (winsock2.h)

10/22/2022 • 2 minutes to read • Edit Online

The **WSAGetServiceClassNameByClassId** function retrieves the name of the service associated with the specified type. This name is the generic service name, like FTP or SNA, and not the name of a specific instance of that service.

Syntax

```
INT WSAAPI WSAGetServiceClassNameByClassIdA(
  [in]      LPGUID  lpServiceClassId,
  [out]     LPSTR   lpszServiceClassName,
  [in, out] LPDWORD lpdwBufferLength
);
```

Parameters

`[in] lpServiceClassId`

A pointer to the GUID for the service class.

`[out] lpszServiceClassName`

A pointer to the service name.

`[in, out] lpdwBufferLength`

On input, the length of the buffer returned by *lpszServiceClassName*, in characters. On output, the length of the service name copied into *lpszServiceClassName*, in characters.

Return value

The **WSAGetServiceClassNameByClassId** function returns a value of zero if successful. Otherwise, the value SOCKET_ERROR is returned, and a specific error number can be retrieved by calling WSAGetLastError.

ERROR CODE	MEANING
WSA_INVALID_PARAMETER	The *lpServiceClassId* parameter specified is invalid.
WSA_NOT_ENOUGH_MEMORY	There was insufficient memory to perform the operation.
WSAEACCES	The calling routine does not have sufficient privileges to access the information.
WSAEFAULT	The specified buffer pointed to by *lpszServiceClassName* is too small. Pass in a larger buffer.

WSAENOBUFS	No buffer space available.
WSAEOPNOTSUPP	The operation is not supported for the type of object referenced. This error is returned by some namespace providers that do not support getting service class information.
WSANO_DATA	The *lpServiceClassId* is valid, but no data of the requested type was found.
WSANOTINITIALISED	The WS2_32.DLL has not been initialized. The application must first call WSAStartup before calling any Windows Sockets functions.

Remarks

> **NOTE**
>
> The winsock2.h header defines WSAGetServiceClassNameByClassId as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see Conventions for Function Prototypes.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps | UWP apps]
Minimum supported server	Windows 2000 Server [desktop apps | UWP apps]
Target Platform	Windows
Header	winsock2.h
Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

WSAStartup

Winsock Functions

Winsock Reference

WSAGetServiceClassNameByClassIdW function (winsock2.h)

10/22/2022 • 2 minutes to read • Edit Online

The **WSAGetServiceClassNameByClassId** function retrieves the name of the service associated with the specified type. This name is the generic service name, like FTP or SNA, and not the name of a specific instance of that service.

Syntax

```
INT WSAAPI WSAGetServiceClassNameByClassIdW(
  [in]      LPGUID  lpServiceClassId,
  [out]     LPWSTR  lpszServiceClassName,
  [in, out] LPDWORD lpdwBufferLength
);
```

Parameters

`[in] lpServiceClassId`

A pointer to the GUID for the service class.

`[out] lpszServiceClassName`

A pointer to the service name.

`[in, out] lpdwBufferLength`

On input, the length of the buffer returned by *lpszServiceClassName*, in characters. On output, the length of the service name copied into *lpszServiceClassName*, in characters.

Return value

The **WSAGetServiceClassNameByClassId** function returns a value of zero if successful. Otherwise, the value SOCKET_ERROR is returned, and a specific error number can be retrieved by calling WSAGetLastError.

ERROR CODE	MEANING
WSA_INVALID_PARAMETER	The *lpServiceClassId* parameter specified is invalid.
WSA_NOT_ENOUGH_MEMORY	There was insufficient memory to perform the operation.
WSAEACCES	The calling routine does not have sufficient privileges to access the information.
WSAEFAULT	The specified buffer pointed to by *lpszServiceClassName* is too small. Pass in a larger buffer.

WSAENOBUFS	No buffer space available.
WSAEOPNOTSUPP	The operation is not supported for the type of object referenced. This error is returned by some namespace providers that do not support getting service class information.
WSANO_DATA	The *lpServiceClassId* is valid, but no data of the requested type was found.
WSANOTINITIALISED	The WS2_32.DLL has not been initialized. The application must first call WSAStartup before calling any Windows Sockets functions.

Remarks

> **NOTE**
>
> The winsock2.h header defines WSAGetServiceClassNameByClassId as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see Conventions for Function Prototypes.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps | UWP apps]
Minimum supported server	Windows 2000 Server [desktop apps | UWP apps]
Target Platform	Windows
Header	winsock2.h
Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

WSAStartup

Winsock Functions

Winsock Reference

WSAHtonl function (winsock2.h)

10/22/2022 • 2 minutes to read • Edit Online

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

Syntax

```
int WSAAPI WSAHtonl(
  [in]  SOCKET s,
  [in]  u_long hostlong,
  [out] u_long *lpnetlong
);
```

Parameters

`[in] s`

A descriptor identifying a socket.

`[in] hostlong`

A 32-bit number in host byte order.

`[out] lpnetlong`

A pointer to a 32-bit number to receive the number in network byte order.

Return value

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

ERROR CODE	MEANING
WSANOTINITIALISED	A successful WSAStartup call must occur before using this function.
WSAENETDOWN	The network subsystem has failed.
WSAENOTSOCK	The descriptor is not a socket.
WSAEFAULT	The *lpnetlong* parameter is **NULL** or the address pointed to is not completely contained in a valid part of the user address space.

Remarks

The **WSAHtonl** function takes a 32-bit number in host byte order and returns a 32-bit number in network byte order in the 32-bit number pointed to by the *lpnetlong* parameter. The socket passed in the *s* parameter is used

to determine the network byte order required based on the Winsock catalog protocol entry associated with the socket. This feature supports Winsock providers that use different network byte orders.

If the socket is for the AF_INET or AF_INET6 address family, the **WSAHtonl** function can be used to convert an IPv4 address in host byte order to the IPv4 address in network byte order. This function does not do any checking to determine if the *hostlong* parameter is a valid IPv4 address.

The **WSAHtonl** function requires that the Winsock DLL has previously been loaded with a successful call to the WSAStartup function. For use with the AF_INET or AF_INET6 family, the htonl function does not require that the Winsock DLL be loaded.

Windows Phone 8: This function is supported for Windows Phone Store apps on Windows Phone 8 and later.

Windows 8.1 and **Windows Server 2012 R2**: This function is supported for Windows Store apps on Windows 8.1, Windows Server 2012 R2, and later.

Requirements

Minimum supported client	Windows 8.1, Windows Vista [desktop apps | UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps | UWP apps]
Target Platform	Windows
Header	winsock2.h
Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

InetNtop

WSAHtons

WSANtohl

WSANtohs

Winsock Functions

Winsock Reference

htonl

htons

inet_addr

inet_ntoa

ntohl

ntohs

WSAHtons function (winsock2.h)

10/22/2022 • 2 minutes to read • Edit Online

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

Syntax

```
int WSAAPI WSAHtons(
  [in]  SOCKET  s,
  [in]  u_short hostshort,
  [out] u_short *lpnetshort
);
```

Parameters

`[in] s`

A descriptor identifying a socket.

`[in] hostshort`

A 16-bit number in host byte order.

`[out] lpnetshort`

A pointer to a 16-bit buffer to receive the number in network byte order.

Return value

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

ERROR CODE	MEANING
WSANOTINITIALISED	A successful WSAStartup call must occur before using this function.
WSAENETDOWN	The network subsystem has failed.
WSAENOTSOCK	The descriptor is not a socket.
WSAEFAULT	The *lpnetshort* parameter is **NULL** or the address pointed to is not completely contained in a valid part of the user address space.

Remarks

The **WSAHtons** function takes a 16-bit number in host byte order and returns a 16-bit number in network byte order in the 16-bit number pointed to by the *lpnetshort* parameter. The socket passed in the *s* parameter is used

to determine the network byte order required based on the Winsock catalog protocol entry associated with the socket. This feature supports Winsock providers that use different network byte orders.

If the socket is for the AF_INET or AF_INET6 address family, the **WSAHtons** function can be used to convert an IP port number in host byte order to the IP port number in network byte order.

The **WSAHtons** function requires that the Winsock DLL has previously been loaded with a successful call to the WSAStartup function. For use with the AF_INET OR AF_INET6 address family, the htons function does not require that the Winsock DLL be loaded.

Windows Phone 8: This function is supported for Windows Phone Store apps on Windows Phone 8 and later.

Windows 8.1 and **Windows Server 2012 R2**: This function is supported for Windows Store apps on Windows 8.1, Windows Server 2012 R2, and later.

Requirements

Minimum supported client	Windows 8.1, Windows Vista [desktop apps | UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps | UWP apps]
Target Platform	Windows
Header	winsock2.h
Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

InetNtop

WSAHtonl

WSANtohl

WSANtohs

Winsock Functions

Winsock Reference

htonl

htons

inet_addr

inet_ntoa

ntohl

ntohs

WSAInstallServiceClassA function (winsock2.h)

10/22/2022 • 2 minutes to read • Edit Online

The **WSAInstallServiceClass** function registers a service class schema within a namespace. This schema includes the class name, class identifier, and any namespace-specific information that is common to all instances of the service, such as the SAP identifier or object identifier.

Syntax

```
INT WSAAPI WSAInstallServiceClassA(
  [in] LPWSASERVICECLASSINFOA lpServiceClassInfo
);
```

Parameters

`[in] lpServiceClassInfo`

Service class to namespace specific–type mapping information. Multiple mappings can be handled at one time.

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

Return value

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

ERROR CODE	MEANING
WSA_INVALID_PARAMETER	The namespace provider cannot supply the requested class information.
WSA_NOT_ENOUGH_MEMORY	There was insufficient memory to perform the operation.
WSAEACCES	The calling function does not have sufficient privileges to install the service.
WSAEALREADY	Service class information has already been registered for this service class identifier. To modify service class information, first use WSARemoveServiceClass, and then reinstall with updated class information data.
WSAEINVAL	The service class information was not valid or improperly structured. This error is returned if the *lpServiceClassInfo* parameter is **NULL**.
WSAEOPNOTSUPP	The operation is not supported. This error is returned if the namespace provider does not implement this function.

WSANO_DATA	The requested name is valid, but no data of the requested type was found.
WSANOTINITIALISED	The WS2_32.DLL has not been initialized. The application must first call WSAStartup before calling any Windows Sockets functions.

Remarks

> **NOTE**
>
> The winsock2.h header defines WSAInstallServiceClass as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see Conventions for Function Prototypes.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winsock2.h
Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

WSAGetLastError

WSAGetServiceClassInfo

WSASERVICECLASSINFO

Winsock Functions

Winsock Reference

WSAInstallServiceClassW function (winsock2.h)

10/22/2022 • 2 minutes to read • Edit Online

The **WSAInstallServiceClass** function registers a service class schema within a namespace. This schema includes the class name, class identifier, and any namespace-specific information that is common to all instances of the service, such as the SAP identifier or object identifier.

Syntax

```
INT WSAAPI WSAInstallServiceClassW(
  [in] LPWSASERVICECLASSINFOW lpServiceClassInfo
);
```

Parameters

`[in] lpServiceClassInfo`

Service class to namespace specific–type mapping information. Multiple mappings can be handled at one time.

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

Return value

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

ERROR CODE	MEANING
WSA_INVALID_PARAMETER	The namespace provider cannot supply the requested class information.
WSA_NOT_ENOUGH_MEMORY	There was insufficient memory to perform the operation.
WSAEACCES	The calling function does not have sufficient privileges to install the service.
WSAEALREADY	Service class information has already been registered for this service class identifier. To modify service class information, first use WSARemoveServiceClass, and then reinstall with updated class information data.
WSAEINVAL	The service class information was not valid or improperly structured. This error is returned if the *lpServiceClassInfo* parameter is **NULL**.
WSAEOPNOTSUPP	The operation is not supported. This error is returned if the namespace provider does not implement this function.

WSANO_DATA	The requested name is valid, but no data of the requested type was found.
WSANOTINITIALISED	The WS2_32.DLL has not been initialized. The application must first call WSAStartup before calling any Windows Sockets functions.

Remarks

> **NOTE**
>
> The winsock2.h header defines WSAInstallServiceClass as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see Conventions for Function Prototypes.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winsock2.h
Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

WSAGetLastError

WSAGetServiceClassInfo

WSASERVICECLASSINFO

Winsock Functions

Winsock Reference

WSAIoctl function (winsock2.h)

10/22/2022 • 8 minutes to read • Edit Online

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

Syntax

```
int WSAAPI WSAIoctl(
  [in]  SOCKET                             s,
  [in]  DWORD                              dwIoControlCode,
  [in]  LPVOID                             lpvInBuffer,
  [in]  DWORD                              cbInBuffer,
  [out] LPVOID                             lpvOutBuffer,
  [in]  DWORD                              cbOutBuffer,
  [out] LPDWORD                            lpcbBytesReturned,
  [in]  LPWSAOVERLAPPED                    lpOverlapped,
  [in]  LPWSAOVERLAPPED_COMPLETION_ROUTINE lpCompletionRoutine
);
```

Parameters

`[in] s`

A descriptor identifying a socket.

`[in] dwIoControlCode`

The control code of operation to perform.

`[in] lpvInBuffer`

A pointer to the input buffer.

`[in] cbInBuffer`

The size, in bytes, of the input buffer.

`[out] lpvOutBuffer`

A pointer to the output buffer.

`[in] cbOutBuffer`

The size, in bytes, of the output buffer.

`[out] lpcbBytesReturned`

A pointer to actual number of bytes of output.

`[in] lpOverlapped`

A pointer to a WSAOVERLAPPED structure (ignored for non-overlapped sockets).

`[in] lpCompletionRoutine`

Type: _In_opt_ **LPWSAOVERLAPPED_COMPLETION_ROUTINE**

> **Note** A pointer to the completion routine called when the operation has been completed (ignored for non-overlapped sockets). See Remarks.

Return value

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

ERROR CODE	MEANING
WSA_IO_PENDING	An overlapped operation was successfully initiated and completion will be indicated at a later time.
WSAENETDOWN	The network subsystem has failed.
WSAEFAULT	The *lpvInBuffer*, *lpvOutBuffer*, *lpcbBytesReturned*, *lpOverlapped*, or *lpCompletionRoutine* parameter is not totally contained in a valid part of the user address space, or the *cbInBuffer* or *cbOutBuffer* parameter is too small.
WSAEINVAL	The *dwIoControlCode* 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.
WSAEINPROGRESS	The function is invoked when a callback is in progress.
WSAENOTSOCK	The descriptor *s* is not a socket.
WSAEOPNOTSUPP	The specified IOCTL command cannot be realized. (For example, the FLOWSPEC structures specified in **SIO_SET_QOS** or **SIO_SET_GROUP_QOS** cannot be satisfied.)
WSAEWOULDBLOCK	The socket is marked as non-blocking and the requested operation would block.
WSAENOPROTOOPT	The socket option is not supported on the specified protocol. For example, an attempt to use the **SIO_GET_BROADCAST_ADDRESS** IOCTL was made on an IPv6 socket or an attempt to use the TCP **SIO_KEEPALIVE_VALS** IOCTL was made on a datagram socket.

Remarks

The **WSAIoctl** function is used to set or retrieve operating parameters associated with the socket, the transport protocol, or the communications subsystem.

If both *lpOverlapped* and *lpCompletionRoutine* are **NULL**, the socket in this function will be treated as a non-

overlapped socket. For a non-overlapped socket, *lpOverlapped* and *lpCompletionRoutine* parameters are ignored, which causes the function to behave like the standard ioctlsocket function except that the function can block if socket *s* is in blocking mode. If socket *s* is in non-blocking mode, this function can return WSAEWOULDBLOCK when the specified operation cannot be finished immediately. In this case, the application may change the socket to blocking mode and reissue the request or wait for the corresponding network event (such as FD_ROUTING_INTERFACE_CHANGE or FD_ADDRESS_LIST_CHANGE in the case of **SIO_ROUTING_INTERFACE_CHANGE** or **SIO_ADDRESS_LIST_CHANGE**) using a Windows message (using WSAAsyncSelect)-based or event (using WSAEventSelect)-based notification mechanism.

For overlapped sockets, operations that cannot be completed immediately will be initiated, and completion will be indicated at a later time. The **DWORD** value pointed to by the *lpcbBytesReturned* parameter that is returned may be ignored. The final completion status and bytes returned can be retrieved when the appropriate completion method is signaled when the operation has completed.

Any IOCTL may block indefinitely, depending on the service provider's implementation. If the application cannot tolerate blocking in a **WSAIoctl** call, overlapped I/O would be advised for IOCTLs that are especially likely to block including:

SIO_ADDRESS_LIST_CHANGE

SIO_FINDROUTE

SIO_FLUSH

SIO_GET_QOS

SIO_GET_GROUP_QOS

SIO_ROUTING_INTERFACE_CHANGE

SIO_SET_QOS

SIO_SET_GROUP_QOS

Some protocol-specific IOCTLs may also be especially likely to block. Check the relevant protocol-specific annex for any available information.

The prototype for the completion routine pointed to by the *lpCompletionRoutine* parameter is as follows:

```
#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")


void CALLBACK CompletionRoutine (
  IN DWORD dwError,
  IN DWORD cbTransferred,
  IN LPWSAOVERLAPPED lpOverlapped,
  IN DWORD dwFlags
);
```

The CompletionRoutine is a placeholder for an application-supplied function name. The *dwError* parameter

specifies the completion status for the overlapped operation as indicated by *lpOverlapped* parameter. The *cbTransferred* parameter specifies the number of bytes received. The *dwFlags* parameter is not used for this IOCTL. The completion routine does not return a value.

It is possible to adopt an encoding scheme that preserves the currently defined ioctlsocket opcodes while providing a convenient way to partition the opcode identifier space in as much as the *dwIoControlCode* parameter is now a 32-bit entity. The *dwIoControlCode* parameter is 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 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.

The following Unix IOCTL codes (commands) are supported.

The following Windows Sockets 2 commands are supported.

If an overlapped operation completes immediately, **WSAIoctl** returns a value of zero and the *lpcbBytesReturned* parameter is updated with the number of bytes in the output buffer. If the overlapped operation is successfully

initiated and will complete later, this function returns SOCKET_ERROR and indicates error code WSA_IO_PENDING. In this case, *lpcbBytesReturned* is not updated. When the overlapped operation completes the amount of data in the output buffer is indicated either through the *cbTransferred* parameter in the completion routine (if specified), or through the *lpcbTransfer* parameter in WSAGetOverlappedResult.

When called with an overlapped socket, the *lpOverlapped* parameter must be valid for the duration of the overlapped operation. The *lpOverlapped* parameter contains the address of a WSAOVERLAPPED structure.

If the *lpCompletionRoutine* parameter is **NULL**, the *hEvent* parameter of *lpOverlapped* is signaled when the overlapped operation completes if it contains a valid event object handle. An application can use WSAWaitForMultipleEvents or WSAGetOverlappedResult to wait or poll on the event object.

> **Note** All I/O initiated by a given thread is canceled when that thread exits. For overlapped sockets, pending asynchronous operations can fail if the thread is closed before the operations complete. See ExitThread for more information.

If *lpCompletionRoutine* is not **NULL**, the *hEvent* parameter is ignored and can be used by the application to pass context information to the completion routine. A caller that passes a non-**NULL** *lpCompletionRoutine* and later calls WSAGetOverlappedResult for the same overlapped I/O request may not set the *fWait* parameter for that invocation of **WSAGetOverlappedResult** to **TRUE**. In this case, the usage of the *hEvent* parameter is undefined, and attempting to wait on the *hEvent* parameter would produce unpredictable results.

The prototype of the completion routine is as follows:

```
void CALLBACK CompletionRoutine(
  IN DWORD dwError,
  IN DWORD cbTransferred,
  IN LPWSAOVERLAPPED lpOverlapped,
  IN DWORD dwFlags
);
```

This **CompletionRoutine** is a placeholder for an application-defined or library-defined function. The completion routine is invoked only if the thread is in an alertable state. To put a thread into an alertable state, use the WSAWaitForMultipleEvents, WaitForSingleObjectEx, or WaitForMultipleObjectsEx function with the *fAlertable* or *bAlertable* parameter set to **TRUE**.

The *dwError* parameter of **CompletionRoutine** specifies the completion status for the overlapped operation as indicated by *lpOverlapped*. The *cbTransferred* parameter specifies the number of bytes returned. Currently, no flag values are defined and *dwFlags* will be zero. The **CompletionRoutine** function does not return a value.

Returning from this function allows invocation of another pending completion routine for this socket. The completion routines can be called in any order, not necessarily in the same order the overlapped operations are completed.

Compatibility

The IOCTL codes with T == 0 are a subset of the IOCTL codes used in Berkeley sockets. In particular, there is no command that is equivalent to **FIOASYNC**.

> **Note** Some IOCTL codes require additional header files. For example, use of the **SIO_RCVALL** IOCTL requires the Mstcpip.h header file.

Windows Phone 8: This function is supported for Windows Phone Store apps on Windows Phone 8 and later.

Windows 8.1 and **Windows Server 2012 R2**: This function is supported for Windows Store apps on

Windows 8.1, Windows Server 2012 R2, and later.

Requirements

Minimum supported client	Windows 8.1, Windows Vista [desktop apps | UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps | UWP apps]
Target Platform	Windows
Header	winsock2.h
Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

SOL_SOCKET Socket Options

WSASocket

Winsock Functions

Winsock Reference

getsockopt

ioctlsocket

setsockopt

socket

WSAIsBlocking function (winsock2.h)

10/22/2022 • 2 minutes to read • Edit Online

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

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

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

Syntax

```
BOOL WSAAPI WSAIsBlocking();
```

Return value

None

Requirements

Target Platform	Windows
Header	winsock2.h

See also

Winsock Functions

Winsock Reference

WSAJoinLeaf function (winsock2.h)

10/22/2022 • 10 minutes to read • Edit Online

The **WSAJoinLeaf** function joins a leaf node into a multipoint session, exchanges connect data, and specifies needed quality of service based on the specified FLOWSPEC structures.

Syntax

```
SOCKET WSAAPI WSAJoinLeaf(
  [in]  SOCKET         s,
  [in]  const sockaddr *name,
  [in]  int            namelen,
  [in]  LPWSABUF       lpCallerData,
  [out] LPWSABUF       lpCalleeData,
  [in]  LPQOS          lpSQOS,
  [in]  LPQOS          lpGQOS,
  [in]  DWORD          dwFlags
);
```

Parameters

`[in] s`

Descriptor identifying a multipoint socket.

`[in] name`

Name of the peer to which the socket is to be joined.

`[in] namelen`

Length of *name*, in bytes.

`[in] lpCallerData`

Pointer to the user data that is to be transferred to the peer during multipoint session establishment.

`[out] lpCalleeData`

Pointer to the user data that is to be transferred back from the peer during multipoint session establishment.

`[in] lpSQOS`

Pointer to the FLOWSPEC structures for socket *s*, one for each direction.

`[in] lpGQOS`

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

`[in] dwFlags`

Flags to indicate that the socket is acting as a sender (JL_SENDER_ONLY), receiver (JL_RECEIVER_ONLY), or both (JL_BOTH).

Return value

If no error occurs, **WSAJoinLeaf** returns a value of type SOCKET that is a descriptor for the newly created multipoint socket. Otherwise, a value of INVALID_SOCKET is returned, and a specific error code can be retrieved by calling WSAGetLastError.

On a blocking socket, the return value indicates success or failure of the join operation.

With a nonblocking socket, successful initiation of a join operation is indicated by a return of a valid socket descriptor. Subsequently, an FD_CONNECT indication will be given on the original socket *s* when the join operation completes, either successfully or otherwise. The application must use either WSAAsyncSelect or WSAEventSelect with interest registered for the FD_CONNECT event in order to determine when the join operation has completed and checks the associated error code to determine the success or failure of the operation. The select function cannot be used to determine when the join operation completes.

Also, until the multipoint session join attempt completes all subsequent calls to **WSAJoinLeaf** on the same socket will fail with the error code WSAEALREADY. After the **WSAJoinLeaf** operation completes successfully, a subsequent attempt will usually fail with the error code WSAEISCONN. An exception to the WSAEISCONN rule occurs for a c_root socket that allows root-initiated joins. In such a case, another join may be initiated after a prior **WSAJoinLeaf** operation completes.

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

ERROR CODE	MEANING
WSANOTINITIALISED	A successful WSAStartup call must occur before using this function.
WSAEADDRINUSE	The socket's local address is already in use and the socket was not marked to allow address reuse with SO_REUSEADDR. This error usually occurs at the time of bind, but could be delayed until this function if the **bind** was to a partially wildcard address (involving ADDR_ANY) and if a specific address needs to be committed at the time of this function.
WSAEADDRNOTAVAIL	The remote address is not a valid address (such as ADDR_ANY).
WSAEAFNOSUPPORT	Addresses in the specified family cannot be used with this socket.
WSAEALREADY	A nonblocking WSAJoinLeaf call is in progress on the specified socket.
WSAECONNREFUSED	The attempt to join was forcefully rejected.
WSAEFAULT	The *name* or the *namelen* parameter is not a valid part of the user address space, the *namelen* parameter is too small, the buffer length for *lpCalleeData*, *lpSQOS*, and *lpGQOS* are too small, or the buffer length for *lpCallerData* is too large.

WSAEINVAL	A WSAJoinLeaf function call was performed on a UDP socket that was opened without setting its WSA_FLAG_MULTIPOINT_C_LEAF or WSA_FLAG_MULTIPOINT_D_LEAF multipoint flag.
WSAEISCONN	The socket is already a member of the multipoint session.
WSAEINTR	A blocking Windows Socket 1.1 call was canceled through WSACancelBlockingCall.
WSAEINPROGRESS	A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function.
WSAENETDOWN	The network subsystem has failed.
WSAENETUNREACH	The network cannot be reached from this host at this time.
WSAENOBUFS	No buffer space is available. The socket cannot be joined.
WSAENOTSOCK	The descriptor is not a socket.
WSAEOPNOTSUPP	The FLOWSPEC structures specified in *lpSQOS* and *lpGQOS* cannot be satisfied.
WSAEPROTONOSUPPORT	The *lpCallerData* augment is not supported by the service provider.
WSAETIMEDOUT	The attempt to join timed out without establishing a multipoint session.

Remarks

The **WSAJoinLeaf** function is used to join a leaf node to a multipoint session, and to perform a number of other ancillary operations that occur at session join time as well. If the socket *s* is unbound, unique values are assigned to the local association by the system, and the socket is marked as bound.

The **WSAJoinLeaf** function has the same parameters and semantics as WSAConnect except that it returns a socket descriptor (as in WSAAccept), and it has an additional *dwFlags* parameter. Only multipoint sockets created using WSASocket with appropriate multipoint flags set can be used for input parameter *s* in this function. The returned socket descriptor will not be usable until after the join operation completes. For example, if the socket is in nonblocking mode after a corresponding FD_CONNECT indication has been received from WSAAsyncSelect or WSAEventSelect on the original socket *s*, except that closesocket may be invoked on this new socket descriptor to cancel a pending join operation. A root application in a multipoint session may call **WSAJoinLeaf** one or more times in order to add a number of leaf nodes, however at most one multipoint connection request may be outstanding at a time. Refer to Multipoint and Multicast Semantics for additional information.

For nonblocking sockets it is often not possible to complete the connection immediately. In such a case, this function returns an as-yet unusable socket descriptor and the operation proceeds. There is no error code such as WSAEWOULDBLOCK in this case, since the function has effectively returned a successful start indication. When the final outcome success or failure becomes known, it may be reported through WSAAsyncSelect or WSAEventSelect depending on how the client registers for notification on the original socket *s*. In either case, the notification is announced with FD_CONNECT and the error code associated with the FD_CONNECT indicates either success or a specific reason for failure. The select function cannot be used to detect completion notification for **WSAJoinLeaf**.

The socket descriptor returned by **WSAJoinLeaf** is different depending on whether the input socket descriptor, *s*, is a c_root or a c_leaf. When used with a c_root socket, the *name* parameter designates a particular leaf node to be added and the returned socket descriptor is a c_leaf socket corresponding to the newly added leaf node. The newly created socket has the same properties as *s*, including asynchronous events registered with WSAAsyncSelect or with WSAEventSelect. It is not intended to be used for exchange of multipoint data, but rather is used to receive network event indications (for example, FD_CLOSE) for the connection that exists to the particular c_leaf. Some multipoint implementations can also allow this socket to be used for side chats between the root and an individual leaf node. An FD_CLOSE indication will be received for this socket if the corresponding leaf node calls closesocket to drop out of the multipoint session. Symmetrically, invoking **closesocket** on the c_leaf socket returned from **WSAJoinLeaf** will cause the socket in the corresponding leaf node to get 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 other words, a new socket descriptor is not allocated. In a rooted control scheme, the root application would put its c_root socket in listening mode by calling listen. The standard FD_ACCEPT notification will be delivered when the leaf node requests to join itself to the multipoint session. The root application uses the usual accept or WSAAccept functions to admit the new leaf node. The value returned from either **accept** or **WSAAccept** is also a c_leaf socket descriptor just like those returned from **WSAJoinLeaf**. To accommodate multipoint schemes that allow both root-initiated and leaf-initiated joins, it is acceptable for a c_root socket that is already in listening mode to be used as an input to **WSAJoinLeaf**.

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

The *lpCallerData* is a value parameter that contains any user data that is to be sent along with the multipoint session join request. If *lpCallerData* is **NULL**, no user data will be passed to the peer. The *lpCalleeData* is a result parameter that will contain any user data passed back from the peer as part of the multipoint session establishment. The **len** member of the WSABUF structure pointed to by the *lpCalleeData* parameter initially contains the length of the buffer allocated by the application and pointed to by the **buf** member of the **WSABUF** structure. The **len** member of the **WSABUF** structure pointed to by the *lpCalleeData* parameter will be set to zero if no user data has been passed back. The *lpCalleeData* information will be valid when the multipoint join operation is complete.

For blocking sockets, this will be when the **WSAJoinLeaf** function returns. For nonblocking sockets, this will be after the join operation has completed. For example, this could occur after FD_CONNECT notification on the original socket *s*). If *lpCalleeData* is **NULL**, no user data will be passed back. The exact format of the user data is specific to the address family to which the socket belongs.

At multipoint session establishment time, an application can use the *lpSQOS* and/or *lpGQOS* parameters to override any previous quality of service specification made for the socket through WSAIoctl with the SIO_SET_QOS or SIO_SET_GROUP_QOS opcodes.

The *lpSQOS* parameter specifies the FLOWSPEC structures for socket *s*, one for each direction, followed by any additional provider-specific parameters. If either the associated transport provider in general or the specific type

of socket in particular cannot honor the quality of service request, an error will be returned as indicated in the following. The respective sending or receiving flow specification values will be ignored for any unidirectional sockets. If no provider-specific parameters are specified, the **buf** and **len** members of the WSABUF structure pointed to by the *lpCalleeData* parameter should be set to **NULL** and zero, respectively. A **NULL** value for *lpSQOS* indicates no application-supplied quality of service.

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

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

> **Note** When issuing a blocking Winsock call such as **WSAJoinLeaf**, Winsock may need to wait for a network event before the call can complete. Winsock performs an alertable wait in this situation, which can be interrupted by an asynchronous procedure call (APC) scheduled on the same thread. Issuing another blocking Winsock call inside an APC that interrupted an ongoing blocking Winsock call on the same thread will lead to undefined behavior, and must never be attempted by Winsock clients.

Windows Phone 8: This function is supported for Windows Phone Store apps on Windows Phone 8 and later.

Windows 8.1 and **Windows Server 2012 R2**: This function is supported for Windows Store apps on Windows 8.1, Windows Server 2012 R2, and later.

Requirements

Minimum supported client	Windows 8.1, Windows Vista [desktop apps | UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps | UWP apps]
Target Platform	Windows
Header	winsock2.h
Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

WSAAccept

WSAAsyncSelect

WSABUF

WSAEventSelect

WSASocket

Winsock Functions

Winsock Reference

accept

bind

select

WSALookupServiceBeginA function (winsock2.h)

10/22/2022 • 6 minutes to read • Edit Online

The **WSALookupServiceBegin** function initiates a client query that is constrained by the information contained within a WSAQUERYSET structure. **WSALookupServiceBegin** only returns a handle, which should be used by subsequent calls to WSALookupServiceNext to get the actual results.

Syntax

```
INT WSAAPI WSALookupServiceBeginA(
  [in]  LPWSAQUERYSETA lpqsRestrictions,
  [in]  DWORD          dwControlFlags,
  [out] LPHANDLE       lphLookup
);
```

Parameters

`[in] lpqsRestrictions`

A pointer to the search criteria. See the Remarks for details.

`[in] dwControlFlags`

A set of flags that controls the depth of the search.

Supported values for the *dwControlFlags* parameter are defined in the *Winsock2.h* header file and can be a combination of the following options.

FLAG	MEANING
LUP_DEEP 0x0001	Queries deep as opposed to just the first level.
LUP_CONTAINERS 0x0002	Returns containers only.
LUP_NOCONTAINERS 0x0004	Do not return containers.
LUP_NEAREST 0x0008	If possible, returns results in the order of distance. The measure of distance is provider specific.
LUP_RETURN_NAME 0x0010	Retrieves the name as *lpszServiceInstanceName*.

LUP_RETURN_TYPE 0x0020	Retrieves the type as *lpServiceClassId*.
LUP_RETURN_VERSION 0x0040	Retrieves the version as *lpVersion*.
LUP_RETURN_COMMENT 0x0080	Retrieves the comment as *lpszComment*.
LUP_RETURN_ADDR 0x0100	Retrieves the addresses as *lpcsaBuffer*.
LUP_RETURN_BLOB 0x0200	Retrieves the private data as *lpBlob*.
LUP_RETURN_ALIASES 0x0400	Any available alias information is to be returned in successive calls to WSALookupServiceNext, and each alias returned will have the RESULT_IS_ALIAS flag set.
LUP_RETURN_QUERY_STRING 0x0800	Retrieves the query string used for the request.
LUP_RETURN_ALL 0x0FF0	A set of flags that retrieves all of the LUP_RETURN_* values.
LUP_FLUSHPREVIOUS 0x1000	Used as a value for the *dwControlFlags* parameter in WSALookupServiceNext. Setting this flag instructs the provider to discard the last result set, which was too large for the specified buffer, and move on to the next result set.
LUP_FLUSHCACHE 0x2000	If the provider has been caching information, ignores the cache, and queries the namespace itself.
LUP_RES_SERVICE 0x8000	This indicates whether prime response is in the remote or local part of CSADDR_INFO structure. The other part needs to be usable in either case.

`[out] lphLookup`

A handle to be used when calling WSALookupServiceNext in order to start retrieving the results set.

Return value

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

ERROR CODE	MEANING
WSA_NOT_ENOUGH_MEMORY	There was insufficient memory to perform the operation.
WSAEINVAL	One or more parameters were missing or invalid for this provider.
WSANO_DATA	The name was found in the database but no data matching the given restrictions was located.
WSANOTINITIALISED	The WS2_32.DLL has not been initialized. The application must first call WSAStartup before calling any Windows Sockets functions.
WSASERVICE_NOT_FOUND	No such service is known. The service cannot be found in the specified name space. This error is returned for a bluetooth service discovery request if no remote bluetooth devices were found.

Remarks

The *lpqsRestrictions* parameter points to a buffer containing a WSAQUERYSET structure. At a minimum, the **dwSize** member of the **WSAQUERYSET** must be set to the length of the buffer before calling the **WSALookupServiceBegin** function. Applications can restrict the query by specifying other members in the **WSAQUERYSET**.

In most instances, applications interested in only a particular transport protocol should constrain their query by address family and protocol using the **dwNumberOfProtocols** and **lpafpProtocols** members of the WSAQUERYSET rather than by specifiying the namespace in the **dwNameSpace** member.

Information on supported network transport protocols can be retreived using the EnumProtocols, WSAEnumProtocols, WSCEnumProtocols, or WSCEnumProtocols32 function.

It is also possible to constrain the query to a single namespace. For example, a query that only wants results from DNS (not results from the local hosts file and other naming services) would set the **dwNameSpace** member to NS_DNS. For example, a bluetooth device discovery would set the the **dwNameSpace** member to NS_BTH.

Applications can also restrict the query to a specific namespace provider by specifying a pointer to the GUID for the provider in the **lpNSProviderId** member.

Information on namespace providers on the local computer can be retrieved using the WSAEnumNameSpaceProviders, WSAEnumNameSpaceProvidersEx, WSCEnumNameSpaceProviders32, or WSCEnumNameSpaceProvidersEx32 function.

If LUP_CONTAINERS is specified in a call, other restriction values should be avoided. If any are specified, it is up to the name service provider to decide if it can support this restriction over the containers. If it cannot, it should return an error.

Some name service providers can have other means of finding containers. For example, containers might all be of some well-known type, or of a set of well-known types, and therefore a query restriction can be created for finding them. No matter what other means the name service provider has for locating containers, LUP_CONTAINERS and LUP_NOCONTAINERS take precedence. Hence, if a query restriction is given that includes containers, specifying LUP_NOCONTAINERS will prevent the container items from being returned. Similarly, no matter the query restriction, if LUP_CONTAINERS is given, only containers should be returned. If a

namespace does not support containers, and LUP_CONTAINERS is specified, it should simply return WSANO_DATA.

The preferred method of obtaining the containers within another container, is the call:

```
dwStatus = WSALookupServiceBegin(
      lpqsRestrictions,
      LUP_CONTAINERS,
      lphLookup);
```

This call is followed by the requisite number of WSALookupServiceNext calls. This will return all containers contained immediately within the starting context; that is, it is not a deep query. With this, one can map the address space structure by walking the hierarchy, perhaps enumerating the content of selected containers. Subsequent uses of **WSALookupServiceBegin** use the containers returned from a previous call.

As mentioned above, a WSAQUERYSET structure is used as an input parameter to **WSALookupBegin** in order to qualify the query. The following table indicates how the **WSAQUERYSET** is used to construct a query. When a parameter is marked as (Optional) a **NULL** pointer can be specified, indicating that the parameter will not be used as a search criteria. See section Query-Related Data Structures for additional information.

WSAQUERYSET MEMBER	QUERY INTERPRETATION
dwSize	Must be set to sizeof(WSAQUERYSET). This is a versioning mechanism.
dwOutputFlags	Ignored for queries.
lpszServiceInstanceName	(Optional) Referenced string contains service name. The semantics for wildcarding within the string are not defined, but can be supported by certain namespace providers.
lpServiceClassId	(Required) The GUID corresponding to the service class.
lpVersion	(Optional) References desired version number and provides version comparison semantics (that is, version must match exactly, or version must be not less than the value specified).
lpszComment	Ignored for queries.
dwNameSpace See the Important note that follows.	Identifier of a single namespace in which to constrain the search, or NS_ALL to include all namespaces.
lpNSProviderId	(Optional) References the GUID of a specific namespace provider, and limits the query to this provider only.
lpszContext	(Optional) Specifies the starting point of the query in a hierarchical namespace.
dwNumberOfProtocols	Size of the protocol constraint array, can be zero.
lpafpProtocols	(Optional) References an array of AFPROTOCOLS structure. Only services that utilize these protocols will be returned.

lpszQueryString	(Optional) Some namespaces (such as whois++) support enriched SQL-like queries that are contained in a simple text string. This parameter is used to specify that string.
dwNumberOfCsAddrs	Ignored for queries.
lpcsaBuffer	Ignored for queries.
lpBlob	(Optional) This is a pointer to a provider-specific entity.

> **Important** In most instances, applications interested in only a particular transport protocol should constrain their query by address family and protocol rather than by namespace. This would allow an application that needs to locate a TCP/IP service, for example, to have its query processed by all available namespaces such as the local hosts file, DNS, and NIS.

Windows Phone 8: The **WSALookupServiceBeginW** function is supported for Windows Phone Store apps on Windows Phone 8 and later.

Windows 8.1 and **Windows Server 2012 R2**: The **WSALookupServiceBeginW** function is supported for Windows Store apps on Windows 8.1, Windows Server 2012 R2, and later.

> **NOTE**
> The winsock2.h header defines WSALookupServiceBegin as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see Conventions for Function Prototypes.

Requirements

Minimum supported client	Windows 8.1, Windows Vista [desktop apps | UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps | UWP apps]
Target Platform	Windows
Header	winsock2.h
Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

Bluetooth and WSALookupServiceBegin

EnumProtocols

WSAEnumNameSpaceProviders

WSAEnumNameSpaceProvidersEx

WSAEnumProtocols

WSALookupServiceEnd

WSALookupServiceNext

WSAQUERYSET

WSCEnumNameSpaceProviders32

WSCEnumNameSpaceProvidersEx32

WSCEnumProtocols

WSCEnumProtocols32

WSALookupServiceBeginW function (winsock2.h)

10/22/2022 • 6 minutes to read • Edit Online

The **WSALookupServiceBegin** function initiates a client query that is constrained by the information contained within a WSAQUERYSET structure. **WSALookupServiceBegin** only returns a handle, which should be used by subsequent calls to WSALookupServiceNext to get the actual results.

Syntax

```
INT WSAAPI WSALookupServiceBeginW(
  [in]  LPWSAQUERYSETW lpqsRestrictions,
  [in]  DWORD          dwControlFlags,
  [out] LPHANDLE       lphLookup
);
```

Parameters

`[in] lpqsRestrictions`

A pointer to the search criteria. See the Remarks for details.

`[in] dwControlFlags`

A set of flags that controls the depth of the search.

Supported values for the *dwControlFlags* parameter are defined in the *Winsock2.h* header file and can be a combination of the following options.

FLAG	MEANING
LUP_DEEP 0x0001	Queries deep as opposed to just the first level.
LUP_CONTAINERS 0x0002	Returns containers only.
LUP_NOCONTAINERS 0x0004	Do not return containers.
LUP_NEAREST 0x0008	If possible, returns results in the order of distance. The measure of distance is provider specific.
LUP_RETURN_NAME 0x0010	Retrieves the name as *lpszServiceInstanceName*.

LUP_RETURN_TYPE 0x0020	Retrieves the type as *lpServiceClassId*.
LUP_RETURN_VERSION 0x0040	Retrieves the version as *lpVersion*.
LUP_RETURN_COMMENT 0x0080	Retrieves the comment as *lpszComment*.
LUP_RETURN_ADDR 0x0100	Retrieves the addresses as *lpcsaBuffer*.
LUP_RETURN_BLOB 0x0200	Retrieves the private data as *lpBlob*.
LUP_RETURN_ALIASES 0x0400	Any available alias information is to be returned in successive calls to WSALookupServiceNext, and each alias returned will have the RESULT_IS_ALIAS flag set.
LUP_RETURN_QUERY_STRING 0x0800	Retrieves the query string used for the request.
LUP_RETURN_ALL 0x0FF0	A set of flags that retrieves all of the LUP_RETURN_* values.
LUP_FLUSHPREVIOUS 0x1000	Used as a value for the *dwControlFlags* parameter in WSALookupServiceNext. Setting this flag instructs the provider to discard the last result set, which was too large for the specified buffer, and move on to the next result set.
LUP_FLUSHCACHE 0x2000	If the provider has been caching information, ignores the cache, and queries the namespace itself.
LUP_RES_SERVICE 0x8000	This indicates whether prime response is in the remote or local part of CSADDR_INFO structure. The other part needs to be usable in either case.

`[out] lphLookup`

A handle to be used when calling WSALookupServiceNext in order to start retrieving the results set.

Return value

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

ERROR CODE	MEANING
WSA_NOT_ENOUGH_MEMORY	There was insufficient memory to perform the operation.
WSAEINVAL	One or more parameters were missing or invalid for this provider.
WSANO_DATA	The name was found in the database but no data matching the given restrictions was located.
WSANOTINITIALISED	The WS2_32.DLL has not been initialized. The application must first call WSAStartup before calling any Windows Sockets functions.
WSASERVICE_NOT_FOUND	No such service is known. The service cannot be found in the specified name space. This error is returned for a bluetooth service discovery request if no remote bluetooth devices were found.

Remarks

The *lpqsRestrictions* parameter points to a buffer containing a WSAQUERYSET structure. At a minimum, the **dwSize** member of the **WSAQUERYSET** must be set to the length of the buffer before calling the **WSALookupServiceBegin** function. Applications can restrict the query by specifying other members in the **WSAQUERYSET**.

In most instances, applications interested in only a particular transport protocol should constrain their query by address family and protocol using the **dwNumberOfProtocols** and **lpafpProtocols** members of the WSAQUERYSET rather than by specifiying the namespace in the **dwNameSpace** member.

Information on supported network transport protocols can be retreived using the EnumProtocols, WSAEnumProtocols, WSCEnumProtocols, or WSCEnumProtocols32 function.

It is also possible to constrain the query to a single namespace. For example, a query that only wants results from DNS (not results from the local hosts file and other naming services) would set the **dwNameSpace** member to NS_DNS. For example, a bluetooth device discovery would set the the **dwNameSpace** member to NS_BTH.

Applications can also restrict the query to a specific namespace provider by specifying a pointer to the GUID for the provider in the **lpNSProviderId** member.

Information on namespace providers on the local computer can be retrieved using the WSAEnumNameSpaceProviders, WSAEnumNameSpaceProvidersEx, WSCEnumNameSpaceProviders32, or WSCEnumNameSpaceProvidersEx32 function.

If LUP_CONTAINERS is specified in a call, other restriction values should be avoided. If any are specified, it is up to the name service provider to decide if it can support this restriction over the containers. If it cannot, it should return an error.

Some name service providers can have other means of finding containers. For example, containers might all be of some well-known type, or of a set of well-known types, and therefore a query restriction can be created for finding them. No matter what other means the name service provider has for locating containers, LUP_CONTAINERS and LUP_NOCONTAINERS take precedence. Hence, if a query restriction is given that includes containers, specifying LUP_NOCONTAINERS will prevent the container items from being returned.

Similarly, no matter the query restriction, if LUP_CONTAINERS is given, only containers should be returned. If a namespace does not support containers, and LUP_CONTAINERS is specified, it should simply return WSANO_DATA.

The preferred method of obtaining the containers within another container, is the call:

```
dwStatus = WSALookupServiceBegin(
      lpqsRestrictions,
      LUP_CONTAINERS,
      lphLookup);
```

This call is followed by the requisite number of WSALookupServiceNext calls. This will return all containers contained immediately within the starting context; that is, it is not a deep query. With this, one can map the address space structure by walking the hierarchy, perhaps enumerating the content of selected containers. Subsequent uses of **WSALookupServiceBegin** use the containers returned from a previous call.

As mentioned above, a WSAQUERYSET structure is used as an input parameter to **WSALookupBegin** in order to qualify the query. The following table indicates how the **WSAQUERYSET** is used to construct a query. When a parameter is marked as (Optional) a **NULL** pointer can be specified, indicating that the parameter will not be used as a search criteria. See section Query-Related Data Structures for additional information.

WSAQUERYSET MEMBER	QUERY INTERPRETATION
dwSize	Must be set to sizeof(WSAQUERYSET). This is a versioning mechanism.
dwOutputFlags	Ignored for queries.
lpszServiceInstanceName	(Optional) Referenced string contains service name. The semantics for wildcarding within the string are not defined, but can be supported by certain namespace providers.
lpServiceClassId	(Required) The GUID corresponding to the service class.
lpVersion	(Optional) References desired version number and provides version comparison semantics (that is, version must match exactly, or version must be not less than the value specified).
lpszComment	Ignored for queries.
dwNameSpace See the Important note that follows.	Identifier of a single namespace in which to constrain the search, or NS_ALL to include all namespaces.
lpNSProviderId	(Optional) References the GUID of a specific namespace provider, and limits the query to this provider only.
lpszContext	(Optional) Specifies the starting point of the query in a hierarchical namespace.
dwNumberOfProtocols	Size of the protocol constraint array, can be zero.
lpafpProtocols	(Optional) References an array of AFPROTOCOLS structure. Only services that utilize these protocols will be returned.

lpszQueryString	(Optional) Some namespaces (such as whois++) support enriched SQL-like queries that are contained in a simple text string. This parameter is used to specify that string.
dwNumberOfCsAddrs	Ignored for queries.
lpcsaBuffer	Ignored for queries.
lpBlob	(Optional) This is a pointer to a provider-specific entity.

> **Important** In most instances, applications interested in only a particular transport protocol should constrain their query by address family and protocol rather than by namespace. This would allow an application that needs to locate a TCP/IP service, for example, to have its query processed by all available namespaces such as the local hosts file, DNS, and NIS.

Windows Phone 8: The **WSALookupServiceBeginW** function is supported for Windows Phone Store apps on Windows Phone 8 and later.

Windows 8.1 and **Windows Server 2012 R2**: The **WSALookupServiceBeginW** function is supported for Windows Store apps on Windows 8.1, Windows Server 2012 R2, and later.

> **NOTE**
> The winsock2.h header defines WSALookupServiceBegin as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see Conventions for Function Prototypes.

Requirements

Minimum supported client	Windows 8.1, Windows Vista [desktop apps | UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps | UWP apps]
Target Platform	Windows
Header	winsock2.h
Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

Bluetooth and WSALookupServiceBegin

EnumProtocols

WSAEnumNameSpaceProviders

WSAEnumNameSpaceProvidersEx

WSAEnumProtocols

WSALookupServiceEnd

WSALookupServiceNext

WSAQUERYSET

WSCEnumNameSpaceProviders32

WSCEnumNameSpaceProvidersEx32

WSCEnumProtocols

WSCEnumProtocols32

WSALookupServiceEnd function (winsock2.h)

10/22/2022 • 2 minutes to read • Edit Online

The **WSALookupServiceEnd** function is called to free the handle after previous calls to WSALookupServiceBegin and WSALookupServiceNext.

If you call **WSALookupServiceEnd** from another thread while an existing WSALookupServiceNext is blocked, the end call will have the same effect as a cancel and will cause the **WSALookupServiceNext** call to return immediately.

Syntax

```
INT WSAAPI WSALookupServiceEnd(
  [in] HANDLE hLookup
);
```

Parameters

`[in] hLookup`

Handle previously obtained by calling WSALookupServiceBegin.

Return value

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

ERROR CODE	MEANING
WSA_INVALID_HANDLE	The handle is not valid.
WSANOTINITIALISED	The WS2_32.DLL has not been initialized. The application must first call WSAStartup before calling any Windows Sockets functions.
WSA_NOT_ENOUGH_MEMORY	There was insufficient memory to perform the operation.

Remarks

Windows Phone 8: This function is supported for Windows Phone Store apps on Windows Phone 8 and later.

Windows 8.1 and **Windows Server 2012 R2**: This function is supported for Windows Store apps on Windows 8.1, Windows Server 2012 R2, and later.

Requirements

Minimum supported client	Windows 8.1, Windows Vista [desktop apps | UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps | UWP apps]
Target Platform	Windows
Header	winsock2.h
Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

Bluetooth and WSALookupServiceEnd

WSALookupServiceBegin

WSALookupServiceNext

Winsock Functions

Winsock Reference

WSALookupServiceNextA function (winsock2.h)

10/22/2022 • 7 minutes to read • Edit Online

The **WSALookupServiceNext** function is called after obtaining a handle from a previous call to WSALookupServiceBegin in order to retrieve the requested service information.

The provider will pass back a WSAQUERYSET structure in the *lpqsResults* buffer. The client should continue to call this function until it returns WSA_E_NO_MORE, indicating that all of **WSAQUERYSET** has been returned.

Syntax

```
INT WSAAPI WSALookupServiceNextA(
  [in]      HANDLE         hLookup,
  [in]      DWORD          dwControlFlags,
  [in, out] LPDWORD        lpdwBufferLength,
  [out]     LPWSAQUERYSETA lpqsResults
);
```

Parameters

`[in] hLookup`

A handle returned from the previous call to WSALookupServiceBegin.

`[in] dwControlFlags`

A set of flags that controls the operation. The values passed in the *dwControlFlags* parameter to the WSALookupServiceBegin function determine the possible criteria. Any values passed in the *dwControlFlags* parameter to the **WSALookupServiceNext** function further restrict the criteria for the service lookup.

Currently, LUP_FLUSHPREVIOUS is defined as a means to cope with a result set that is too large. If an application does not (or cannot) supply a large enough buffer, setting LUP_FLUSHPREVIOUS instructs the provider to discard the last result set—which was too large—and move on to the next set for this call.

Supported values for the *dwControlFlags* parameter are defined in the *Winsock2.h* header file and can be a combination of the following options.

FLAG	MEANING
LUP_DEEP 0x0001	Queries deep as opposed to just the first level.
LUP_CONTAINERS 0x0002	Returns containers only.
LUP_NOCONTAINERS 0x0004	Do not return containers.

LUP_NEAREST 0x0008	If possible, returns results in the order of distance. The measure of distance is provider specific.
LUP_RETURN_NAME 0x0010	Retrieves the name as *lpszServiceInstanceName*.
LUP_RETURN_TYPE 0x0020	Retrieves the type as *lpServiceClassId*.
LUP_RETURN_VERSION 0x0040	Retrieves the version as *lpVersion*.
LUP_RETURN_COMMENT 0x0080	Retrieves the comment as *lpszComment*.
LUP_RETURN_ADDR 0x0100	Retrieves the addresses as *lpcsaBuffer*.
LUP_RETURN_BLOB 0x0200	Retrieves the private data as *lpBlob*.
LUP_RETURN_ALIASES 0x0400	Any available alias information is to be returned in successive calls to **WSALookupServiceNext**, and each alias returned will have the RESULT_IS_ALIAS flag set.
LUP_RETURN_QUERY_STRING 0x0800	Retrieves the query string used for the request.
LUP_RETURN_ALL 0x0FF0	A set of flags that retrieves all of the LUP_RETURN_* values.
LUP_FLUSHPREVIOUS 0x1000	Used as a value for the *dwControlFlags* parameter in **WSALookupServiceNext**. Setting this flag instructs the provider to discard the last result set, which was too large for the specified buffer, and move on to the next result set.
LUP_FLUSHCACHE 0x2000	If the provider has been caching information, ignores the cache, and queries the namespace itself.
LUP_RES_SERVICE 0x8000	This indicates whether prime response is in the remote or local part of CSADDR_INFO structure. The other part needs to be usable in either case.

`[in, out] lpdwBufferLength`

On input, the number of bytes contained in the buffer pointed to by *lpqsResults*. On output, if the function fails and the error is WSAEFAULT, then it contains the minimum number of bytes to pass for the *lpqsResults* to retrieve the record.

`[out] lpqsResults`

A pointer to a block of memory, which will contain one result set in a WSAQUERYSET structure on return.

Return value

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

ERROR CODE	MEANING
WSA_E_CANCELLED	A call to WSALookupServiceEnd was made while this call was still processing. The call has been canceled. The data in the *lpqsResults* buffer is undefined. In Windows Sockets version 2, conflicting error codes are defined for WSAECANCELLED (10103) and WSA_E_CANCELLED (10111). The error code WSAECANCELLED will be removed in a future version and only WSA_E_CANCELLED will remain. For Windows Sockets version 2, however, applications should check for both WSAECANCELLED and WSA_E_CANCELLED for the widest possible compatibility with namespace providers that use either one.
WSA_E_NO_MORE	There is no more data available. In Windows Sockets version 2, conflicting error codes are defined for WSAENOMORE (10102) and WSA_E_NO_MORE (10110). The error code WSAENOMORE will be removed in a future version and only WSA_E_NO_MORE will remain. For Windows Sockets version 2, however, applications should check for both WSAENOMORE and WSA_E_NO_MORE for the widest possible compatibility with name-space providers that use either one.
WSAEFAULT	The *lpqsResults* buffer was too small to contain a WSAQUERYSET set.
WSAEINVAL	One or more required parameters were invalid or missing.
WSA_INVALID_HANDLE	The specified Lookup handle is invalid.
WSANOTINITIALISED	The WS2_32.DLL has not been initialized. The application must first call WSAStartup before calling any Windows Sockets functions.
WSANO_DATA	The name was found in the database, but no data matching the given restrictions was located.
WSA_NOT_ENOUGH_MEMORY	There was insufficient memory to perform the operation.

Remarks

The *dwControlFlags* parameter specified in this function and the ones specified at the time of WSALookupServiceBegin are treated as restrictions for the purpose of combination. The restrictions are combined between the ones at **WSALookupServiceBegin** time and the ones at **WSALookupServiceNext** time. Therefore the flags at **WSALookupServiceNext** can never increase the amount of data returned beyond what was requested at **WSALookupServiceBegin**, although it is not an error to specify more or fewer flags. The flags specified at a given **WSALookupServiceNext** apply only to that call.

The *dwControlFlags* LUP_FLUSHPREVIOUS and LUP_RES_SERVICE are exceptions to the combined restrictions rule (because they are behavior flags instead of restriction flags). If either of these flags are used in **WSALookupServiceNext** they have their defined effect regardless of the setting of the same flags at WSALookupServiceBegin.

For example, if LUP_RETURN_VERSION is specified at WSALookupServiceBegin the service provider retrieves records including the version. If LUP_RETURN_VERSION is NOT specified at **WSALookupServiceNext**, the returned information does not include the version, even though it was available. No error is generated.

Also for example, if LUP_RETURN_BLOB is NOT specified at WSALookupServiceBegin but is specified at **WSALookupServiceNext**, the returned information does not include the private data. No error is generated.

If the **WSALookupServiceNext** function fails with an error of WSAEFAULT, this indicates that the buffer pointed to by the *lpqsResults* parameter was too small to contain the query results. A new buffer for a WSAQUERYSET should be provided with a size specified by the value pointed to by the *lpdwBufferLength* parameter. This new buffer for the **WSAQUERYSET** needs to have some of the members of the **WSAQUERYSET** specified before calling the **WSALookupServiceNext** function again. At a minimum, the **dwSize** member of the **WSAQUERYSET** must be set to the new size of the buffer.

Query Results

The following table describes how the query results are represented in the WSAQUERYSET structure.

WSAQUERYSET MEMBER	RESULT INTERPRETATION
dwSize	Will be set to sizeof(WSAQUERYSET). This is used as a versioning mechanism.
dwOutputFlags	RESULT_IS_ALIAS flag indicates this is an alias result.
lpszServiceInstanceName	Referenced string contains service name.
lpServiceClassId	The GUID corresponding to the service class.
lpVersion	References version number of the particular service instance.
lpszComment	Optional comment string specified by service instance.
dwNameSpace	Namespace in which the service instance was found.
lpNSProviderId	Identifies the specific namespace provider that supplied this query result.
lpszContext	Specifies the context point in a hierarchical namespace at which the service is located.
dwNumberOfProtocols	Undefined for results.

lpafpProtocols	Undefined for results, all needed protocol information is in the CSADDR_INFO structures.
lpszQueryString	When *dwControlFlags* includes LUP_RETURN_QUERY_STRING, this parameter returns the unparsed remainder of the *lpszServiceInstanceName* specified in the original query. For example, in a namespace that identifies services by hierarchical names that specify a host name and a file path within that host, the address returned might be the host address and the unparsed remainder might be the file path. If the *lpszServiceInstanceName* is fully parsed and LUP_RETURN_QUERY_STRING is used, this parameter is **NULL** or points to a zero-length string.
dwNumberOfCsAddrs	Indicates the number of elements in the array of CSADDR_INFO structures.
lpcsaBuffer	A pointer to an array of CSADDR_INFO structures, with one complete transport address contained within each element.
lpBlob	(Optional) This is a pointer to a provider-specific entity.

Windows Phone 8: The **WSALookupServiceNextW** function is supported for Windows Phone Store apps on Windows Phone 8 and later.

Windows 8.1 and **Windows Server 2012 R2**: The **WSALookupServiceNextW** function is supported for Windows Store apps on Windows 8.1, Windows Server 2012 R2, and later.

> **NOTE**
> The winsock2.h header defines WSALookupServiceNext as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see Conventions for Function Prototypes.

Requirements

Minimum supported client	Windows 8.1, Windows Vista [desktop apps | UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps | UWP apps]
Target Platform	Windows
Header	winsock2.h
Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

Bluetooth and WSALookupServiceNext

WSALookupServiceBegin

WSALookupServiceEnd

WSAQUERYSET

Winsock Functions

Winsock Reference

WSALookupServiceNextW function (winsock2.h)

10/22/2022 • 7 minutes to read • Edit Online

The **WSALookupServiceNext** function is called after obtaining a handle from a previous call to WSALookupServiceBegin in order to retrieve the requested service information.

The provider will pass back a WSAQUERYSET structure in the *lpqsResults* buffer. The client should continue to call this function until it returns WSA_E_NO_MORE, indicating that all of **WSAQUERYSET** has been returned.

Syntax

```
INT WSAAPI WSALookupServiceNextW(
  [in]      HANDLE         hLookup,
  [in]      DWORD          dwControlFlags,
  [in, out] LPDWORD        lpdwBufferLength,
  [out]     LPWSAQUERYSETW lpqsResults
);
```

Parameters

`[in] hLookup`

A handle returned from the previous call to WSALookupServiceBegin.

`[in] dwControlFlags`

A set of flags that controls the operation. The values passed in the *dwControlFlags* parameter to the WSALookupServiceBegin function determine the possible criteria. Any values passed in the *dwControlFlags* parameter to the **WSALookupServiceNext** function further restrict the criteria for the service lookup.

Currently, LUP_FLUSHPREVIOUS is defined as a means to cope with a result set that is too large. If an application does not (or cannot) supply a large enough buffer, setting LUP_FLUSHPREVIOUS instructs the provider to discard the last result set—which was too large—and move on to the next set for this call.

Supported values for the *dwControlFlags* parameter are defined in the *Winsock2.h* header file and can be a combination of the following options.

FLAG	MEANING
LUP_DEEP 0x0001	Queries deep as opposed to just the first level.
LUP_CONTAINERS 0x0002	Returns containers only.
LUP_NOCONTAINERS 0x0004	Do not return containers.

LUP_NEAREST 0x0008	If possible, returns results in the order of distance. The measure of distance is provider specific.
LUP_RETURN_NAME 0x0010	Retrieves the name as *lpszServiceInstanceName*.
LUP_RETURN_TYPE 0x0020	Retrieves the type as *lpServiceClassId*.
LUP_RETURN_VERSION 0x0040	Retrieves the version as *lpVersion*.
LUP_RETURN_COMMENT 0x0080	Retrieves the comment as *lpszComment*.
LUP_RETURN_ADDR 0x0100	Retrieves the addresses as *lpcsaBuffer*.
LUP_RETURN_BLOB 0x0200	Retrieves the private data as *lpBlob*.
LUP_RETURN_ALIASES 0x0400	Any available alias information is to be returned in successive calls to **WSALookupServiceNext**, and each alias returned will have the RESULT_IS_ALIAS flag set.
LUP_RETURN_QUERY_STRING 0x0800	Retrieves the query string used for the request.
LUP_RETURN_ALL 0x0FF0	A set of flags that retrieves all of the LUP_RETURN_* values.
LUP_FLUSHPREVIOUS 0x1000	Used as a value for the *dwControlFlags* parameter in **WSALookupServiceNext**. Setting this flag instructs the provider to discard the last result set, which was too large for the specified buffer, and move on to the next result set.
LUP_FLUSHCACHE 0x2000	If the provider has been caching information, ignores the cache, and queries the namespace itself.
LUP_RES_SERVICE 0x8000	This indicates whether prime response is in the remote or local part of CSADDR_INFO structure. The other part needs to be usable in either case.

`[in, out] lpdwBufferLength`

On input, the number of bytes contained in the buffer pointed to by *lpqsResults*. On output, if the function fails and the error is WSAEFAULT, then it contains the minimum number of bytes to pass for the *lpqsResults* to retrieve the record.

`[out] lpqsResults`

A pointer to a block of memory, which will contain one result set in a WSAQUERYSET structure on return.

Return value

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

ERROR CODE	MEANING
WSA_E_CANCELLED	A call to WSALookupServiceEnd was made while this call was still processing. The call has been canceled. The data in the *lpqsResults* buffer is undefined. In Windows Sockets version 2, conflicting error codes are defined for WSAECANCELLED (10103) and WSA_E_CANCELLED (10111). The error code WSAECANCELLED will be removed in a future version and only WSA_E_CANCELLED will remain. For Windows Sockets version 2, however, applications should check for both WSAECANCELLED and WSA_E_CANCELLED for the widest possible compatibility with namespace providers that use either one.
WSA_E_NO_MORE	There is no more data available. In Windows Sockets version 2, conflicting error codes are defined for WSAENOMORE (10102) and WSA_E_NO_MORE (10110). The error code WSAENOMORE will be removed in a future version and only WSA_E_NO_MORE will remain. For Windows Sockets version 2, however, applications should check for both WSAENOMORE and WSA_E_NO_MORE for the widest possible compatibility with name-space providers that use either one.
WSAEFAULT	The *lpqsResults* buffer was too small to contain a WSAQUERYSET set.
WSAEINVAL	One or more required parameters were invalid or missing.
WSA_INVALID_HANDLE	The specified Lookup handle is invalid.
WSANOTINITIALISED	The WS2_32.DLL has not been initialized. The application must first call WSAStartup before calling any Windows Sockets functions.
WSANO_DATA	The name was found in the database, but no data matching the given restrictions was located.
WSA_NOT_ENOUGH_MEMORY	There was insufficient memory to perform the operation.

Remarks

The *dwControlFlags* parameter specified in this function and the ones specified at the time of WSALookupServiceBegin are treated as restrictions for the purpose of combination. The restrictions are combined between the ones at **WSALookupServiceBegin** time and the ones at **WSALookupServiceNext** time. Therefore the flags at **WSALookupServiceNext** can never increase the amount of data returned beyond what was requested at **WSALookupServiceBegin**, although it is not an error to specify more or fewer flags. The flags specified at a given **WSALookupServiceNext** apply only to that call.

The *dwControlFlags* LUP_FLUSHPREVIOUS and LUP_RES_SERVICE are exceptions to the combined restrictions rule (because they are behavior flags instead of restriction flags). If either of these flags are used in **WSALookupServiceNext** they have their defined effect regardless of the setting of the same flags at WSALookupServiceBegin.

For example, if LUP_RETURN_VERSION is specified at WSALookupServiceBegin the service provider retrieves records including the version. If LUP_RETURN_VERSION is NOT specified at **WSALookupServiceNext**, the returned information does not include the version, even though it was available. No error is generated.

Also for example, if LUP_RETURN_BLOB is NOT specified at WSALookupServiceBegin but is specified at **WSALookupServiceNext**, the returned information does not include the private data. No error is generated.

If the **WSALookupServiceNext** function fails with an error of WSAEFAULT, this indicates that the buffer pointed to by the *lpqsResults* parameter was too small to contain the query results. A new buffer for a WSAQUERYSET should be provided with a size specified by the value pointed to by the *lpdwBufferLength* parameter. This new buffer for the **WSAQUERYSET** needs to have some of the members of the **WSAQUERYSET** specified before calling the **WSALookupServiceNext** function again. At a minimum, the **dwSize** member of the **WSAQUERYSET** must be set to the new size of the buffer.

Query Results

The following table describes how the query results are represented in the WSAQUERYSET structure.

WSAQUERYSET MEMBER	RESULT INTERPRETATION
dwSize	Will be set to sizeof(WSAQUERYSET). This is used as a versioning mechanism.
dwOutputFlags	RESULT_IS_ALIAS flag indicates this is an alias result.
lpszServiceInstanceName	Referenced string contains service name.
lpServiceClassId	The GUID corresponding to the service class.
lpVersion	References version number of the particular service instance.
lpszComment	Optional comment string specified by service instance.
dwNameSpace	Namespace in which the service instance was found.
lpNSProviderId	Identifies the specific namespace provider that supplied this query result.
lpszContext	Specifies the context point in a hierarchical namespace at which the service is located.
dwNumberOfProtocols	Undefined for results.

lpafpProtocols	Undefined for results, all needed protocol information is in the CSADDR_INFO structures.
lpszQueryString	When *dwControlFlags* includes LUP_RETURN_QUERY_STRING, this parameter returns the unparsed remainder of the *lpszServiceInstanceName* specified in the original query. For example, in a namespace that identifies services by hierarchical names that specify a host name and a file path within that host, the address returned might be the host address and the unparsed remainder might be the file path. If the *lpszServiceInstanceName* is fully parsed and LUP_RETURN_QUERY_STRING is used, this parameter is **NULL** or points to a zero-length string.
dwNumberOfCsAddrs	Indicates the number of elements in the array of CSADDR_INFO structures.
lpcsaBuffer	A pointer to an array of CSADDR_INFO structures, with one complete transport address contained within each element.
lpBlob	(Optional) This is a pointer to a provider-specific entity.

Windows Phone 8: The **WSALookupServiceNextW** function is supported for Windows Phone Store apps on Windows Phone 8 and later.

Windows 8.1 and **Windows Server 2012 R2**: The **WSALookupServiceNextW** function is supported for Windows Store apps on Windows 8.1, Windows Server 2012 R2, and later.

> **NOTE**
> The winsock2.h header defines WSALookupServiceNext as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see Conventions for Function Prototypes.

Requirements

Minimum supported client	Windows 8.1, Windows Vista [desktop apps | UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps | UWP apps]
Target Platform	Windows
Header	winsock2.h
Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

Bluetooth and WSALookupServiceNext

WSALookupServiceBegin

WSALookupServiceEnd

WSAQUERYSET

Winsock Functions

Winsock Reference

WSANAMESPACE_INFOA structure (winsock2.h)

10/22/2022 • 2 minutes to read • Edit Online

The **WSANAMESPACE_INFO** structure contains all registration information for a namespace provider.

Syntax

```
typedef struct _WSANAMESPACE_INFOA {
  GUID  NSProviderId;
  DWORD dwNameSpace;
  BOOL  fActive;
  DWORD dwVersion;
  LPSTR lpszIdentifier;
} WSANAMESPACE_INFOA, *PWSANAMESPACE_INFOA, *LPWSANAMESPACE_INFOA;
```

Members

`NSProviderId`

Type: **GUID**

A unique GUID for this namespace provider.

`dwNameSpace`

Type: **DWORD**

The namespace supported by this provider.

Possible values for the **dwNameSpace** member are listed in the *Winsock2.h* include file. Several namespace providers are included with Windows Vista and later. Other namespace providers can be installed, so the following possible values are only those commonly available. Many other values are possible.

VALUE	MEANING
NS_BTH	The Bluetooth namespace. This namespace identifier is supported on Windows Vista and later.
NS_DNS	The domain name system (DNS) namespace.
NS_EMAIL	The email namespace. This namespace identifier is supported on Windows Vista and later.
NS_NLA	The network location awareness (NLA) namespace. This namespace identifier is supported on Windows XP and later.

NS_NTDS	The Windows NT directory service (NTDS) namespace.
NS_PNRPNAME	The peer-to-peer name space for a specific peer name. This namespace identifier is supported on Windows Vista and later.
NS_PNRPCLOUD	The peer-to-peer name space for a collection of peer names. This namespace identifier is supported on Windows Vista and later.

`fActive`

Type: **BOOL**

If **TRUE**, indicates that this namespace provider is active. If **FALSE**, the namespace provider is inactive and is not accessible for queries, even if the query specifically references this namespace provider.

`dwVersion`

Type: **DWORD**

The version number of the namespace provider.

`lpszIdentifier`

Type: **LPTSTR**

A display string that identifies the namespace provider.

Remarks

The **WSANAMESPACE_INFO** structure is used by the WSAEnumNameSpaceProviders and WSCEnumNameSpaceProviders32 functions to return information on available namespace providers. The **WSANAMESPACE_INFO** structure contains the provider-specific information on the namespace entry passed to the WSCInstallNameSpace and WSCInstallNameSpace32 functions when the namespace provider was installed.

When UNICODE or _UNICODE is defined, **WSANAMESPACE_INFO** is defined to **WSANAMESPACE_INFOW**, the Unicode version of this data structure and the **lpszIdentifier** string member is defined to the **LPWSTR** data type.

When UNICODE or _UNICODE is not defined, **WSANAMESPACE_INFO** is defined to **WSANAMESPACE_INFOA**, the ANSI version of this data structure and the **lpszIdentifier** string member is defined to the **LPSTR** data type.

On Windows Vista and later, WSANAMESPACE_INFOEX, an enhanced version of the **WSANAMESPACE_INFO** structure, is returned by calls to the WSAEnumNameSpaceProvidersEx and WSCEnumNameSpaceProvidersEx32 functions

> **NOTE**
>
> The winsock2.h header defines WSANAMESPACE_INFO as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see Conventions for Function Prototypes.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	winsock2.h

See also

WSAEnumNameSpaceProviders

WSAEnumNameSpaceProvidersEx

WSANAMESPACE_INFOEX

WSCEnumNameSpaceProviders32

WSCEnumNameSpaceProvidersEx32

WSCInstallNameSpace

WSCInstallNameSpace32

WSANAMESPACE_INFOEXA structure (winsock2.h)

10/22/2022 • 2 minutes to read • Edit Online

The **WSANAMESPACE_INFOEX** structure contains all registration information for a namespace provider.

Syntax

```
typedef struct _WSANAMESPACE_INFOEXA {
  GUID  NSProviderId;
  DWORD dwNameSpace;
  BOOL  fActive;
  DWORD dwVersion;
  LPSTR lpszIdentifier;
  BLOB  ProviderSpecific;
} WSANAMESPACE_INFOEXA, *PWSANAMESPACE_INFOEXA, *LPWSANAMESPACE_INFOEXA;
```

Members

`NSProviderId`

Type: **GUID**

A unique GUID for this namespace provider.

`dwNameSpace`

Type: **DWORD**

The namespace supported by this provider.

Possible values for the **dwNameSpace** member are listed in the *Winsock2.h* include file. Several namespace providers are included with Windows Vista and later. Other namespace providers can be installed, so the following possible values are only those commonly available. Many other values are possible.

VALUE	MEANING
NS_BTH	The Bluetooth namespace. This namespace identifier is supported on Windows Vista and later.
NS_DNS	The domain name system (DNS) namespace.
NS_EMAIL	The email namespace. This namespace identifier is supported on Windows Vista and later.
NS_NLA	The network location awareness (NLA) namespace. This namespace identifier is supported on Windows XP and later.

NS_NTDS	The Windows NT directory service (NTDS) namespace.
NS_PNRPNAME	The peer-to-peer name space for a specific peer name. This namespace identifier is supported on Windows Vista and later.
NS_PNRPCLOUD	The peer-to-peer name space for a collection of peer names. This namespace identifier is supported on Windows Vista and later.

`fActive`

Type: **BOOL**

If **TRUE**, indicates that this namespace provider is active. If **FALSE**, the namespace provider is inactive and is not accessible for queries, even if the query specifically references this namespace provider.

`dwVersion`

Type: **DWORD**

The version number of the namespace provider.

`lpszIdentifier`

Type: **LPTSTR**

A display string that identifies the namespace provider.

`ProviderSpecific`

Type: **BLOB**

A provider-specific data blob associated with namespace entry.

Remarks

The **WSANAMESPACE_INFOEX** structure is an enhanced version of the WSANAMESPACE_INFO structure that is used by the WSAEnumNameSpaceProvidersEx and the WSCEnumNameSpaceProvidersEx32 functions to return information on available namespace providers. The **WSANAMESPACE_INFOEX** structure contains the provider-specific data blob associated with the namespace entry passed in the *lpProviderInfo* parameter to the WSCInstallNameSpaceEx and WSCInstallNameSpaceEx32 functions.

Currently, the only namespace included with Windows that uses information in the **ProviderSpecific** member of the **WSANAMESPACE_INFOEX** structure are namespace providers for the NS_EMAIL namespace. The format of the **ProviderSpecific** member for an NS_EMAIL namespace provider is a NAPI_PROVIDER_INSTALLATION_BLOB structure.

When UNICODE or _UNICODE is defined, **WSANAMESPACE_INFOEX** is defined to **WSANAMESPACE_INFOEXW**, the Unicode version of this structure and the **lpszIdentifier** string member is defined to the **LPWSTR** data type.

When UNICODE or _UNICODE is not defined, **WSANAMESPACE_INFOEX** is defined to **WSANAMESPACE_INFOEXA**, the ANSI version of this structure and the **lpszIdentifier** string member is defined to the **LPSTR** data type.

The WSCEnumNameSpaceProvidersEx32 function is a Unicode only function and returns **WSANAMESPACE_INFOEXW** structures.

> **NOTE**
>
> The winsock2.h header defines WSANAMESPACE_INFOEX as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see Conventions for Function Prototypes.

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Header	winsock2.h

See also

NAPI_PROVIDER_INSTALLATION_BLOB

WSAEnumNameSpaceProvidersEx

WSANAMESPACE_INFO

WSCEnumNameSpaceProvidersEx32

WSCInstallNameSpaceEx

WSCInstallNameSpaceEx32

WSANAMESPACE_INFOEXW structure (winsock2.h)

10/22/2022 • 2 minutes to read • Edit Online

The **WSANAMESPACE_INFOEX** structure contains all registration information for a namespace provider.

Syntax

```
typedef struct _WSANAMESPACE_INFOEXW {
  GUID   NSProviderId;
  DWORD  dwNameSpace;
  BOOL   fActive;
  DWORD  dwVersion;
  LPWSTR lpszIdentifier;
  BLOB   ProviderSpecific;
} WSANAMESPACE_INFOEXW, *PWSANAMESPACE_INFOEXW, *LPWSANAMESPACE_INFOEXW;
```

Members

`NSProviderId`

Type: **GUID**

A unique GUID for this namespace provider.

`dwNameSpace`

Type: **DWORD**

The namespace supported by this provider.

Possible values for the **dwNameSpace** member are listed in the *Winsock2.h* include file. Several namespace providers are included with Windows Vista and later. Other namespace providers can be installed, so the following possible values are only those commonly available. Many other values are possible.

VALUE	MEANING
NS_BTH	The Bluetooth namespace. This namespace identifier is supported on Windows Vista and later.
NS_DNS	The domain name system (DNS) namespace.
NS_EMAIL	The email namespace. This namespace identifier is supported on Windows Vista and later.
NS_NLA	The network location awareness (NLA) namespace. This namespace identifier is supported on Windows XP and later.

NS_NTDS	The Windows NT directory service (NTDS) namespace.
NS_PNRPNAME	The peer-to-peer name space for a specific peer name. This namespace identifier is supported on Windows Vista and later.
NS_PNRPCLOUD	The peer-to-peer name space for a collection of peer names. This namespace identifier is supported on Windows Vista and later.

`fActive`

Type: **BOOL**

If **TRUE**, indicates that this namespace provider is active. If **FALSE**, the namespace provider is inactive and is not accessible for queries, even if the query specifically references this namespace provider.

`dwVersion`

Type: **DWORD**

The version number of the namespace provider.

`lpszIdentifier`

Type: **LPTSTR**

A display string that identifies the namespace provider.

`ProviderSpecific`

Type: **BLOB**

A provider-specific data blob associated with namespace entry.

Remarks

The **WSANAMESPACE_INFOEX** structure is an enhanced version of the WSANAMESPACE_INFO structure that is used by the WSAEnumNameSpaceProvidersEx and the WSCEnumNameSpaceProvidersEx32 functions to return information on available namespace providers. The **WSANAMESPACE_INFOEX** structure contains the provider-specific data blob associated with the namespace entry passed in the *lpProviderInfo* parameter to the WSCInstallNameSpaceEx and WSCInstallNameSpaceEx32 functions.

Currently, the only namespace included with Windows that uses information in the **ProviderSpecific** member of the **WSANAMESPACE_INFOEX** structure are namespace providers for the NS_EMAIL namespace. The format of the **ProviderSpecific** member for an NS_EMAIL namespace provider is a NAPI_PROVIDER_INSTALLATION_BLOB structure.

When UNICODE or _UNICODE is defined, **WSANAMESPACE_INFOEX** is defined to **WSANAMESPACE_INFOEXW**, the Unicode version of this structure and the **lpszIdentifier** string member is defined to the **LPWSTR** data type.

When UNICODE or _UNICODE is not defined, **WSANAMESPACE_INFOEX** is defined to **WSANAMESPACE_INFOEXA**, the ANSI version of this structure and the **lpszIdentifier** string member is defined to the **LPSTR** data type.

The WSCEnumNameSpaceProvidersEx32 function is a Unicode only function and returns

WSANAMESPACE_INFOEXW structures.

> **NOTE**
>
> The winsock2.h header defines WSANAMESPACE_INFOEX as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see Conventions for Function Prototypes.

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Header	winsock2.h

See also

NAPI_PROVIDER_INSTALLATION_BLOB

WSAEnumNameSpaceProvidersEx

WSANAMESPACE_INFO

WSCEnumNameSpaceProvidersEx32

WSCInstallNameSpaceEx

WSCInstallNameSpaceEx32

WSANAMESPACE_INFOW structure (winsock2.h)

10/22/2022 • 2 minutes to read • Edit Online

The **WSANAMESPACE_INFO** structure contains all registration information for a namespace provider.

Syntax

```
typedef struct _WSANAMESPACE_INFOW {
  GUID   NSProviderId;
  DWORD  dwNameSpace;
  BOOL   fActive;
  DWORD  dwVersion;
  LPWSTR lpszIdentifier;
} WSANAMESPACE_INFOW, *PWSANAMESPACE_INFOW, *LPWSANAMESPACE_INFOW;
```

Members

`NSProviderId`

Type: **GUID**

A unique GUID for this namespace provider.

`dwNameSpace`

Type: **DWORD**

The namespace supported by this provider.

Possible values for the **dwNameSpace** member are listed in the *Winsock2.h* include file. Several namespace providers are included with Windows Vista and later. Other namespace providers can be installed, so the following possible values are only those commonly available. Many other values are possible.

VALUE	MEANING
NS_BTH	The Bluetooth namespace. This namespace identifier is supported on Windows Vista and later.
NS_DNS	The domain name system (DNS) namespace.
NS_EMAIL	The email namespace. This namespace identifier is supported on Windows Vista and later.
NS_NLA	The network location awareness (NLA) namespace. This namespace identifier is supported on Windows XP and later.

NS_NTDS	The Windows NT directory service (NTDS) namespace.
NS_PNRPNAME	The peer-to-peer name space for a specific peer name. This namespace identifier is supported on Windows Vista and later.
NS_PNRPCLOUD	The peer-to-peer name space for a collection of peer names. This namespace identifier is supported on Windows Vista and later.

`fActive`

Type: **BOOL**

If **TRUE**, indicates that this namespace provider is active. If **FALSE**, the namespace provider is inactive and is not accessible for queries, even if the query specifically references this namespace provider.

`dwVersion`

Type: **DWORD**

The version number of the namespace provider.

`lpszIdentifier`

Type: **LPTSTR**

A display string that identifies the namespace provider.

Remarks

The **WSANAMESPACE_INFO** structure is used by the WSAEnumNameSpaceProviders and WSCEnumNameSpaceProviders32 functions to return information on available namespace providers. The **WSANAMESPACE_INFO** structure contains the provider-specific information on the namespace entry passed to the WSCInstallNameSpace and WSCInstallNameSpace32 functions when the namespace provider was installed.

When UNICODE or _UNICODE is defined, **WSANAMESPACE_INFO** is defined to **WSANAMESPACE_INFOW**, the Unicode version of this data structure and the **lpszIdentifier** string member is defined to the **LPWSTR** data type.

When UNICODE or _UNICODE is not defined, **WSANAMESPACE_INFO** is defined to **WSANAMESPACE_INFOA**, the ANSI version of this data structure and the **lpszIdentifier** string member is defined to the **LPSTR** data type.

On Windows Vista and later, WSANAMESPACE_INFOEX, an enhanced version of the **WSANAMESPACE_INFO** structure, is returned by calls to the WSAEnumNameSpaceProvidersEx and WSCEnumNameSpaceProvidersEx32 functions

> **NOTE**
>
> The winsock2.h header defines WSANAMESPACE_INFO as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see Conventions for Function Prototypes.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	winsock2.h

See also

WSAEnumNameSpaceProviders

WSAEnumNameSpaceProvidersEx

WSANAMESPACE_INFOEX

WSCEnumNameSpaceProviders32

WSCEnumNameSpaceProvidersEx32

WSCInstallNameSpace

WSCInstallNameSpace32

WSANETWORKEVENTS structure (winsock2.h)

10/22/2022 • 2 minutes to read • Edit Online

The **WSANETWORKEVENTS** structure is used to store a socket's internal information about network events.

Syntax

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

Members

`lNetworkEvents`

Indicates which of the FD_XXX network events have occurred.

`iErrorCode[FD_MAX_EVENTS]`

Array that contains any associated error codes, with an array index that corresponds to the position of event bits in **lNetworkEvents**. The identifiers FD_READ_BIT, FD_WRITE_BIT and others can be used to index the **iErrorCode** array.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	winsock2.h

See also

WSAEnumNetworkEvents

WSAEventSelect

WSANSCLASSINFOA structure (winsock2.h)

10/22/2022 • 2 minutes to read • Edit Online

The **WSANSCLASSINFO** structure provides individual parameter information for a specific Windows Sockets namespace.

Syntax

```
typedef struct _WSANSClassInfoA {
  LPSTR  lpszName;
  DWORD  dwNameSpace;
  DWORD  dwValueType;
  DWORD  dwValueSize;
  LPVOID lpValue;
} WSANSCLASSINFOA, *PWSANSCLASSINFOA, *LPWSANSCLASSINFOA;
```

Members

`lpszName`

String value associated with the parameter, such as SAPID, TCPPORT, and so forth.

`dwNameSpace`

GUID associated with the namespace.

`dwValueType`

Value type for the parameter, such as REG_DWORD or REG_SZ, and so forth.

`dwValueSize`

Size of the parameter provided in **lpValue**, in bytes.

`lpValue`

Pointer to the value of the parameter.

Remarks

The **WSANSCLASSINFO** structure is defined differently depending on whether ANSI or UNICODE is used. The above syntax block applies to ANSI; for UNICODE, the datatype for **lpszName** is **LPWSTR**.

> **NOTE**
>
> The winsock2.h header defines WSANSCLASSINFO as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see Conventions for Function Prototypes.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	winsock2.h

See also

WSASERVICECLASSINFO

WSANSCLASSINFOW structure (winsock2.h)

10/22/2022 • 2 minutes to read • Edit Online

The **WSANSCLASSINFO** structure provides individual parameter information for a specific Windows Sockets namespace.

Syntax

```
typedef struct _WSANSClassInfoW {
  LPWSTR lpszName;
  DWORD  dwNameSpace;
  DWORD  dwValueType;
  DWORD  dwValueSize;
  LPVOID lpValue;
} WSANSCLASSINFOW, *PWSANSCLASSINFOW, *LPWSANSCLASSINFOW;
```

Members

`lpszName`

String value associated with the parameter, such as SAPID, TCPPORT, and so forth.

`dwNameSpace`

GUID associated with the namespace.

`dwValueType`

Value type for the parameter, such as REG_DWORD or REG_SZ, and so forth.

`dwValueSize`

Size of the parameter provided in **lpValue**, in bytes.

`lpValue`

Pointer to the value of the parameter.

Remarks

The **WSANSCLASSINFO** structure is defined differently depending on whether ANSI or UNICODE is used. The above syntax block applies to ANSI; for UNICODE, the datatype for **lpszName** is **LPWSTR**.

> **NOTE**
>
> The winsock2.h header defines WSANSCLASSINFO as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see Conventions for Function Prototypes.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	winsock2.h

See also

WSASERVICECLASSINFO

WSANSPIoctl function (winsock2.h)

10/22/2022 • 6 minutes to read • Edit Online

The Windows Sockets **WSANSPIoctl** function enables developers to make I/O control calls to a registered namespace.

Syntax

```
INT WSAAPI WSANSPIoctl(
  [in]      HANDLE          hLookup,
  [in]      DWORD           dwControlCode,
  [in]      LPVOID          lpvInBuffer,
  [in, out] DWORD           cbInBuffer,
  [out]     LPVOID          lpvOutBuffer,
  [in]      DWORD           cbOutBuffer,
  [out]     LPDWORD         lpcbBytesReturned,
  [in]      LPWSACOMPLETION lpCompletion
);
```

Parameters

`[in] hLookup`

The lookup handle returned from a previous call to the WSALookupServiceBegin function.

`[in] dwControlCode`

The control code of the operation to perform.

The values that may be used for the *dwControlCode* parameter are determined by the namespace provider.

The following value is supported by several Microsoft namespace providers including the Network Location Awareness (NS_NLA) namespace provider. This IOCTL is defined in the Winsock2.h header file.

VALUE	MEANING
SIO_NSP_NOTIFY_CHANGE	This operation checks if the results returned with previous calls using the *hLookup* parameter are still valid. These previous calls include the initial call to the WSALookupServiceBegin function to retrieve the *hLookup* parameter. These previous calls may also include calls to the WSALookupServiceNext function using the *hLookup* parameter.

`[in] lpvInBuffer`

A pointer to the input buffer.

`[in, out] cbInBuffer`

The size, in bytes, of the input buffer.

`[out] lpvOutBuffer`

A pointer to the output buffer.

`[in] cbOutBuffer`

The size, in bytes, of the output buffer.

`[out] lpcbBytesReturned`

A pointer to the number of bytes returned.

`[in] lpCompletion`

A pointer to a WSACOMPLETION structure, used for asynchronous processing. Set *lpCompletion* to **NULL** to force blocking (synchronous) execution.

Return value

Success returns NO_ERROR. Failure returns SOCKET_ERROR, and a specific error code can be retrieved by calling the WSAGetLastError function. The following table describes the error codes.

ERROR CODE	DESCRIPTION
WSA_INVALID_HANDLE	The *hLookup* parameter was not a valid query handle returned by WSALookupServiceBegin.
WSA_IO_PENDING	An overlapped operation was successfully initiated and completion will be indicated at a later time.
WSAEFAULT	The *lpvInBuffer*, *cbInBuffer*, *lpvOutBuffer*, *cbOutBuffer*, or *lpCompletion* argument is not totally contained in a valid part of the user address space. Alternatively, the *cbInBuffer* or *cbOutBuffer* argument is too small, and the argument is modified to reflect the required allocation size.
WSAEINVAL	A supplied parameter is not acceptable, or the operation inappropriately returns results from multiple namespaces when it does not make sense for the specified operation.
WSAENETDOWN	The network subsystem has failed.
WSAEOPNOTSUPP	The operation is not supported. This error is returned if the namespace provider does not implement this function. This error can also be returned if the specified *dwControlCode* is an unrecognized command.
WSAEWOULDBLOCK	The socket is not using overlapped I/O (asynchronous processing), yet the *lpCompletion* parameter is non-**NULL**. This error is used as a special notification for the SIO_NSP_NOTIFY_CHANGE IOCTL when the *lpCompletion* parameter is **NULL** (a poll) to indicate that a query set remains valid.

Remarks

The **WSANSPIoctl** function is used to set or retrieve operating parameters associated with a query handle to a namespace provider. The *hLookup* parameter is a handle to the namespace provider query previously returned by the WSALookupServiceBegin function (not a socket handle).

Any IOCTL sent to a namespace provider may block indefinitely, depending upon the implementation of the namespace. If an application cannot tolerate blocking in a **WSANSPIoctl** function call, overlapped I/O should be used and the *lpCompletion* parameter should point to a WSACOMPLETION structure. To make a **WSANSPIoctl** function call nonblocking and return immediately, set the **Type** member of the **WSACOMPLETION** structure to **NSP_NOTIFY_IMMEDIATELY**.

If *lpCompletion* is **NULL**, the **WSANSPIoctl** function executes as a blocking call. The namespace provider should return immediately and should not block. But each namespace is responsible for enforcing this behavior.

The following IOCTL code is supported by several Microsoft name space provider:

SIO_NSP_NOTIFY_CHANGE
This operation checks if the results returned with previous calls using the *hLookup* parameter are still valid. These previous calls include the initial call to the WSALookupServiceBegin function to retrieve the *hLookup* parameter. These previous calls may also include calls to the WSALookupServiceNext function using the *hLookup* parameter.

The Microsoft namespace providers that support this IOCTL include the following

- NS_NLA - The Network Location Awareness (NLA) namespace provider.
- NS_PNRPNAME - The Peer Name Resolution Protocol (PNRP) namespace provider.
- NS_PNRPCLOUD - The Peer Name Resolution Protocol (PNRP) cloud namespace provider.

Other non-Microsoft namespace providers may be installed that also support this IOCTL.

When the *lpCompletion* parameter is **NULL**, this IOCTL implements a special behavior. If the *lpCompletion* parameter is **NULL** for this IOCTL, this operation is a poll and returns immediately. If the query set remains valid, WSAEWOULDBLOCK is returned as notification that the query set remains valid. If the query set has changed and is invalid, **NO_ERROR** is returned indicating success in polling for invalidation of the query set.

If the *lpCompletion* parameter is not **NULL** and points to an WSACOMPLETION structure, then the **WSANSPIoctl** function returns WSA_IO_PENDING if the overlapped operation was successfully initiated and completion will be indicated at a later time. The method specified in the **WSACOMPLETION** structure is used to notify the application if the query set is still valid.

Not all name resolution protocols are able to support this feature, and therefore, this function call may fail with WSAEOPNOTSUPP. A query containing data from multiple providers cannot call this IOCTL, and will return WSAEINVAL.

The *lpvInBuffer*, *cbInBuffer*, *lpvOutBuffer*, and *cbOutBuffer* parameters are currently ignored by Microsoft namespace providers.

For single-threaded applications, a typical method to use the **WSANSPIoctl** function is as follows. Call the **WSANSPIoctl** function with the *dwControlCode* parameter set to **SIO_NSP_NOTIFY_CHANGE** with no completion routine (the *lpCompletion* parameter set to **NULL**) after every WSALookupServiceNext function call to make sure the query data is still valid. If the data becomes invalid, call the WSALookupServiceEnd function to close the query handle. Call the WSALookupServiceBegin function to retrieve a new query handle and begin the query again.

For multi-threaded applications, a typical method to use the **WSANSPIoctl** function is as follows. Call the **WSANSPIoctl** function with the *dwControlCode* parameter set to **SIO_NSP_NOTIFY_CHANGE** with a completion routine after the initial call to the WSALookupServiceBegin function. The application would use the mechanism for notification specified in the completion routine to be notified when data is no longer valid. One common mechanism is to use an event specified in the completion routine. If the data becomes invalid, call the WSALookupServiceEnd function to close the query handle. Call the WSALookupServiceBegin and the **WSANSPIoctl** functions to retrieve a new query handle and begin the query again.

Some protocols may simply cache the information locally and invalidate it after some time, in which case notification may be issued to indicate the local cache has been invalidated.

For name resolution protocols where changes are infrequent, it is possible for a namespace provider to indicate a global change event that may not be applicable to the query on which change notification was requested and issued.

Immediate poll operations are usually much less expensive since they do not require a notification object. In most cases, this is implemented as a simple Boolean variable check. Asynchronous notification, however, may necessitate the creation of dedicated worker threads and/or inter-process communication channels, depending on the implementation of the namespace provider service, and will incur processing overhead related to the notification object involved with signaling the change event.

To cancel an asynchronous notification request, end the original query with a WSALookupServiceEnd function call on the affected query handle. Canceling the asynchronous notification for LUP_NOTIFY_HWND will not post any message, however, an overlapped operation will be completed and notification will be delivered with the error WSA_OPERATION_ABORTED.

> **Note** All I/O initiated by a given thread is canceled when that thread exits. For overlapped sockets, pending asynchronous operations can fail if the thread is closed before the operations complete. See ExitThread for more information.

Windows Phone 8: This function is supported for Windows Phone Store apps on Windows Phone 8 and later.
Windows 8.1 and **Windows Server 2012 R2**: This function is supported for Windows Store apps on Windows 8.1, Windows Server 2012 R2, and later.

Requirements

Minimum supported client	Windows 8.1, Windows Vista [desktop apps | UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps | UWP apps]
Target Platform	Windows
Header	winsock2.h

See also

ExitThread

WSACOMPLETION

WSAGetLastError

WSALookupServiceBegin

WSALookupServiceEnd

WSALookupServiceNext

Winsock Functions

WSANtohl function (winsock2.h)

10/22/2022 • 2 minutes to read • Edit Online

The **WSANtohl** function converts a **u_long** from network byte order to host byte order.

Syntax

```
int WSAAPI WSANtohl(
  [in]  SOCKET s,
  [in]  u_long netlong,
  [out] u_long *lphostlong
);
```

Parameters

`[in] s`

A descriptor identifying a socket.

`[in] netlong`

A 32-bit number in network byte order.

`[out] lphostlong`

A pointer to a 32-bit number to receive the number in host byte order.

Return value

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

ERROR CODE	MEANING
WSANOTINITIALISED	A successful WSAStartup call must occur before using this function.
WSAENETDOWN	The network subsystem has failed.
WSAENOTSOCK	The descriptor is not a socket.
WSAEFAULT	The *lphostlong* parameter is **NULL** or the address pointed to is not completely contained in a valid part of the user address space.

Remarks

The **WSANtohl** function takes a 32-bit number in network byte order and returns a 32-bit number in host byte order in the 32-bit number pointed to by the *lphostlong* parameter. The socket passed in the *s* parameter is used

to determine the network byte order required based on the Winsock catalog protocol entry associated with the socket. This feature supports Winsock providers that use different network byte orders.

If the socket is for the AF_INET or AF_INET6 address family, the **WSANtohl** function can be used to convert an IPv4 address in network byte order to the IPv4 address in host byte order. This function does not do any checking to determine if the *netlong* parameter is a valid IPv4 address.

The **WSANtohl** function requires that the Winsock DLL has previously been loaded with a successful call to the WSAStartup function. For use with the AF_INET or AF_INET6 family, the ntohl function does not require that the Winsock DLL be loaded.

Windows Phone 8: This function is supported for Windows Phone Store apps on Windows Phone 8 and later.

Windows 8.1 and **Windows Server 2012 R2**: This function is supported for Windows Store apps on Windows 8.1, Windows Server 2012 R2, and later.

Requirements

Minimum supported client	Windows 8.1, Windows Vista [desktop apps | UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps | UWP apps]
Target Platform	Windows
Header	winsock2.h
Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

InetNtop

WSAHtonl

WSAHtons

WSANtohs

Winsock Functions

Winsock Reference

htonl

htons

inet_addr

inet_ntoa

ntohl

ntohs

WSANtohs function (winsock2.h)

10/22/2022 • 2 minutes to read • Edit Online

The **WSANtohs** function converts a **u_short** from network byte order to host byte order.

Syntax

```
int WSAAPI WSANtohs(
  [in]  SOCKET  s,
  [in]  u_short netshort,
  [out] u_short *lphostshort
);
```

Parameters

`[in] s`

A descriptor identifying a socket.

`[in] netshort`

A 16-bit number in network byte order.

`[out] lphostshort`

A pointer to a 16-bit number to receive the number in host byte order.

Return value

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

ERROR CODE	MEANING
WSANOTINITIALISED	A successful WSAStartup call must occur before using this function.
WSAENETDOWN	The network subsystem has failed.
WSAENOTSOCK	The descriptor is not a socket.
WSAEFAULT	The *lphostshort* parameter is **NULL** or the address pointed to is not completely contained in a valid part of the user address space.

Remarks

The **WSANtohs** function takes a 16-bit number in network byte order and returns a 16-bit number in host byte order in the 16-bit number pointed to by the *lphostshort* parameter. The socket passed in the *s* parameter is

used to determine the network byte order required based on the Winsock catalog protocol entry associated with the socket. This feature supports Winsock providers that use different network byte orders.

If the socket is for the AF_INET or AF_INET6 address family, the **WSANtohs** function can be used to convert an IP port number in network byte order to the IP port number in host byte order.

The **WSANtohs** function requires that the Winsock DLL has previously been loaded with a successful call to the WSAStartup function. For use with the AF_INET OR AF_INET6 address family, the ntohs function does not require that the Winsock DLL be loaded.

Windows Phone 8: This function is supported for Windows Phone Store apps on Windows Phone 8 and later.

Windows 8.1 and **Windows Server 2012 R2**: This function is supported for Windows Store apps on Windows 8.1, Windows Server 2012 R2, and later.

Requirements

Minimum supported client	Windows 8.1, Windows Vista [desktop apps | UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps | UWP apps]
Target Platform	Windows
Header	winsock2.h
Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

InetNtop

WSAHtonl

WSAHtons

WSANtohl

Winsock Functions

Winsock Reference

htonl

htons

inet_addr

inet_ntoa

ntohl

ntohs

WSAOVERLAPPED structure (winsock2.h)

10/22/2022 • 2 minutes to read • Edit Online

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

Syntax

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

Members

`Internal`

Type: **ULONG_PTR**

Reserved for internal use. The Internal member is used internally by the entity that implements overlapped I/O. For service providers that create sockets as installable file system (IFS) handles, this parameter is used by the underlying operating system. Other service providers (non-IFS providers) are free to use this parameter as necessary.

`InternalHigh`

Type: **ULONG_PTR**

Reserved. Used internally by the entity that implements overlapped I/O. For service providers that create sockets as IFS handles, this parameter is used by the underlying operating system. NonIFS providers are free to use this parameter as necessary.

`Offset`

Type: **DWORD**

Reserved for use by service providers.

`OffsetHigh`

Type: **DWORD**

Reserved for use by service providers.

`hEvent`

Type: **HANDLE**

If an overlapped I/O operation is issued without an I/O completion routine (the operation's *lpCompletionRoutine* parameter is set to null), then this parameter should either contain a valid handle to a WSAEVENT object or be null. If the *lpCompletionRoutine* parameter of the call is non-null then applications are free to use this parameter

as necessary.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	winsock2.h

See also

WSACleanup

WSACloseEvent

WSACreateEvent

WSAGetOverlappedResult

WSARecv

WSASend

WSASendTo

WSASocket

WSAStartup

bind

closesocket

WSAPoll function (winsock2.h)

10/22/2022 • 5 minutes to read • Edit Online

The **WSAPoll** function determines status of one or more sockets.

Syntax

```
int WSAAPI WSAPoll(
  [in, out] LPWSAPOLLFD fdArray,
  [in]      ULONG       fds,
  [in]      INT         timeout
);
```

Parameters

`[in, out] fdArray`

An array of one or more **POLLFD** structures specifying the set of sockets for which status is requested. The array must contain at least one structure with a valid socket. Upon return, this parameter receives the updated sockets with the **revents** status flags member set on each one that matches the status query criteria.

`[in] fds`

The number of **WSAPOLLFD** structures in *fdarray*. This is not necessarily the number of sockets for which status is requested.

`[in] timeout`

A value that specifies the wait behavior, based on the following values.

VALUE	MEANING
Greater than zero	The time, in milliseconds, to wait.
Zero	Return immediately.
Less than zero	Wait indefinitely.

Return value

Returns one of the following values.

RETURN VALUE	DESCRIPTION
Zero	No sockets were in the queried state before the timer expired.
Greater than zero	The number of elements in *fdarray* for which an **revents** member of the **POLLFD** structure is nonzero.

SOCKET_ERROR	An error occurred. Call the WSAGetLastError function to retrieve the extended error code.

EXTENDED ERROR CODE	MEANING
WSAENETDOWN	The network subsystem has failed.
WSAEFAULT	An exception occurred while reading user input parameters.
WSAEINVAL	An invalid parameter was passed. This error is returned if the WSAPOLLFD structures pointed to by the *fdarray* parameter when requesting socket status. This error is also returned if none of the sockets specified in the **fd** member of any of the **WSAPOLLFD** structures pointed to by the *fdarray* parameter were valid.
WSAENOBUFS	The function was unable to allocate sufficient memory.

Remarks

The **WSAPoll** function is defined on Windows Vista and later.

The WSAPOLLFD structures. An application sets the appropriate flags in the **events** member of the **WSAPOLLFD** structure to specify the type of status requested for each corresponding socket. The **WSAPoll** function returns the status of a socket in the **revents** member of the **WSAPOLLFD** structure.

For each socket, a caller can request information on read or write status. Error conditions are always returned, so information on them need not be requested.

The WSAPOLLFD structure pointed to by the *fdarray* parameter. All sockets that do not meet these criteria and have no error condition will have the corresponding **revents** member set to 0.

A combination of the following flags can be set in the WSAPOLLFD structure for a given socket when requesting status for that socket:

FLAG	DESCRIPTION
POLLPRI	Priority data may be read without blocking. This flag is not supported by the Microsoft Winsock provider.
POLLRDBAND	Priority band (out-of-band) data may be read without blocking.
POLLRDNORM	Normal data may be read without blocking.
POLLWRNORM	Normal data may be written without blocking.

The **POLLIN** flag is defined as the combination of the **POLLRDNORM** and **POLLRDBAND** flag values. The **POLLOUT** flag is defined as the same as the **POLLWRNORM** flag value.

The WSAPOLLFD structure must only contain a combination of the above flags that are supported by the Winsock provider. Any other values are considered errors and **WSAPoll** will return **SOCKET_ERROR**. A subsequent call to the WSAGetLastError function will retrieve the extended error code of WSAEINVAL. If the **POLLPRI** flag is set on a socket for the Microsoft Winsock provider, the **WSAPoll** function will fail.

When the WSAPOLLFD structures pointed to by the *fdarray* parameter to indicate socket status:

FLAG	DESCRIPTION
POLLERR	An error has occurred.
POLLHUP	A stream-oriented connection was either disconnected or aborted.
POLLNVAL	An invalid socket was used.
POLLPRI	Priority data may be read without blocking. This flag is not returned by the Microsoft Winsock provider.
POLLRDBAND	Priority band (out-of-band) data may be read without blocking.
POLLRDNORM	Normal data may be read without blocking.
POLLWRNORM	Normal data may be written without blocking.

With regard to TCP and UDP sockets:

- WSAPOLLFD structure as normal data as **POLLRDNORM**.
- WSAPOLLFD structure, a subsequent recv operation is guaranteed to complete without blocking.
- WSAPOLLFD structure by **POLLWRNORM**.
- WSAPOLLFD structure by **POLLRDNORM**. A subsequent call to accept is guaranteed to complete without blocking.
- WSAPOLLFD structure by **POLLRDBAND**.
- WSAPOLLFD structure when a remote peer shuts down a send operation (a TCP FIN was received). A subsequent recv function request will return zero bytes.
- WSAPOLLFD structure when the remote peer initiates a graceful disconnect.
- WSAPOLLFD structure returned when a remote peer suddenly disconnects.
- WSAPOLLFD structure when the local socket is closed.

The number of elements (not sockets) in *fdarray* is indicated by *nfds*. Members of *fdarray* which have their **fd** member set to a negative value are ignored and their **revents** will be set to **POLLNVAL** upon return. This behavior is useful to an application which maintains a fixed *fdarray* allocation and will not compact the array to remove unused entries or to reallocate memory. It is not necessary to clear **revents** for any element prior to calling **WSAPoll**.

The timeout argument specifies how long the function is to wait before returning. A positive value contains the number of milliseconds to wait before returning. A zero value forces **WSAPoll** to return immediately, and a negative value indicates that **WSAPoll** should wait indefinitely.

> **Note** When issuing a blocking Winsock call such as **WSAPoll** with the *timeout* parameter set to a negative number, Winsock may need to wait for a network event before the call can complete. Winsock performs an alertable wait in this situation, which can be interrupted by an asynchronous procedure call (APC) scheduled on the same thread. Issuing another blocking Winsock call inside an APC that interrupted an ongoing blocking Winsock call on the same thread will lead to undefined behavior, and must never be attempted by Winsock clients.

> **Note** As of Windows 10 version 2004, when a TCP socket fails to connect, (POLLHUP \| POLLERR \| POLLWRNORM) is indicated.

Windows 8.1 and **Windows Server 2012 R2**: This function is supported for Windows Store apps on Windows 8.1, Windows Server 2012 R2, and later.

Requirements

Minimum supported client	Windows 8.1, Windows Vista [desktop apps | UWP apps]
Minimum supported server	Windows Server 2008 [desktop apps | UWP apps]
Target Platform	Windows
Header	winsock2.h (include Winsock2.h)
Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

WSAGetLastError

WSAPOLLFD

WSAPOLLFD structure (winsock2.h)

10/22/2022 • 2 minutes to read • Edit Online

The **WSAPOLLFD** structure stores socket information used by the WSAPoll function.

Syntax

```
typedef struct pollfd {
  SOCKET fd;
  SHORT  events;
  SHORT  revents;
} WSAPOLLFD, *PWSAPOLLFD, *LPWSAPOLLFD;
```

Members

`fd`

Type: **SOCKET**

The identifier of the socket for which to find status. This parameter is ignored if set to a negative value. See Remarks.

`events`

Type: **short**

A set of flags indicating the type of status being requested. This must be one or more of the following.

FLAG	MEANING
POLLPRI	Priority data may be read without blocking. This flag is not supported by the Microsoft Winsock provider.
POLLRDBAND	Priority band (out-of-band) data can be read without blocking.
POLLRDNORM	Normal data can be read without blocking.
POLLWRNORM	Normal data can be written without blocking.

The POLLIN flag is defined as the combination of the **POLLRDNORM** and **POLLRDBAND** flag values. The POLLOUT flag is defined as the same as the **POLLWRNORM** flag value.

`revents`

Type: **short**

A set of flags that indicate, upon return from the WSAPoll function call, the results of the status query. This can a combination of the following flags.

FLAG	DESCRIPTION

POLLERR	An error has occurred.
POLLHUP	A stream-oriented connection was either disconnected or aborted.
POLLNVAL	An invalid socket was used.
POLLPRI	Priority data may be read without blocking. This flag is not returned by the Microsoft Winsock provider.
POLLRDBAND	Priority band (out-of-band) data may be read without blocking.
POLLRDNORM	Normal data may be read without blocking.
POLLWRNORM	Normal data may be written without blocking.

The POLLIN flag is defined as the combination of the **POLLRDNORM** and **POLLRDBAND** flag values. The POLLOUT flag is defined as the same as the **POLLWRNORM** flag value.

For sockets that do not satisfy the status query, and have no error, the **revents** member is set to zero upon return.

Remarks

The **WSAPOLLFD** structure is defined on Windows Vista and later.

The **WSAPOLLFD** structure is used by the WSAPoll function to determine the status of one or more sockets. The set of sockets for which status is requested is specified in *fdarray* parameter, which is an array of **WSAPOLLFD** structures. An application sets the appropriate flags in the **events** member of the **WSAPOLLFD** structure to specify the type of status requested for each corresponding socket. The **WSAPoll** function returns the status of a socket in the **revents** member of the **WSAPOLLFD** structure.

If the **fd** member of the **WSAPOLLFD** structure is set to a negative value, the structure is ignored by the WSAPoll function call, and the **revents** member is cleared upon return. This is useful to applications that maintain a fixed allocation for the *fdarray* parameter of **WSAPoll**; such applications need not waste resources compacting elements of the array for unused entries or reallocating memory. It is unnecessary to clear the **revents** member prior to calling the **WSAPoll** function.

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Header	winsock2.h

See also

WSAPoll

accept

connect

recv

WSAPROTOCOL_INFOA structure (winsock2.h)

10/22/2022 • 10 minutes to read • Edit Online

The **WSAPROTOCOL_INFO** structure is used to store or retrieve complete information for a given protocol.

Syntax

```
typedef struct _WSAPROTOCOL_INFOA {
  DWORD            dwServiceFlags1;
  DWORD            dwServiceFlags2;
  DWORD            dwServiceFlags3;
  DWORD            dwServiceFlags4;
  DWORD            dwProviderFlags;
  GUID             ProviderId;
  DWORD            dwCatalogEntryId;
  WSAPROTOCOLCHAIN ProtocolChain;
  int              iVersion;
  int              iAddressFamily;
  int              iMaxSockAddr;
  int              iMinSockAddr;
  int              iSocketType;
  int              iProtocol;
  int              iProtocolMaxOffset;
  int              iNetworkByteOrder;
  int              iSecurityScheme;
  DWORD            dwMessageSize;
  DWORD            dwProviderReserved;
  CHAR             szProtocol[WSAPROTOCOL_LEN + 1];
} WSAPROTOCOL_INFOA, *LPWSAPROTOCOL_INFOA;
```

Members

`dwServiceFlags1`

Type: **DWORD**

A bitmask that describes the services provided by the protocol. The possible values for this member are defined in the *Winsock2.h* header file.

The following values are possible.

VALUE	MEANING
XP1_CONNECTIONLESS 0x00000001	Provides connectionless (datagram) service. If not set, the protocol supports connection-oriented data transfer.
XP1_GUARANTEED_DELIVERY 0x00000002	Guarantees that all data sent will reach the intended destination.

XP1_GUARANTEED_ORDER 0x00000004	Guarantees that data only arrives in the order in which it was sent and that it is not duplicated. This characteristic does not necessarily mean that the data is always delivered, but that any data that is delivered is delivered in the order in which it was sent.
XP1_MESSAGE_ORIENTED 0x00000008	Honors message boundaries—as opposed to a stream-oriented protocol where there is no concept of message boundaries.
XP1_PSEUDO_STREAM 0x00000010	A message-oriented protocol, but message boundaries are ignored for all receipts. This is convenient when an application does not desire message framing to be done by the protocol.
XP1_GRACEFUL_CLOSE 0x00000020	Supports two-phase (graceful) close. If not set, only abortive closes are performed.
XP1_EXPEDITED_DATA 0x00000040	Supports expedited (urgent) data.
XP1_CONNECT_DATA 0x00000080	Supports connect data.
XP1_DISCONNECT_DATA 0x00000100	Supports disconnect data.
XP1_SUPPORT_BROADCAST 0x00000200	Supports a broadcast mechanism.
XP1_SUPPORT_MULTIPOINT 0x00000400	Supports a multipoint or multicast mechanism. Control and data plane attributes are indicated below.
XP1_MULTIPOINT_CONTROL_PLANE 0x00000800	Indicates whether the control plane is rooted (value = 1) or nonrooted (value = 0).
XP1_MULTIPOINT_DATA_PLANE 0x00001000	Indicates whether the data plane is rooted (value = 1) or nonrooted (value = 0).
XP1_QOS_SUPPORTED 0x00002000	Supports quality of service requests.
XP1_INTERRUPT	Bit is reserved.

XP1_UNI_SEND 0x00008000	Protocol is unidirectional in the send direction.
XP1_UNI_RECV 0x00010000	Protocol is unidirectional in the recv direction.
XP1_IFS_HANDLES 0x00020000	Socket descriptors returned by the provider are operating system Installable File System (IFS) handles.
XP1_PARTIAL_MESSAGE 0x00040000	The MSG_PARTIAL flag is supported in WSASend and WSASendTo.
XP1_SAN_SUPPORT_SDP 0x00080000	The protocol provides support for SAN. This value is supported on Windows 7 and Windows Server 2008 R2.

> **Note** Only one of XP1_UNI_SEND or XP1_UNI_RECV values may be set. If a protocol can be unidirectional in either direction, two WSAPROTOCOL_INFOW structures should be used. When neither bit is set, the protocol is considered to be bidirectional.

`dwServiceFlags2`

Type: **DWORD**

Reserved for additional protocol-attribute definitions.

`dwServiceFlags3`

Type: **DWORD**

Reserved for additional protocol-attribute definitions.

`dwServiceFlags4`

Type: **DWORD**

Reserved for additional protocol-attribute definitions.

`dwProviderFlags`

Type: **DWORD**

A set of flags that provides information on how this protocol is represented in the Winsock catalog. The possible values for this member are defined in the *Winsock2.h* header file.

The following flag values are possible.

VALUE	MEANING

PFL_MULTIPLE_PROTO_ENTRIES 0x00000001	Indicates that this is one of two or more entries for a single protocol (from a given provider) which is capable of implementing multiple behaviors. An example of this is SPX which, on the receiving side, can behave either as a message-oriented or a stream-oriented protocol.
PFL_RECOMMENDED_PROTO_ENTRY 0x00000002	Indicates that this is the recommended or most frequently used entry for a protocol that is capable of implementing multiple behaviors.
PFL_HIDDEN 0x00000004	Set by a provider to indicate to the Ws2_32.dll that this protocol should not be returned in the result buffer generated by WSAEnumProtocols. Obviously, a Windows Sockets 2 application should never see an entry with this bit set.
PFL_MATCHES_PROTOCOL_ZERO 0x00000008	Indicates that a value of zero in the *protocol* parameter of socket or WSASocket matches this protocol entry.
PFL_NETWORKDIRECT_PROVIDER 0x00000010	Set by a provider to indicate support for network direct access. This value is supported on Windows 7 and Windows Server 2008 R2.

`ProviderId`

Type: **GUID**

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

`dwCatalogEntryId`

Type: **DWORD**

A unique identifier assigned by the WS2_32.DLL for each **WSAPROTOCOL_INFO** structure.

`ProtocolChain`

Type: **WSAPROTOCOLCHAIN**

The WSAPROTOCOLCHAIN structure associated with the protocol. If the length of the chain is 0, this **WSAPROTOCOL_INFO** entry represents a layered protocol which has Windows Sockets 2 SPI as both its top and bottom edges. If the length of the chain equals 1, this entry represents a base protocol whose Catalog Entry identifier is in the **dwCatalogEntryId** member of the **WSAPROTOCOL_INFO** structure. If the length of the chain is larger than 1, this entry represents a protocol chain which consists of one or more layered protocols on top of a base protocol. The corresponding Catalog Entry identifiers are in the ProtocolChain.ChainEntries array starting with the layered protocol at the top (the zero element in the ProtocolChain.ChainEntries array) and ending with the base protocol. Refer to the Windows Sockets 2 Service Provider Interface specification for more information on protocol chains.

`iVersion`

Type: **int**

The protocol version identifier.

`iAddressFamily`

Type: **int**

A value to pass as the address family parameter to the socket or WSASocket function in order to open a socket for this protocol. This value also uniquely defines the structure of a protocol address for a sockaddr used by the protocol.

On the Windows SDK released for Windows Vista and later, the 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.

On versions of the Platform SDK for Windows Server 2003 and older, the possible values for the address family are defined in the *Winsock2.h* header file.

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. Note that the values for the AF_ address family and PF_ protocol family constants are identical (for example, **AF_INET** and **PF_INET**), so either constant can be used.

The table below lists common values for address family although many other values are possible.

IADDRESSFAMILY	MEANING
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 including Windows 7, Windows Server 2008, Windows Vista, Windows Server 2003, or Windows XP. The Windows Sockets provider for NetBIOS only supports sockets where the *type* parameter is set to **SOCK_DGRAM**. The Windows Sockets provider for NetBIOS is not directly related to the NetBIOS 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 supported on Windows XP with SP2 or later if the computer has a Bluetooth adapter and driver installed.

`iMaxSockAddr`

Type: **int**

The maximum address length, in bytes.

`iMinSockAddr`

Type: **int**

The minimum address length, in bytes.

`iSocketType`

Type: **int**

A value to pass as the socket type parameter to the socket or WSASocket function in order to open a socket for this protocol. Possible values for the socket type are defined in the *Winsock2.h* header file.

The following table lists the possible values for the **iSocketType** member supported for Windows Sockets 2:

ISOCKETTYPE	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 IP_HDRINCL socket option must be set on the socket. To manipulate the IPv6 header, the IPV6_HDRINCL 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 reliable multicast programming. This 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.

`iProtocol`

Type: **int**

A value to pass as the *protocol* parameter to the socket or WSASocket function in order to open a socket for this protocol. The possible options for the **iProtocol** member are specific to the address family and socket type specified.

On the Windows SDK released for Windows Vista and later, this member can be one of the values from 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.

On versions of the Platform SDK for Windows Server 2003 and earlier, the possible values for the **iProtocol** member are defined in the *Winsock2.h* and *Wsrm.h* header files.

The table below lists common values for the **iProtocol** although many other values are possible.

IPROTOCOL	MEANING
IPPROTO_ICMP 1	The Internet Control Message Protocol (ICMP). This value is supported on Windows XP and later.
IPPROTO_IGMP 2	The Internet Group Management Protocol (IGMP). This value is supported on Windows XP and later.

BTHPROTO_RFCOMM 3	The Bluetooth Radio Frequency Communications (Bluetooth RFCOMM) protocol. This value is supported on Windows XP with SP2 or later.
IPPROTO_TCP 6	The Transmission Control Protocol (TCP).
IPPROTO_UDP 17	The User Datagram Protocol (UDP).
IPPROTO_ICMPV6 58	The Internet Control Message Protocol Version 6 (ICMPv6). This value is supported on Windows XP and later.
IPPROTO_RM 113	The PGM protocol for reliable multicast. On the Windows SDK released for Windows Vista and later, this protocol is also called **IPPROTO_PGM**. This value is only supported if the Reliable Multicast Protocol is installed.

`iProtocolMaxOffset`

Type: **int**

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

`iNetworkByteOrder`

Type: **int**

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

`iSecurityScheme`

Type: **int**

The type of security scheme employed (if any). A value of SECURITY_PROTOCOL_NONE (0) is used for protocols that do not incorporate security provisions.

`dwMessageSize`

Type: **DWORD**

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

VALUE	MEANING

0	The protocol is stream-oriented and hence the concept of message size is not relevant.
0x1	The maximum outbound (send) message size is dependent on the underlying network MTU (maximum sized transmission unit) and hence cannot be known until after a socket is bound. Applications should use getsockopt to retrieve the value of SO_MAX_MSG_SIZE after the socket has been bound to a local address.
0xFFFFFFFF	The protocol is message-oriented, but there is no maximum limit to the size of messages that may be transmitted.

`dwProviderReserved`

Type: **DWORD**

Reserved for use by service providers.

`szProtocol[WSAPROTOCOL_LEN + 1]`

Type: **TCHAR[WSAPROTOCOL_LEN+1]**

An array of characters that contains a human-readable name identifying the protocol, for example "MSAFD Tcpip [UDP/IP]". The maximum number of characters allowed is WSAPROTOCOL_LEN, which is defined to be 255.

Remarks

> **NOTE**
> The winsock2.h header defines WSAPROTOCOL_INFO as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see Conventions for Function Prototypes.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	winsock2.h

See also

WSAEnumProtocols

WSAPROTOCOLCHAIN

WSAPROTOCOL_INFOW

WSASend

WSASendTo

WSASocket

WSCInstallProvider

WSCInstallProvider64_32

WSCUpdateProvider

WSCUpdateProvider32

getsockopt

socket

WSAPROTOCOL_INFOW structure (winsock2.h)

10/22/2022 • 11 minutes to read • Edit Online

The **WSAPROTOCOL_INFOW** structure is used to store or retrieve complete information for a given protocol. The protocol name is represented as an array of Unicode characters.

Syntax

```
typedef struct _WSAPROTOCOL_INFOW {
  DWORD            dwServiceFlags1;
  DWORD            dwServiceFlags2;
  DWORD            dwServiceFlags3;
  DWORD            dwServiceFlags4;
  DWORD            dwProviderFlags;
  GUID             ProviderId;
  DWORD            dwCatalogEntryId;
  WSAPROTOCOLCHAIN ProtocolChain;
  int              iVersion;
  int              iAddressFamily;
  int              iMaxSockAddr;
  int              iMinSockAddr;
  int              iSocketType;
  int              iProtocol;
  int              iProtocolMaxOffset;
  int              iNetworkByteOrder;
  int              iSecurityScheme;
  DWORD            dwMessageSize;
  DWORD            dwProviderReserved;
  WCHAR            szProtocol[WSAPROTOCOL_LEN + 1];
} WSAPROTOCOL_INFOW, *LPWSAPROTOCOL_INFOW;
```

Members

`dwServiceFlags1`

Type: **DWORD**

A bitmask that describes the services provided by the protocol. The possible values for this member are defined in the *Winsock2.h* header file.

The following values are possible.

VALUE	MEANING
XP1_CONNECTIONLESS 0x00000001	Provides connectionless (datagram) service. If not set, the protocol supports connection-oriented data transfer.
XP1_GUARANTEED_DELIVERY 0x00000002	Guarantees that all data sent will reach the intended destination.

XP1_GUARANTEED_ORDER 0x00000004	Guarantees that data only arrives in the order in which it was sent and that it is not duplicated. This characteristic does not necessarily mean that the data is always delivered, but that any data that is delivered is delivered in the order in which it was sent.
XP1_MESSAGE_ORIENTED 0x00000008	Honors message boundaries—as opposed to a stream-oriented protocol where there is no concept of message boundaries.
XP1_PSEUDO_STREAM 0x00000010	A message-oriented protocol, but message boundaries are ignored for all receipts. This is convenient when an application does not desire message framing to be done by the protocol.
XP1_GRACEFUL_CLOSE 0x00000020	Supports two-phase (graceful) close. If not set, only abortive closes are performed.
XP1_EXPEDITED_DATA 0x00000040	Supports expedited (urgent) data.
XP1_CONNECT_DATA 0x00000080	Supports connect data.
XP1_DISCONNECT_DATA 0x00000100	Supports disconnect data.
XP1_SUPPORT_BROADCAST 0x00000200	Supports a broadcast mechanism.
XP1_SUPPORT_MULTIPOINT 0x00000400	Supports a multipoint or multicast mechanism. Control and data plane attributes are indicated below.
XP1_MULTIPOINT_CONTROL_PLANE 0x00000800	Indicates whether the control plane is rooted (value = 1) or nonrooted (value = 0).
XP1_MULTIPOINT_DATA_PLANE 0x00001000	Indicates whether the data plane is rooted (value = 1) or nonrooted (value = 0).
XP1_QOS_SUPPORTED 0x00002000	Supports quality of service requests.
XP1_INTERRUPT	Bit is reserved.

XP1_UNI_SEND 0x00008000	Protocol is unidirectional in the send direction.
XP1_UNI_RECV 0x00010000	Protocol is unidirectional in the recv direction.
XP1_IFS_HANDLES 0x00020000	Socket descriptors returned by the provider are operating system Installable File System (IFS) handles.
XP1_PARTIAL_MESSAGE 0x00040000	The MSG_PARTIAL flag is supported in WSASend and WSASendTo.
XP1_SAN_SUPPORT_SDP 0x00080000	The protocol provides support for SAN. This value is supported on Windows 7 and Windows Server 2008 R2.

> **Note** Only one of XP1_UNI_SEND or XP1_UNI_RECV values may be set. If a protocol can be unidirectional in either direction, two **WSAPROTOCOL_INFOW** structures should be used. When neither bit is set, the protocol is considered to be bidirectional.

`dwServiceFlags2`

Type: **DWORD**

Reserved for additional protocol-attribute definitions.

`dwServiceFlags3`

Type: **DWORD**

Reserved for additional protocol-attribute definitions.

`dwServiceFlags4`

Type: **DWORD**

Reserved for additional protocol-attribute definitions.

`dwProviderFlags`

Type: **DWORD**

A set of flags that provides information on how this protocol is represented in the Winsock catalog. The possible values for this member are defined in the *Winsock2.h* header file.

The following values are possible.

VALUE	MEANING

PFL_MULTIPLE_PROTO_ENTRIES 0x00000001	Indicates that this is one of two or more entries for a single protocol (from a given provider) which is capable of implementing multiple behaviors. An example of this is SPX which, on the receiving side, can behave either as a message-oriented or a stream-oriented protocol.
PFL_RECOMMENDED_PROTO_ENTRY 0x00000002	Indicates that this is the recommended or most frequently used entry for a protocol that is capable of implementing multiple behaviors.
PFL_HIDDEN 0x00000004	Set by a provider to indicate to the Ws2_32.dll that this protocol should not be returned in the result buffer generated by WSAEnumProtocols. Obviously, a Windows Sockets 2 application should never see an entry with this bit set.
PFL_MATCHES_PROTOCOL_ZERO 0x00000008	Indicates that a value of zero in the *protocol* parameter of socket or WSASocket matches this protocol entry.
PFL_NETWORKDIRECT_PROVIDER 0x00000010	Set by a provider to indicate support for network direct access. This value is supported on Windows 7 and Windows Server 2008 R2.

`ProviderId`

Type: **GUID**

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

`dwCatalogEntryId`

Type: **DWORD**

A unique identifier assigned by the WS2_32.DLL for each WSAPROTOCOL_INFO structure.

`ProtocolChain`

Type: **WSAPROTOCOLCHAIN**

The WSAPROTOCOLCHAIN structure associated with the protocol. If the length of the chain is 0, this WSAPROTOCOL_INFO entry represents a layered protocol which has Windows Sockets 2 SPI as both its top and bottom edges. If the length of the chain equals 1, this entry represents a base protocol whose Catalog Entry identifier is in the **dwCatalogEntryId** member of the **WSAPROTOCOL_INFO** structure. If the length of the chain is larger than 1, this entry represents a protocol chain which consists of one or more layered protocols on top of a base protocol. The corresponding Catalog Entry identifiers are in the ProtocolChain.ChainEntries array starting with the layered protocol at the top (the zero element in the ProtocolChain.ChainEntries array) and ending with the base protocol. Refer to the Windows Sockets 2 Service Provider Interface specification for more information on protocol chains.

`iVersion`

Type: **int**

The protocol version identifier.

`iAddressFamily`

Type: **int**

A value to pass as the address family parameter to the socket or WSASocket function in order to open a socket for this protocol. This value also uniquely defines the structure of a protocol address for a sockaddr used by the protocol.

On the Windows SDK released for Windows Vista and later, the 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.

On versions of the Platform SDK for Windows Server 2003 and older, the possible values for the address family are defined in the *Winsock2.h* header file.

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. Note that the values for the AF_ address family and PF_ protocol family constants are identical (for example, **AF_INET** and **PF_INET**), so either constant can be used.

The table below lists common values for address family although many other values are possible.

IADDRESSFAMILY	MEANING
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 including Windows 7, Windows Server 2008, Windows Vista, Windows Server 2003, or Windows XP. The Windows Sockets provider for NetBIOS only supports sockets where the *type* parameter is set to **SOCK_DGRAM**. The Windows Sockets provider for NetBIOS is not directly related to the NetBIOS 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 supported on Windows XP with SP2 or later if the computer has a Bluetooth adapter and driver installed.

`iMaxSockAddr`

Type: **int**

The maximum address length, in bytes.

`iMinSockAddr`

Type: **int**

The minimum address length, in bytes.

`iSocketType`

Type: **int**

A value to pass as the socket type parameter to the socket or WSASocket function in order to open a socket for this protocol. Possible values for the socket type are defined in the *Winsock2.h* header file.

The following table lists the possible values for the **iSocketType** member supported for Windows Sockets 2:

ISOCKETTYPE	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 IP_HDRINCL socket option must be set on the socket. To manipulate the IPv6 header, the IPV6_HDRINCL 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 reliable multicast programming. This 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.

`iProtocol`

Type: **int**

A value to pass as the *protocol* parameter to the socket or WSASocket function in order to open a socket for this protocol. The possible options for the **iProtocol** member are specific to the address family and socket type specified.

On the Windows SDK released for Windows Vista and later, this member can be one of the values from 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.

On versions of the Platform SDK for Windows Server 2003 and earlier, the possible values for the **iProtocol** member are defined in the *Winsock2.h* and *Wsrm.h* header files.

The table below lists common values for the **iProtocol** although many other values are possible.

IPROTOCOL	MEANING
IPPROTO_ICMP 1	The Internet Control Message Protocol (ICMP). This value is supported on Windows XP and later.
IPPROTO_IGMP 2	The Internet Group Management Protocol (IGMP). This value is supported on Windows XP and later.

BTHPROTO_RFCOMM 3	The Bluetooth Radio Frequency Communications (Bluetooth RFCOMM) protocol. This value is supported on Windows XP with SP2 or later.
IPPROTO_TCP 6	The Transmission Control Protocol (TCP).
IPPROTO_UDP 17	The User Datagram Protocol (UDP).
IPPROTO_ICMPV6 58	The Internet Control Message Protocol Version 6 (ICMPv6). This value is supported on Windows XP and later.
IPPROTO_RM 113	The PGM protocol for reliable multicast. On the Windows SDK released for Windows Vista and later, this protocol is also called **IPPROTO_PGM**. This value is only supported if the Reliable Multicast Protocol is installed.

`iProtocolMaxOffset`

Type: **int**

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

`iNetworkByteOrder`

Type: **int**

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

`iSecurityScheme`

Type: **int**

The type of security scheme employed (if any). A value of SECURITY_PROTOCOL_NONE (0) is used for protocols that do not incorporate security provisions.

`dwMessageSize`

Type: **DWORD**

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

VALUE	MEANING

0	The protocol is stream-oriented and hence the concept of message size is not relevant.
0x1	The maximum outbound (send) message size is dependent on the underlying network MTU (maximum sized transmission unit) and hence cannot be known until after a socket is bound. Applications should use getsockopt to retrieve the value of SO_MAX_MSG_SIZE after the socket has been bound to a local address.
0xFFFFFFFF	The protocol is message-oriented, but there is no maximum limit to the size of messages that may be transmitted.

`dwProviderReserved`

Type: **DWORD**

Reserved for use by service providers.

`szProtocol[WSAPROTOCOL_LEN + 1]`

Type: **WCHAR[WSAPROTOCOL_LEN+1]**

An array of Unicode characters that contains a human-readable name identifying the protocol, for example "MSAFD Tcpip [UDP/IP]". The maximum number of characters allowed is WSAPROTOCOL_LEN, which is defined to be 255.

Remarks

> **NOTE**
> The winsock2.h header defines WSAPROTOCOL_INFO as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see Conventions for Function Prototypes.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	winsock2.h

See also

WSAPROTOCOLCHAIN

WSAPROTOCOL_INFO

WSASend

WSASendTo

WSASocket

WSCEnumProtocols

WSCEnumProtocols32

getsockopt

socket

WSAPROTOCOLCHAIN structure (winsock2.h)

10/22/2022 • 2 minutes to read • Edit Online

The **WSAPROTOCOLCHAIN** structure contains a counted list of Catalog Entry identifiers that comprise a protocol chain.

Syntax

```
typedef struct _WSAPROTOCOLCHAIN {
  int   ChainLen;
  DWORD ChainEntries[MAX_PROTOCOL_CHAIN];
} WSAPROTOCOLCHAIN, *LPWSAPROTOCOLCHAIN;
```

Members

`ChainLen`

Length of the chain, in bytes. The following settings apply:

Setting **ChainLen** to zero indicates a layered protocol

Setting **ChainLen** to one indicates a base protocol

Setting **ChainLen** to greater than one indicates a protocol chain

`ChainEntries[MAX_PROTOCOL_CHAIN]`

Array of protocol chain entries.

Remarks

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

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	winsock2.h

See also

WSAEnumProtocols

WSAProviderConfigChange function (winsock2.h)

10/22/2022 • 2 minutes to read • Edit Online

The **WSAProviderConfigChange** function notifies the application when the provider configuration is changed.

Syntax

```
INT WSAAPI WSAProviderConfigChange(
  [in, out] LPHANDLE                          lpNotificationHandle,
  [in]      LPWSAOVERLAPPED                   lpOverlapped,
  [in]      LPWSAOVERLAPPED_COMPLETION_ROUTINE lpCompletionRoutine
);
```

Parameters

`[in, out] lpNotificationHandle`

Pointer to notification handle. If the notification handle is set to **NULL** (the handle value not the pointer itself), this function returns a notification handle in the location pointed to by *lpNotificationHandle*.

`[in] lpOverlapped`

Pointer to a WSAOVERLAPPED structure.

`[in] lpCompletionRoutine`

Type: _In_opt_ **LPWSAOVERLAPPED_COMPLETION_ROUTINE**

Pointer to the completion routine called when the provider change notification is received.

Return value

If no error occurs the **WSAProviderConfigChange** returns 0. Otherwise, a value of SOCKET_ERROR is returned and a specific error code may be retrieved by calling WSAGetLastError. The error code WSA_IO_PENDING indicates that the overlapped operation has been successfully initiated and that completion (and thus change event) will be indicated at a later time.

ERROR CODE	MEANING
WSANOTINITIALISED	A successful WSAStartup call must occur before using this function.
WSAENETDOWN	The network subsystem has failed.
WSA_NOT_ENOUGH_MEMORY	Not enough free memory available to complete the operation.
WSA_INVALID_HANDLE	Value pointed by *lpNotificationHandle* parameter is not a valid notification handle.

WSAEOPNOTSUPP	Current operating system environment does not support provider installation or removal without restart.

Remarks

The **WSAProviderConfigChange** function notifies the application of provider (both transport and namespace) installation or removal in Windows operating environments that support such configuration change without requiring a restart. When called for the first time (*lpNotificationHandle* parameter points to **NULL** handle), this function completes immediately and returns notification handle in the location pointed by *lpNotificationHandle* that can be used in subsequent calls to receive notifications of provider installation and removal. The second and any subsequent calls only complete when provider information changes since the time the call was made It is expected (but not required) that the application uses overlapped I/O on second and subsequent calls to **WSAProviderConfigChange**, in which case the call will return immediately and application will be notified of provider configuration changes using the completion mechanism chosen through specified overlapped completion parameters.

Notification handle returned by **WSAProviderConfigChange** is like any regular operating system handle that should be closed (when no longer needed) using Windows CloseHandle call.

The following sequence of actions can be used to guarantee that application always has current protocol configuration information:

- Call **WSAProviderConfigChange**
- Call WSAEnumProtocols and/or WSAEnumNameSpaceProviders
- Whenever **WSAProviderConfigChange** notifies application of provider configuration change (through blocking or overlapped I/O), the whole sequence of actions should be repeated.

Windows 8.1 and **Windows Server 2012 R2**: This function is supported for Windows Store apps on Windows 8.1, Windows Server 2012 R2, and later.

Requirements

Minimum supported client	Windows 8.1, Windows Vista [desktop apps | UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps | UWP apps]
Target Platform	Windows
Header	winsock2.h
Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

CloseHandle

WSAEnumNameSpaceProviders

WSAEnumProtocols

Winsock Functions

Winsock Reference

WSAQUERYSET2A structure (winsock2.h)

10/22/2022 • 3 minutes to read • Edit Online

The **WSAQUERYSET2** structure provides relevant information about a given service, including service class ID, service name , applicable namespace identifier and protocol information, as well as a set of transport addresses at which the service listens.

Syntax

```
typedef struct _WSAQuerySet2A {
  DWORD         dwSize;
  LPSTR         lpszServiceInstanceName;
  LPWSAVERSION  lpVersion;
  LPSTR         lpszComment;
  DWORD         dwNameSpace;
  LPGUID        lpNSProviderId;
  LPSTR         lpszContext;
  DWORD         dwNumberOfProtocols;
  LPAFPROTOCOLS lpafpProtocols;
  LPSTR         lpszQueryString;
  DWORD         dwNumberOfCsAddrs;
  LPCSADDR_INFO lpcsaBuffer;
  DWORD         dwOutputFlags;
  LPBLOB        lpBlob;
} WSAQUERYSET2A, *PWSAQUERYSET2A, *LPWSAQUERYSET2A;
```

Members

`dwSize`

Type: **DWORD**

The size, in bytes, of the **WSAQUERYSET2** structure. This member is used as a versioning mechanism since the size of the **WSAQUERYSET2** structure may change on later versions of Windows.

`lpszServiceInstanceName`

Type: **LPTSTR**

A pointer to an optional **NULL**-terminated string that contains service name. The semantics for using wildcards within the string are not defined, but can be supported by certain namespace providers.

`lpVersion`

Type: **LPWSAVERSION**

A pointer to an optional desired version number of the namespace provider. This member provides version comparison semantics (that is, the version requested must match exactly, or version must be not less than the value supplied).

`lpszComment`

Type: **LPTSTR**

This member is ignored for queries.

`dwNameSpace`

Type: **DWORD**

A namespace identifier that determines which namespace providers are queried. Passing a specific namespace identifier will result in only namespace providers that support the specified namespace being queried. Specifying **NS_ALL** will result in all installed and active namespace providers being queried.

Options for the **dwNameSpace** member are listed in the *Winsock2.h* include file. Several new namespace providers are included with Windows Vista and later. Other namespace providers can be installed, so the following possible values are only those commonly available. Many other values are possible.

VALUE	MEANING
NS_ALL	All installed and active namespaces.
NS_BTH	The Bluetooth namespace. This namespace identifier is supported on Windows Vista and later.
NS_DNS	The domain name system (DNS) namespace.
NS_EMAIL	The email namespace. This namespace identifier is supported on Windows Vista and later.
NS_NLA	The network location awareness (NLA) namespace. This namespace identifier is supported on Windows XP and later.
NS_PNRPNAME	The peer-to-peer name space for a specific peer name. This namespace identifier is supported on Windows Vista and later.
NS_PNRPCLOUD	The peer-to-peer name space for a collection of peer names. This namespace identifier is supported on Windows Vista and later.

`lpNSProviderId`

Type: **LPGUID**

A pointer to an optional GUID of a specific namespace provider to query in the case where multiple namespace providers are registered under a single namespace such as **NS_DNS**. Passing the GUID for a specific namespace provider will result in only the specified namespace provider being queried. The WSAEnumNameSpaceProviders and WSAEnumNameSpaceProvidersEx functions can be called to retrieve the GUID for a namespace provider.

`lpszContext`

Type: **LPTSTR**

A pointer to an optional starting point of the query in a hierarchical namespace.

`dwNumberOfProtocols`

Type: **DWORD**

The size, in bytes, of the protocol constraint array. This member can be zero.

`lpafpProtocols`

Type: **LPAFPROTOCOLS**

A pointer to an optional array of AFPROTOCOLS structures. Only services that utilize these protocols will be returned.

`lpszQueryString`

Type: **LPTSTR**

A pointer to an optional **NULL**-terminated query string. Some namespaces, such as Whois++, support enriched SQL-like queries that are contained in a simple text string. This parameter is used to specify that string.

`dwNumberOfCsAddrs`

Type: **DWORD**

This member is ignored for queries.

`lpcsaBuffer`

Type: **LPCSADDR_INFO**

This member is ignored for queries.

`dwOutputFlags`

Type: **DWORD**

This member is ignored for queries.

`lpBlob`

Type: **LPBLOB**

An optional pointer to data that is used to query or set provider-specific namespace information. The format of this information is specific to the namespace provider.

Remarks

The **WSAQUERYSET2** structure is used as part of the namespace service provider version-2 (NSPv2) architecture available on Windows Vista and later.

On Windows Vista and Windows Server 2008, the **WSAQUERYSET2** structure can only be used for operations on NS_EMAIL namespace providers.

> **NOTE**
>
> The winsock2.h header defines WSAQUERYSET2 as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see Conventions for Function Prototypes.

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Header	winsock2.h

See also

WSAEnumNameSpaceProviders

WSAEnumNameSpaceProvidersEx

WSASetService

WSAQUERYSET2W structure (winsock2.h)

10/22/2022 • 3 minutes to read • Edit Online

The **WSAQUERYSET2** structure provides relevant information about a given service, including service class ID, service name , applicable namespace identifier and protocol information, as well as a set of transport addresses at which the service listens.

Syntax

```
typedef struct _WSAQuerySet2W {
  DWORD         dwSize;
  LPWSTR        lpszServiceInstanceName;
  LPWSAVERSION  lpVersion;
  LPWSTR        lpszComment;
  DWORD         dwNameSpace;
  LPGUID        lpNSProviderId;
  LPWSTR        lpszContext;
  DWORD         dwNumberOfProtocols;
  LPAFPROTOCOLS lpafpProtocols;
  LPWSTR        lpszQueryString;
  DWORD         dwNumberOfCsAddrs;
  LPCSADDR_INFO lpcsaBuffer;
  DWORD         dwOutputFlags;
  LPBLOB        lpBlob;
} WSAQUERYSET2W, *PWSAQUERYSET2W, *LPWSAQUERYSET2W;
```

Members

`dwSize`

Type: **DWORD**

The size, in bytes, of the **WSAQUERYSET2** structure. This member is used as a versioning mechanism since the size of the **WSAQUERYSET2** structure may change on later versions of Windows.

`lpszServiceInstanceName`

Type: **LPTSTR**

A pointer to an optional **NULL**-terminated string that contains service name. The semantics for using wildcards within the string are not defined, but can be supported by certain namespace providers.

`lpVersion`

Type: **LPWSAVERSION**

A pointer to an optional desired version number of the namespace provider. This member provides version comparison semantics (that is, the version requested must match exactly, or version must be not less than the value supplied).

`lpszComment`

Type: **LPTSTR**

This member is ignored for queries.

`dwNameSpace`

Type: **DWORD**

A namespace identifier that determines which namespace providers are queried. Passing a specific namespace identifier will result in only namespace providers that support the specified namespace being queried. Specifying **NS_ALL** will result in all installed and active namespace providers being queried.

Options for the **dwNameSpace** member are listed in the *Winsock2.h* include file. Several new namespace providers are included with Windows Vista and later. Other namespace providers can be installed, so the following possible values are only those commonly available. Many other values are possible.

VALUE	MEANING
NS_ALL	All installed and active namespaces.
NS_BTH	The Bluetooth namespace. This namespace identifier is supported on Windows Vista and later.
NS_DNS	The domain name system (DNS) namespace.
NS_EMAIL	The email namespace. This namespace identifier is supported on Windows Vista and later.
NS_NLA	The network location awareness (NLA) namespace. This namespace identifier is supported on Windows XP and later.
NS_PNRPNAME	The peer-to-peer name space for a specific peer name. This namespace identifier is supported on Windows Vista and later.
NS_PNRPCLOUD	The peer-to-peer name space for a collection of peer names. This namespace identifier is supported on Windows Vista and later.

`lpNSProviderId`

Type: **LPGUID**

A pointer to an optional GUID of a specific namespace provider to query in the case where multiple namespace providers are registered under a single namespace such as **NS_DNS**. Passing the GUID for a specific namespace provider will result in only the specified namespace provider being queried. The WSAEnumNameSpaceProviders and WSAEnumNameSpaceProvidersEx functions can be called to retrieve the GUID for a namespace provider.

`lpszContext`

Type: **LPTSTR**

A pointer to an optional starting point of the query in a hierarchical namespace.

`dwNumberOfProtocols`

Type: **DWORD**

The size, in bytes, of the protocol constraint array. This member can be zero.

`lpafpProtocols`

Type: **LPAFPROTOCOLS**

A pointer to an optional array of AFPROTOCOLS structures. Only services that utilize these protocols will be returned.

`lpszQueryString`

Type: **LPTSTR**

A pointer to an optional **NULL**-terminated query string. Some namespaces, such as Whois++, support enriched SQL-like queries that are contained in a simple text string. This parameter is used to specify that string.

`dwNumberOfCsAddrs`

Type: **DWORD**

This member is ignored for queries.

`lpcsaBuffer`

Type: **LPCSADDR_INFO**

This member is ignored for queries.

`dwOutputFlags`

Type: **DWORD**

This member is ignored for queries.

`lpBlob`

Type: **LPBLOB**

An optional pointer to data that is used to query or set provider-specific namespace information. The format of this information is specific to the namespace provider.

Remarks

The **WSAQUERYSET2** structure is used as part of the namespace service provider version-2 (NSPv2) architecture available on Windows Vista and later.

On Windows Vista and Windows Server 2008, the **WSAQUERYSET2** structure can only be used for operations on NS_EMAIL namespace providers.

> **NOTE**
>
> The winsock2.h header defines WSAQUERYSET2 as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see Conventions for Function Prototypes.

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Header	winsock2.h

See also

WSAEnumNameSpaceProviders

WSAEnumNameSpaceProvidersEx

WSASetService

WSAQUERYSETA structure (winsock2.h)

10/22/2022 • 3 minutes to read • Edit Online

The **WSAQUERYSET** structure provides relevant information about a given service, including service class ID, service name, applicable namespace identifier and protocol information, as well as a set of transport addresses at which the service listens.

Syntax

```
typedef struct _WSAQuerySetA {
  DWORD         dwSize;
  LPSTR         lpszServiceInstanceName;
  LPGUID        lpServiceClassId;
  LPWSAVERSION  lpVersion;
  LPSTR         lpszComment;
  DWORD         dwNameSpace;
  LPGUID        lpNSProviderId;
  LPSTR         lpszContext;
  DWORD         dwNumberOfProtocols;
  LPAFPROTOCOLS lpafpProtocols;
  LPSTR         lpszQueryString;
  DWORD         dwNumberOfCsAddrs;
  LPCSADDR_INFO lpcsaBuffer;
  DWORD         dwOutputFlags;
  LPBLOB        lpBlob;
} WSAQUERYSETA, *PWSAQUERYSETA, *LPWSAQUERYSETA;
```

Members

`dwSize`

Type: **DWORD**

The size, in bytes, of the **WSAQUERYSET** structure. This member is used as a versioning mechanism since the size of the **WSAQUERYSET** structure has changed on later versions of Windows.

`lpszServiceInstanceName`

Type: **LPTSTR**

A pointer to an optional NULL-terminated string that contains service name. The semantics for using wildcards within the string are not defined, but can be supported by certain namespace providers.

`lpServiceClassId`

Type: **LPGUID**

The GUID corresponding to the service class. This member is required to be set.

`lpVersion`

Type: **LPWSAVERSION**

A pointer to an optional desired version number of the namespace provider. This member provides version comparison semantics (that is, the version requested must match exactly, or version must be not less than the value supplied).

`lpszComment`

Type: **LPTSTR**

This member is ignored for queries.

`dwNameSpace`

Type: **DWORD**

A namespace identifier that determines which namespace providers are queried. Passing a specific namespace identifier will result in only namespace providers that support the specified namespace being queried. Specifying **NS_ALL** will result in all installed and active namespace providers being queried.

Options for the **dwNameSpace** member are listed in the *Winsock2.h* include file. Several new namespace providers are included with Windows Vista and later. Other namespace providers can be installed, so the following possible values are only those commonly available. Many other values are possible.

VALUE	MEANING
NS_ALL	All installed and active namespaces.
NS_BTH	The Bluetooth namespace. This namespace identifier is supported on Windows Vista and later.
NS_DNS	The domain name system (DNS) namespace.
NS_EMAIL	The email namespace. This namespace identifier is supported on Windows Vista and later.
NS_NLA	The network location awareness (NLA) namespace. This namespace identifier is supported on Windows XP and later.
NS_PNRPNAME	The peer-to-peer name space for a specific peer name. This namespace identifier is supported on Windows Vista and later.
NS_PNRPCLOUD	The peer-to-peer name space for a collection of peer names. This namespace identifier is supported on Windows Vista and later.

`lpNSProviderId`

Type: **LPGUID**

A pointer to an optional GUID of a specific namespace provider to query in the case where multiple namespace providers are registered under a single namespace such as **NS_DNS**. Passing the GUID for a specific namespace provider will result in only the specified namespace provider being queried. The WSAEnumNameSpaceProviders and WSAEnumNameSpaceProvidersEx functions can be called to retrieve the GUID for a namespace provider.

`lpszContext`

Type: **LPTSTR**

A pointer to an optional starting point of the query in a hierarchical namespace.

`dwNumberOfProtocols`

Type: **DWORD**

The size, in bytes, of the protocol constraint array. This member can be zero.

`lpafpProtocols`

Type: **LPAFPROTOCOLS**

A pointer to an optional array of AFPROTOCOLS structures. Only services that utilize these protocols will be returned.

`lpszQueryString`

Type: **LPTSTR**

A pointer to an optional NULL-terminated query string. Some namespaces, such as Whois++, support enriched SQL-like queries that are contained in a simple text string. This parameter is used to specify that string.

`dwNumberOfCsAddrs`

Type: **DWORD**

This member is ignored for queries.

`lpcsaBuffer`

Type: **LPCSADDR_INFO**

This member is ignored for queries.

`dwOutputFlags`

Type: **DWORD**

This member is ignored for queries.

`lpBlob`

Type: **LPBLOB**

An optional pointer to data that is used to query or set provider-specific namespace information. The format of this information is specific to the namespace provider.

Remarks

The **WSAQUERYSET** structure is used as part of the original namespace provider version 1 architecture available on Windows 95 and later. A newer version 2 of the namespace architecture is available on Windows Vista and later.

In most instances, applications interested in only a particular transport protocol should constrain their query by address family and protocol rather than by namespace. This would allow an application that needs to locate a TCP/IP service, for example, to have its query processed by all available namespaces such as the local hosts file, DNS, and NIS.

> **NOTE**
> The winsock2.h header defines WSAQUERYSET as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see Conventions for Function Prototypes.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	winsock2.h

See also

Bluetooth and WSAQUERYSET for Device Inquiry

Bluetooth and WSAQUERYSET for Service Inquiry

Bluetooth and WSAQUERYSET for Set Service

WSAEnumNameSpaceProviders

WSAEnumNameSpaceProvidersEx

WSALookupServiceBegin

WSALookupServiceNext

WSAQUERYSET2

WSASetService

WSAQUERYSETW structure (winsock2.h)

10/22/2022 • 3 minutes to read • Edit Online

The **WSAQUERYSET** structure provides relevant information about a given service, including service class ID, service name, applicable namespace identifier and protocol information, as well as a set of transport addresses at which the service listens.

Syntax

```
typedef struct _WSAQuerySetW {
  DWORD         dwSize;
  LPWSTR        lpszServiceInstanceName;
  LPGUID        lpServiceClassId;
  LPWSAVERSION  lpVersion;
  LPWSTR        lpszComment;
  DWORD         dwNameSpace;
  LPGUID        lpNSProviderId;
  LPWSTR        lpszContext;
  DWORD         dwNumberOfProtocols;
  LPAFPROTOCOLS lpafpProtocols;
  LPWSTR        lpszQueryString;
  DWORD         dwNumberOfCsAddrs;
  LPCSADDR_INFO lpcsaBuffer;
  DWORD         dwOutputFlags;
  LPBLOB        lpBlob;
} WSAQUERYSETW, *PWSAQUERYSETW, *LPWSAQUERYSETW;
```

Members

`dwSize`

Type: **DWORD**

The size, in bytes, of the **WSAQUERYSET** structure. This member is used as a versioning mechanism since the size of the **WSAQUERYSET** structure has changed on later versions of Windows.

`lpszServiceInstanceName`

Type: **LPTSTR**

A pointer to an optional NULL-terminated string that contains service name. The semantics for using wildcards within the string are not defined, but can be supported by certain namespace providers.

`lpServiceClassId`

Type: **LPGUID**

The GUID corresponding to the service class. This member is required to be set.

`lpVersion`

Type: **LPWSAVERSION**

A pointer to an optional desired version number of the namespace provider. This member provides version comparison semantics (that is, the version requested must match exactly, or version must be not less than the value supplied).

`lpszComment`

Type: **LPTSTR**

This member is ignored for queries.

`dwNameSpace`

Type: **DWORD**

A namespace identifier that determines which namespace providers are queried. Passing a specific namespace identifier will result in only namespace providers that support the specified namespace being queried. Specifying **NS_ALL** will result in all installed and active namespace providers being queried.

Options for the **dwNameSpace** member are listed in the *Winsock2.h* include file. Several new namespace providers are included with Windows Vista and later. Other namespace providers can be installed, so the following possible values are only those commonly available. Many other values are possible.

VALUE	MEANING
NS_ALL	All installed and active namespaces.
NS_BTH	The Bluetooth namespace. This namespace identifier is supported on Windows Vista and later.
NS_DNS	The domain name system (DNS) namespace.
NS_EMAIL	The email namespace. This namespace identifier is supported on Windows Vista and later.
NS_NLA	The network location awareness (NLA) namespace. This namespace identifier is supported on Windows XP and later.
NS_PNRPNAME	The peer-to-peer name space for a specific peer name. This namespace identifier is supported on Windows Vista and later.
NS_PNRPCLOUD	The peer-to-peer name space for a collection of peer names. This namespace identifier is supported on Windows Vista and later.

`lpNSProviderId`

Type: **LPGUID**

A pointer to an optional GUID of a specific namespace provider to query in the case where multiple namespace providers are registered under a single namespace such as **NS_DNS**. Passing the GUID for a specific namespace provider will result in only the specified namespace provider being queried. The WSAEnumNameSpaceProviders and WSAEnumNameSpaceProvidersEx functions can be called to retrieve the GUID for a namespace provider.

`lpszContext`

Type: **LPTSTR**

A pointer to an optional starting point of the query in a hierarchical namespace.

`dwNumberOfProtocols`

Type: **DWORD**

The size, in bytes, of the protocol constraint array. This member can be zero.

`lpafpProtocols`

Type: **LPAFPROTOCOLS**

A pointer to an optional array of AFPROTOCOLS structures. Only services that utilize these protocols will be returned.

`lpszQueryString`

Type: **LPTSTR**

A pointer to an optional NULL-terminated query string. Some namespaces, such as Whois++, support enriched SQL-like queries that are contained in a simple text string. This parameter is used to specify that string.

`dwNumberOfCsAddrs`

Type: **DWORD**

This member is ignored for queries.

`lpcsaBuffer`

Type: **LPCSADDR_INFO**

This member is ignored for queries.

`dwOutputFlags`

Type: **DWORD**

This member is ignored for queries.

`lpBlob`

Type: **LPBLOB**

An optional pointer to data that is used to query or set provider-specific namespace information. The format of this information is specific to the namespace provider.

Remarks

The **WSAQUERYSET** structure is used as part of the original namespace provider version 1 architecture available on Windows 95 and later. A newer version 2 of the namespace architecture is available on Windows Vista and later.

In most instances, applications interested in only a particular transport protocol should constrain their query by address family and protocol rather than by namespace. This would allow an application that needs to locate a TCP/IP service, for example, to have its query processed by all available namespaces such as the local hosts file, DNS, and NIS.

> **NOTE**
> The winsock2.h header defines WSAQUERYSET as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see Conventions for Function Prototypes.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	winsock2.h

See also

Bluetooth and WSAQUERYSET for Device Inquiry

Bluetooth and WSAQUERYSET for Service Inquiry

Bluetooth and WSAQUERYSET for Set Service

WSAEnumNameSpaceProviders

WSAEnumNameSpaceProvidersEx

WSALookupServiceBegin

WSALookupServiceNext

WSAQUERYSET2

WSASetService

WSARecv function (winsock2.h)

10/22/2022 • 16 minutes to read • Edit Online

The **WSARecv** function receives data from a connected socket or a bound connectionless socket.

Syntax

```
int WSAAPI WSARecv(
  [in]      SOCKET                             s,
  [in, out] LPWSABUF                           lpBuffers,
  [in]      DWORD                              dwBufferCount,
  [out]     LPDWORD                            lpNumberOfBytesRecvd,
  [in, out] LPDWORD                            lpFlags,
  [in]      LPWSAOVERLAPPED                    lpOverlapped,
  [in]      LPWSAOVERLAPPED_COMPLETION_ROUTINE lpCompletionRoutine
);
```

Parameters

`[in] s`

A descriptor identifying a connected socket.

`[in, out] lpBuffers`

A pointer to an array of WSABUF structures. Each **WSABUF** structure contains a pointer to a buffer and the length, in bytes, of the buffer.

`[in] dwBufferCount`

The number of WSABUF structures in the *lpBuffers* array.

`[out] lpNumberOfBytesRecvd`

A pointer to the number, in bytes, of data received by this call if the receive operation completes immediately.

Use **NULL** for this parameter if the *lpOverlapped* parameter is not **NULL** to avoid potentially erroneous results. This parameter can be **NULL** only if the *lpOverlapped* parameter is not **NULL**.

`[in, out] lpFlags`

A pointer to flags used to modify the behavior of the **WSARecv** function call. For more information, see the Remarks section.

`[in] lpOverlapped`

A pointer to a WSAOVERLAPPED structure (ignored for nonoverlapped sockets).

`[in] lpCompletionRoutine`

Type: _In_opt_ **LPWSAOVERLAPPED_COMPLETION_ROUTINE**

A pointer to the completion routine called when the receive operation has been completed (ignored for nonoverlapped sockets).

Return value

If no error occurs and the receive operation has completed immediately, **WSARecv** returns zero. In this case, the completion routine will have already been scheduled to be called once the calling thread is in the alertable state. Otherwise, a value of **SOCKET_ERROR** is returned, and a specific error code can be retrieved by calling WSAGetLastError. The error code WSA_IO_PENDING indicates that the overlapped operation has been successfully initiated and that completion will be indicated at a later time. Any other error code indicates that the overlapped operation was not successfully initiated and no completion indication will occur.

ERROR CODE	MEANING
WSAECONNABORTED	The virtual circuit was terminated due to a time-out or other failure.
WSAECONNRESET	For a stream socket, the virtual circuit was reset by the remote side. The application should close the socket as it is no longer usable. For a UDP datagram socket, this error would indicate that a previous send operation resulted in an ICMP "Port Unreachable" message.
WSAEDISCON	Socket *s* is message oriented and the virtual circuit was gracefully closed by the remote side.
WSAEFAULT	The *lpBuffers* parameter is not completely contained in a valid part of the user address space.
WSAEINPROGRESS	A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function.
WSAEINTR	The (blocking) call was canceled by the WSACancelBlockingCall function.
WSAEINVAL	The socket has not been bound (for example, with bind).
WSAEMSGSIZE	The message was too large to fit into the specified buffer and (for unreliable protocols only) any trailing portion of the message that did not fit into the buffer has been discarded.
WSAENETDOWN	The network subsystem has failed.
WSAENETRESET	For a connection-oriented socket, this error indicates that the connection has been broken due to *keep-alive* activity that detected a failure while the operation was in progress. For a datagram socket, this error indicates that the time to live has expired.
WSAENOTCONN	The socket is not connected.
WSAENOTSOCK	The descriptor is not a socket.

WSAEOPNOTSUPP	**MSG_OOB** was specified, but the socket is not stream-style such as type SOCK_STREAM, OOB data is not supported in the communication domain associated with this socket, or the socket is unidirectional and supports only send operations.
WSAESHUTDOWN	The socket has been shut down; it is not possible to call WSARecv on a socket after shutdown has been invoked with *how* set to **SD_RECEIVE** or **SD_BOTH**.
WSAETIMEDOUT	The connection has been dropped because of a network failure or because the peer system failed to respond.
WSAEWOULDBLOCK	**Windows NT:** Overlapped sockets: there are too many outstanding overlapped I/O requests. Nonoverlapped sockets: The socket is marked as nonblocking and the receive operation cannot be completed immediately.
WSANOTINITIALISED	A successful WSAStartup call must occur before using this function.
WSA_IO_PENDING	An overlapped operation was successfully initiated and completion will be indicated at a later time.
WSA_OPERATION_ABORTED	The overlapped operation has been canceled due to the closure of the socket.

Remarks

The **WSARecv** function provides some additional features compared with the standard recv function in three important areas:

- It can be used in conjunction with overlapped sockets to perform overlapped recv operations.
- It allows multiple receive buffers to be specified making it applicable to the scatter/gather type of I/O.
- The *lpFlags* parameter is used both on input and returned on output, allowing applications to sense the output state of the **MSG_PARTIAL** flag bit. However, the **MSG_PARTIAL** flag bit is not supported by all protocols.

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

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

For overlapped sockets, **WSARecv** is used to post one or more buffers into which incoming data will be placed as it becomes available, after which the application-specified completion indication (invocation of the completion routine or setting of an event object) occurs. If the operation does not complete immediately, the final completion status is retrieved through the completion routine or WSAGetOverlappedResult.

> **Note** All I/O initiated by a given thread is canceled when that thread exits. For overlapped sockets, pending asynchronous operations can fail if the thread is closed before the operations complete. See ExitThread for more information.

If both *lpOverlapped* and *lpCompletionRoutine* are **NULL**, the socket in this function will be treated as a nonoverlapped socket.

For nonoverlapped sockets, the blocking semantics are identical to that of the standard recv function and the *lpOverlapped* and *lpCompletionRoutine* parameters are ignored. Any data that has already been received and buffered by the transport will be copied into the specified user buffers. In the case of a blocking socket with no data currently having been received and buffered by the transport, the call will block until data is received. Windows Sockets 2 does not define any standard blocking time-out mechanism for this function. For protocols acting as byte-stream protocols the stack tries to return as much data as possible subject to the available buffer space and amount of received data available. However, receipt of a single byte is sufficient to unblock the caller. There is no guarantee that more than a single byte will be returned. For protocols acting as message-oriented, a full message is required to unblock the caller.

> **Note** The socket options **SO_RCVTIMEO** and **SO_SNDTIMEO** apply only to blocking sockets.

Whether or not a protocol is acting as byte stream is determined by the setting of XP1_MESSAGE_ORIENTED and XP1_PSEUDO_STREAM in its WSAPROTOCOL_INFO structure and the setting of the MSG_PARTIAL flag passed in to this function (for protocols that support it). The following table lists relevant combinations, (an asterisk (*) indicates that the setting of this bit does not matter in this case).

XP1_MESSAGE_ORIENTED	XP1_PSEUDO_STREAM	MSG_PARTIAL	ACTS AS
not set	*	*	Byte stream
*	Set	*	Byte stream
set	Not set	set	Byte stream
set	Not set	not set	Message oriented

The buffers are filled in the order in which they appear in the array pointed to by *lpBuffers*, and the buffers are packed so that no holes are created.

If this function is completed in an overlapped manner, it is the Winsock service provider's responsibility to capture the WSABUF structures before returning from this call. This enables applications to build stack-based **WSABUF** arrays pointed to by the *lpBuffers* parameter.

For byte stream-style sockets (for example, type **SOCK_STREAM**), incoming data is placed into the buffers until the buffers are filled, the connection is closed, or the internally buffered data is exhausted. Regardless of whether or not the incoming data fills all the buffers, the completion indication occurs for overlapped sockets.

For message-oriented sockets (for example, type **SOCK_DGRAM**), an incoming message is placed into the buffers up to the total size of the buffers, and the completion indication occurs for overlapped sockets. If the message is larger than the buffers, the buffers are filled with the first part of the message. If the **MSG_PARTIAL** feature is supported by the underlying service provider, the **MSG_PARTIAL** flag is set in *lpFlags* and subsequent receive operations will retrieve the rest of the message. If **MSG_PARTIAL** is not supported but the protocol is reliable, **WSARecv** generates the error WSAEMSGSIZE and a subsequent receive operation with a larger buffer

can be used to retrieve the entire message. Otherwise, (that is, the protocol is unreliable and does not support **MSG_PARTIAL**), the excess data is lost, and **WSARecv** generates the error WSAEMSGSIZE.

For connection-oriented sockets, **WSARecv** can indicate the graceful termination of the virtual circuit in one of two ways that depend on whether the socket is byte stream or message oriented. For byte streams, zero bytes having been read (as indicated by a zero return value to indicate success, and *lpNumberOfBytesRecvd* value of zero) indicates graceful closure and that no more bytes will ever be read. For message-oriented sockets, where a zero byte message is often allowable, a failure with an error code of WSAEDISCON is used to indicate graceful closure. In any case a return error code of WSAECONNRESET indicates an abortive close has occurred.

The *lpFlags* parameter can be used to influence the behavior of the function invocation beyond the options specified for the associated socket. That is, the semantics of this function are determined by the socket options and the *lpFlags* parameter. The latter is constructed by using the bitwise OR operator with any of the values listed in the following table.

VALUE	MEANING
MSG_PEEK	Peeks at the incoming data. The data is copied into the buffer, but is not removed from the input queue. This flag is valid only for nonoverlapped sockets.
MSG_OOB	Processes OOB data.
MSG_PARTIAL	This flag is for message-oriented sockets only. On output, this flag indicates that the data specified is a portion of the message transmitted by the sender. Remaining portions of the message will be specified in subsequent receive operations. A subsequent receive operation with the **MSG_PARTIAL** flag cleared indicates end of sender's message. As an input parameter, this flag indicates that the receive operation should complete even if only part of a message has been received by the transport provider.

MSG_PUSH_IMMEDIATE	This flag is for stream-oriented sockets only. This flag allows an application that uses stream sockets to tell the transport provider not to delay completion of partially filled pending receive requests. This is a hint to the transport provider that the application is willing to receive any incoming data as soon as possible without necessarily waiting for the remainder of the data that might still be in transit. What constitutes a partially filled pending receive request is a transport-specific matter. In the case of TCP, this refers to the case of incoming TCP segments being placed into the receive request data buffer where none of the TCP segments indicated a PUSH bit value of 1. In this case, TCP may hold the partially filled receive request a little longer to allow the remainder of the data to arrive with a TCP segment that has the PUSH bit set to 1. This flag tells TCP not to hold the receive request but to complete it immediately. Using this flag for large block transfers is not recommended since processing partial blocks is often not optimal. This flag is useful only for cases where receiving and processing the partial data immediately helps decrease processing latency. This flag is a hint rather than an actual guarantee. This flag is supported on Windows 8.1, Windows Server 2012 R2, and later.
MSG_WAITALL	The receive request will complete only when one of the following events occurs: • The buffer supplied by the caller is completely full. • The connection has been closed. • The request has been canceled or an error occurred. Be aware that if the underlying transport provider does not support **MSG_WAITALL**, or if the socket is in a non-blocking mode, then this call will fail with **WSAEOPNOTSUPP**. Also, if **MSG_WAITALL** is specified along with **MSG_OOB**, **MSG_PEEK**, or **MSG_PARTIAL**, then this call will fail with **WSAEOPNOTSUPP**. This flag is not supported on datagram sockets or message-oriented sockets.

For message-oriented sockets, the **MSG_PARTIAL** bit is set in the *lpFlags* parameter if a partial message is received. If a complete message is received, **MSG_PARTIAL** is cleared in *lpFlags*. In the case of delayed completion, the value pointed to by *lpFlags* is not updated. When completion has been indicated, the application should call WSAGetOverlappedResult and examine the flags indicated by the *lpdwFlags* parameter.

> **Note** When issuing a blocking Winsock call such as **WSARecv** with the *lpOverlapped* parameter set to NULL, Winsock may need to wait for a network event before the call can complete. Winsock performs an alertable wait in this situation, which can be interrupted by an asynchronous procedure call (APC) scheduled on the same thread. Issuing another blocking Winsock call inside an APC that interrupted an ongoing blocking Winsock call on the same thread will lead to undefined behavior, and must never be attempted by Winsock clients.

Overlapped Socket I/O

If an overlapped operation completes immediately, **WSARecv** returns a value of zero and the *lpNumberOfBytesRecvd* parameter is updated with the number of bytes received and the flag bits indicated by the *lpFlags* parameter are also updated. If the overlapped operation is successfully initiated and will complete later, **WSARecv** returns **SOCKET_ERROR** and indicates error code WSA_IO_PENDING. In this case, *lpNumberOfBytesRecvd* and *lpFlags* are not updated. When the overlapped operation completes, the amount of data transferred is indicated either through the *cbTransferred* parameter in the completion routine (if specified), or through the *lpcbTransfer* parameter in WSAGetOverlappedResult. Flag values are obtained by examining the *lpdwFlags* parameter of **WSAGetOverlappedResult**.

The **WSARecv** function using overlapped I/O can be called from within the completion routine of a previous **WSARecv**, WSARecvFrom, WSASend or WSASendTo function. For a given socket, I/O completion routines will not be nested. This permits time-sensitive data transmissions to occur entirely within a preemptive context.

The *lpOverlapped* parameter must be valid for the duration of the overlapped operation. If multiple I/O operations are simultaneously outstanding, each must reference a separate WSAOVERLAPPED structure.

If the *lpCompletionRoutine* parameter is **NULL**, the *hEvent* parameter of *lpOverlapped* is signaled when the overlapped operation completes if it contains a valid event object handle. An application can use WSAWaitForMultipleEvents or WSAGetOverlappedResult to wait or poll on the event object.

If *lpCompletionRoutine* is not **NULL**, the *hEvent* parameter is ignored and can be used by the application to pass context information to the completion routine. A caller that passes a non-**NULL** *lpCompletionRoutine* and later calls WSAGetOverlappedResult for the same overlapped I/O request may not set the *fWait* parameter for that invocation of **WSAGetOverlappedResult** to **TRUE**. In this case the usage of the *hEvent* parameter is undefined, and attempting to wait on the *hEvent* parameter would produce unpredictable results.

The completion routine follows the same rules as stipulated for Windows file I/O completion routines. The completion routine will not be invoked until the thread is in an alertable wait state such as can occur when the function WSAWaitForMultipleEvents with the *fAlertable* parameter set to **TRUE** is invoked.

The prototype of the completion routine is as follows:

```
void CALLBACK CompletionROUTINE(
  IN DWORD dwError,
  IN DWORD cbTransferred,
  IN LPWSAOVERLAPPED lpOverlapped,
  IN DWORD dwFlags
);
```

CompletionRoutine is a placeholder for an application-defined or library-defined function name. The *dwError* specifies the completion status for the overlapped operation as indicated by *lpOverlapped*. The *cbTransferred* parameter specifies the number of bytes received. The *dwFlags* parameter contains information that would have appeared in *lpFlags* if the receive operation had completed immediately. This function does not return a value.

Returning from this function allows invocation of another pending completion routine for this socket. When using WSAWaitForMultipleEvents, all waiting completion routines are called before the alertable thread's wait is satisfied with a return code of **WSA_IO_COMPLETION**. The completion routines can be called in any order, not necessarily in the same order the overlapped operations are completed. However, the posted buffers are guaranteed to be filled in the same order in which they are specified.

If you are using I/O completion ports, be aware that the order of calls made to **WSARecv** is also the order in which the buffers are populated. **WSARecv** should not be called on the same socket simultaneously from different threads, because it can result in an unpredictable buffer order.

Example Code

The following example shows how to use the **WSARecv** function in overlapped I/O mode.

```
#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>

// Need to link with Ws2_32.lib
#pragma comment(lib, "ws2_32.lib")

#pragma warning(disable: 4127)  // Conditional expression is a constant

#define DATA_BUFSIZE 4096

int __cdecl main(int argc, char **argv)
{
    WSADATA wsd;
    struct addrinfo *result = NULL, *ptr = NULL, hints;
    WSAOVERLAPPED RecvOverlapped;
    SOCKET ConnSocket = INVALID_SOCKET;
    WSABUF DataBuf;
    DWORD RecvBytes, Flags;
    char buffer[DATA_BUFSIZE];

    int err = 0;
    int rc;

    if (argc != 2) {
        wprintf(L"usage: %s server-name\n", argv[0]);
        return 1;
    }
    // Load Winsock
    rc = WSAStartup(MAKEWORD(2, 2), &wsd);
    if (rc != 0) {
        wprintf(L"Unable to load Winsock: %d\n", rc);
        return 1;
    }
    // Make sure the hints struct is zeroed out
    SecureZeroMemory((PVOID) & hints, sizeof (struct addrinfo));

    // Initialize the hints to retrieve the server address for IPv4
    hints.ai_family = AF_INET;
    hints.ai_socktype = SOCK_STREAM;
    hints.ai_protocol = IPPROTO_TCP;

    rc = getaddrinfo(argv[1], "27015", &hints, &result);
    if (rc != 0) {
        wprintf(L"getaddrinfo failed with error: %d\n", rc);
        return 1;
    }

    for (ptr = result; ptr != NULL; ptr = ptr->ai_next) {

        ConnSocket = socket(ptr->ai_family, ptr->ai_socktype, ptr->ai_protocol);
        if (ConnSocket == INVALID_SOCKET) {
            wprintf(L"socket failed with error: %d\n", WSAGetLastError());
            freeaddrinfo(result);
```

```
            freeaddrinfo(result);
            return 1;
        }

        rc = connect(ConnSocket, ptr->ai_addr, (int) ptr->ai_addrlen);
        if (rc == SOCKET_ERROR) {

            if (WSAECONNREFUSED == (err = WSAGetLastError())) {
                closesocket(ConnSocket);
                ConnSocket = INVALID_SOCKET;
                continue;
            }
            wprintf(L"connect failed with error: %d\n", err);
            freeaddrinfo(result);
            closesocket(ConnSocket);
            return 1;
        }
        break;
    }
    if (ConnSocket == INVALID_SOCKET) {
        wprintf(L"Unable to establish connection with the server!\n");
        freeaddrinfo(result);
        return 1;
    }

    wprintf(L"Client connected...\n");

    // Make sure the RecvOverlapped struct is zeroed out
    SecureZeroMemory((PVOID) & RecvOverlapped, sizeof (WSAOVERLAPPED));

    // Create an event handle and setup an overlapped structure.
    RecvOverlapped.hEvent = WSACreateEvent();
    if (RecvOverlapped.hEvent == NULL) {
        wprintf(L"WSACreateEvent failed: %d\n", WSAGetLastError());
        freeaddrinfo(result);
        closesocket(ConnSocket);
        return 1;
    }

    DataBuf.len = DATA_BUFSIZE;
    DataBuf.buf = buffer;

    // Call WSARecv until the peer closes the connection
    // or until an error occurs
    while (1) {

        Flags = 0;
        rc = WSARecv(ConnSocket, &DataBuf, 1, &RecvBytes, &Flags, &RecvOverlapped, NULL);
        if ((rc == SOCKET_ERROR) && (WSA_IO_PENDING != (err = WSAGetLastError()))) {
            wprintf(L"WSARecv failed with error: %d\n", err);
            break;
        }

        rc = WSAWaitForMultipleEvents(1, &RecvOverlapped.hEvent, TRUE, INFINITE, TRUE);
        if (rc == WSA_WAIT_FAILED) {
            wprintf(L"WSAWaitForMultipleEvents failed with error: %d\n", WSAGetLastError());
            break;
        }

        rc = WSAGetOverlappedResult(ConnSocket, &RecvOverlapped, &RecvBytes, FALSE, &Flags);
        if (rc == FALSE) {
            wprintf(L"WSARecv operation failed with error: %d\n", WSAGetLastError());
            break;
        }

        wprintf(L"Read %d bytes\n", RecvBytes);

        WSAResetEvent(RecvOverlapped.hEvent);
```

```
        // If 0 bytes are received, the connection was closed
        if (RecvBytes == 0)
            break;
    }

    WSACloseEvent(RecvOverlapped.hEvent);
    closesocket(ConnSocket);
    freeaddrinfo(result);

    WSACleanup();

    return 0;
}
```

Windows Phone 8: This function is supported for Windows Phone Store apps on Windows Phone 8 and later.

Windows 8.1 and **Windows Server 2012 R2**: This function is supported for Windows Store apps on Windows 8.1, Windows Server 2012 R2, and later.

Requirements

Minimum supported client	Windows 8.1, Windows Vista [desktop apps | UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps | UWP apps]
Target Platform	Windows
Header	winsock2.h
Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

WSABUF

WSACloseEvent

WSACreateEvent

WSAGetOverlappedResult

WSAOVERLAPPED

WSASocket

WSAWaitForMultipleEvents

Winsock Functions

Winsock Reference

recv

WSARecvDisconnect function (winsock2.h)

10/22/2022 • 2 minutes to read • Edit Online

The **WSARecvDisconnect** function terminates reception on a socket, and retrieves the disconnect data if the socket is connection oriented.

Syntax

```
int WSAAPI WSARecvDisconnect(
  [in]  SOCKET   s,
  [out] LPWSABUF lpInboundDisconnectData
);
```

Parameters

`[in] s`

A descriptor identifying a socket.

`[out] lpInboundDisconnectData`

A pointer to the incoming disconnect data.

Return value

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

ERROR CODE	MEANING
WSANOTINITIALISED	A successful WSAStartup call must occur before using this function.
WSAENETDOWN	The network subsystem has failed.
WSAEFAULT	The buffer referenced by the parameter *lpInboundDisconnectData* is too small.
WSAENOPROTOOPT	The disconnect data is not supported by the indicated protocol family. Note that implementations of TCP/IP that do not support disconnect data are not required to return the WSAENOPROTOOPT error code. See the remarks section for information about the Microsoft implementation of TCP/IP.
WSAEINPROGRESS	A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function.
WSAENOTCONN	The socket is not connected (connection-oriented sockets only).

WSAENOTSOCK	The descriptor is not a socket.

Remarks

The **WSARecvDisconnect** function is used on connection-oriented sockets to disable reception and retrieve any incoming disconnect data from the remote party. This is equivalent to a shutdown (SD_RECEIVE), except that **WSARecvDisconnect** also allows receipt of disconnect data (in protocols that support it).

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

> **Note** The native implementation of TCP/IP on Windows does not support disconnect data. Disconnect data is only supported with Windows Sockets providers that have the XP1_DISCONNECT_DATA flag in their WSAPROTOCOL_INFO structure. Use the WSAEnumProtocols function to obtain **WSAPROTOCOL_INFO** structures for all installed providers.

To successfully receive incoming disconnect data, an application must use other mechanisms to determine that the circuit has been closed. For example, an application needs to receive an FD_CLOSE notification, to receive a zero return value, or to receive a WSAEDISCON or WSAECONNRESET error code from recv/WSARecv.

The **WSARecvDisconnect** function does not close the socket, and resources attached to the socket will not be freed until closesocket is invoked.

The **WSARecvDisconnect** function does not block regardless of the SO_LINGER setting on the socket.

An application should not rely on being able to reuse a socket after it has been disconnected using **WSARecvDisconnect**. In particular, a Windows Sockets provider is not required to support the use of connect or WSAConnect on such a socket.

> **Note** When issuing a blocking Winsock call such as **WSARecvDisconnect**, Winsock may need to wait for a network event before the call can complete. Winsock performs an alertable wait in this situation, which can be interrupted by an asynchronous procedure call (APC) scheduled on the same thread. Issuing another blocking Winsock call inside an APC that interrupted an ongoing blocking Winsock call on the same thread will lead to undefined behavior, and must never be attempted by Winsock clients.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winsock2.h

Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

WSAConnect

WSAEnumProtocols

WSAGetLastError

WSAPROTOCOL_INFO

WSARecv

Winsock Functions

Winsock Reference

closesocket

connect

socket

WSARecvFrom function (winsock2.h)

10/22/2022 • 13 minutes to read • Edit Online

The **WSARecvFrom** function receives a datagram and stores the source address.

Syntax

```
int WSAAPI WSARecvFrom(
  [in]      SOCKET                            s,
  [in, out] LPWSABUF                          lpBuffers,
  [in]      DWORD                             dwBufferCount,
  [out]     LPDWORD                           lpNumberOfBytesRecvd,
  [in, out] LPDWORD                           lpFlags,
  [out]     sockaddr                          *lpFrom,
  [in, out] LPINT                             lpFromlen,
  [in]      LPWSAOVERLAPPED                   lpOverlapped,
  [in]      LPWSAOVERLAPPED_COMPLETION_ROUTINE lpCompletionRoutine
);
```

Parameters

`[in] s`

A descriptor identifying a socket.

`[in, out] lpBuffers`

A pointer to an array of WSABUF structures. Each **WSABUF** structure contains a pointer to a buffer and the length of the buffer.

`[in] dwBufferCount`

The number of WSABUF structures in the *lpBuffers* array.

`[out] lpNumberOfBytesRecvd`

A pointer to the number of bytes received by this call if the **WSARecvFrom** operation completes immediately.

Use **NULL** for this parameter if the *lpOverlapped* parameter is not **NULL** to avoid potentially erroneous results. This parameter can be **NULL** only if the *lpOverlapped* parameter is not **NULL**.

`[in, out] lpFlags`

A pointer to flags used to modify the behavior of the **WSARecvFrom** function call. See remarks below.

`[out] lpFrom`

An optional pointer to a buffer that will hold the source address upon the completion of the overlapped operation.

`[in, out] lpFromlen`

A pointer to the size, in bytes, of the "from" buffer required only if *lpFrom* is specified.

`[in] lpOverlapped`

A pointer to a WSAOVERLAPPED structure (ignored for nonoverlapped sockets).

`[in] lpCompletionRoutine`

Type: _In_opt_ LPWSAOVERLAPPED_COMPLETION_ROUTINE

A pointer to the completion routine called when the **WSARecvFrom** operation has been completed (ignored for nonoverlapped sockets).

Return value

If no error occurs and the receive operation has completed immediately, **WSARecvFrom** returns zero. In this case, the completion routine will have already been scheduled to be called once the calling thread is in the alertable state. Otherwise, a value of **SOCKET_ERROR** is returned, and a specific error code can be retrieved by calling WSAGetLastError. The error code **WSA_IO_PENDING** indicates that the overlapped operation has been successfully initiated and that completion will be indicated at a later time. Any other error code indicates that the overlapped operation was not successfully initiated and no completion indication will occur.

ERROR CODE	MEANING
WSAECONNRESET	The virtual circuit was reset by the remote side executing a hard or abortive close. The application should close the socket as it is no longer usable. For a UPD datagram socket, this error would indicate that a previous send operation resulted in an ICMP "Port Unreachable" message.
WSAEFAULT	The *lpBuffers*, *lpFlags*, *lpFrom*, *lpNumberOfBytesRecvd*, *lpFromlen*, *lpOverlapped*, or *lpCompletionRoutine* parameter is not totally contained in a valid part of the user address space: the *lpFrom* buffer was too small to accommodate the peer address.
WSAEINPROGRESS	A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function.
WSAEINTR	A blocking Windows Socket 1.1 call was canceled through WSACancelBlockingCall.
WSAEINVAL	The socket has not been bound (with bind, for example).
WSAEMSGSIZE	The message was too large for the specified buffer and (for unreliable protocols only) any trailing portion of the message that did not fit into the buffer has been discarded.
WSAENETDOWN	The network subsystem has failed.
WSAENETRESET	For a datagram socket, this error indicates that the time to live has expired.
WSAENOTCONN	The socket is not connected (connection-oriented sockets only).

WSAEWOULDBLOCK	**Windows NT:** Overlapped sockets: There are too many outstanding overlapped I/O requests. Nonoverlapped sockets: The socket is marked as nonblocking and the receive operation cannot be completed immediately.
WSANOTINITIALISED	A successful WSAStartup call must occur before using this function.
WSA_IO_PENDING	An overlapped operation was successfully initiated and completion will be indicated later.
WSA_OPERATION_ABORTED	The overlapped operation has been canceled due to the closure of the socket.

Remarks

The **WSARecvFrom** function provides functionality over and above the standard recvfrom function in three important areas:

- It can be used in conjunction with overlapped sockets to perform overlapped receive operations.
- It allows multiple receive buffers to be specified making it applicable to the scatter/gather type of I/O.
- The *lpFlags* parameter is both an input and an output parameter, allowing applications to sense the output state of the **MSG_PARTIAL** flag bit. Be aware that the **MSG_PARTIAL** flag bit is not supported by all protocols.

The **WSARecvFrom** function is used primarily on a connectionless socket specified by *s*. The socket's local address must be known. For server applications, this is usually done explicitly through bind. Explicit binding is discouraged for client applications. For client applications using this function the socket can become bound implicitly to a local address through sendto, WSASendTo, or WSAJoinLeaf.

For overlapped sockets, this function is used to post one or more buffers into which incoming data will be placed as it becomes available on a (possibly connected) socket, after which the application-specified completion indication (invocation of the completion routine or setting of an event object) occurs. If the operation does not complete immediately, the final completion status is retrieved through the completion routine or WSAGetOverlappedResult. Also, the values indicated by *lpFrom* and *lpFromlen* are not updated until completion is itself indicated. Applications must not use or disturb these values until they have been updated; therefore the application must not use automatic (that is, stack-based) variables for these parameters.

> **Note** All I/O initiated by a given thread is canceled when that thread exits. For overlapped sockets, pending asynchronous operations can fail if the thread is closed before the operations complete. See ExitThread for more information.

If both *lpOverlapped* and *lpCompletionRoutine* are **NULL**, the socket in this function will be treated as a nonoverlapped socket.

For nonoverlapped sockets, the blocking semantics are identical to that of the standard WSARecv function and the *lpOverlapped* and *lpCompletionRoutine* parameters are ignored. Any data that has already been received and buffered by the transport will be copied into the user buffers. For the case of a blocking socket with no data currently having been received and buffered by the transport, the call will block until data is received.

The buffers are filled in the order in which they appear in the array indicated by *lpBuffers*, and the buffers are packed so that no holes are created.

If this function is completed in an overlapped manner, it is the Winsock service provider's responsibility to capture the WSABUF structures before returning from this call. This enables applications to build stack-based **WSABUF** arrays pointed to by the *lpBuffers* parameter.

For connectionless socket types, the address from which the data originated is copied to the buffer indicated by *lpFrom*. The value pointed to by *lpFromlen* is initialized to the size of this buffer, and is modified on completion to indicate the actual size of the address stored there. As stated previously for overlapped sockets, the *lpFrom* and *lpFromlen* parameters are not updated until after the overlapped I/O has completed. The memory pointed to by these parameters must, therefore, remain available to the service provider and cannot be allocated on the application stack frame. The *lpFrom* and *lpFromlen* parameters are ignored for connection-oriented sockets.

For byte stream–style sockets (for example, type SOCK_STREAM), incoming data is placed into the buffers until:

- The buffers are filled.
- The connection is closed.
- The internally buffered data is exhausted.

Regardless of whether or not the incoming data fills all the buffers, the completion indication occurs for overlapped sockets. For message-oriented sockets, an incoming message is placed into the buffers up to the total size of the buffers, and the completion indication occurs for overlapped sockets. If the message is larger than the buffers, the buffers are filled with the first part of the message. If the **MSG_PARTIAL** feature is supported by the underlying service provider, the **MSG_PARTIAL** flag is set in *lpFlags* and subsequent receive operation(s) will retrieve the rest of the message. If **MSG_PARTIAL** is not supported, but the protocol is reliable, **WSARecvFrom** generates the error WSAEMSGSIZE and a subsequent receive operation with a larger buffer can be used to retrieve the entire message. Otherwise, (that is, the protocol is unreliable and does not support **MSG_PARTIAL**), the excess data is lost, and **WSARecvFrom** generates the error WSAEMSGSIZE.

The *lpFlags* parameter can be used to influence the behavior of the function invocation beyond the options specified for the associated socket. That is, the semantics of this function are determined by the socket options and the *lpFlags* parameter. The latter is constructed by using the bitwise OR operator with any of any of the values listed in the following table.

VALUE	MEANING
MSG_PEEK	Previews the incoming data. The data is copied into the buffer, but is not removed from the input queue. This flag is valid only for nonoverlapped sockets.
MSG_OOB	Processes OOB data.
MSG_PARTIAL	This flag is for message-oriented sockets only. On output, this flag indicates that the data is a portion of the message transmitted by the sender. Remaining portions of the message will be transmitted in subsequent receive operations. A subsequent receive operation with **MSG_PARTIAL** flag cleared indicates the end of the sender's message. As an input parameter, this flag indicates that the receive operation should complete even if only part of a message has been received by the service provider.

For message-oriented sockets, the **MSG_PARTIAL** bit is set in the *lpFlags* parameter if a partial message is received. If a complete message is received, **MSG_PARTIAL** is cleared in *lpFlags*. In the case of delayed completion, the value pointed to by *lpFlags* is not updated. When completion has been indicated the application should call WSAGetOverlappedResult and examine the flags pointed to by the *lpdwFlags* parameter.

> **Note** When issuing a blocking Winsock call such as **WSARecvFrom** with the *lpOverlapped* parameter set to NULL, Winsock may need to wait for a network event before the call can complete. Winsock performs an alertable wait in this situation, which can be interrupted by an asynchronous procedure call (APC) scheduled on the same thread. Issuing another blocking Winsock call inside an APC that interrupted an ongoing blocking Winsock call on the same thread will lead to undefined behavior, and must never be attempted by Winsock clients.

Overlapped Socket I/O

If an overlapped operation completes immediately, **WSARecvFrom** returns a value of zero and the *lpNumberOfBytesRecvd* parameter is updated with the number of bytes received and the flag bits pointed by the *lpFlags* parameter are also updated. If the overlapped operation is successfully initiated and will complete later, **WSARecvFrom** returns **SOCKET_ERROR** and indicates error code **WSA_IO_PENDING**. In this case, *lpNumberOfBytesRecvd* and *lpFlags* is not updated. When the overlapped operation completes the amount of data transferred is indicated either through the *cbTransferred* parameter in the completion routine (if specified), or through the *lpcbTransfer* parameter in WSAGetOverlappedResult. Flag values are obtained either through the *dwFlags* parameter of the completion routine, or by examining the *lpdwFlags* parameter of **WSAGetOverlappedResult**.

The **WSARecvFrom** function can be called from within the completion routine of a previous WSARecv, **WSARecvFrom**, WSASend, or WSASendTo function. For a given socket, I/O completion routines will not be nested. This permits time-sensitive data transmissions to occur entirely within a preemptive context.

The *lpOverlapped* parameter must be valid for the duration of the overlapped operation. If multiple I/O operations are simultaneously outstanding, each must reference a separate WSAOVERLAPPED structure.

If the *lpCompletionRoutine* parameter is **NULL**, the *hEvent* parameter of *lpOverlapped* is signaled when the overlapped operation completes if it contains a valid event object handle. An application can use WSAWaitForMultipleEvents or WSAGetOverlappedResult to wait or poll on the event object.

If *lpCompletionRoutine* is not **NULL**, the *hEvent* parameter is ignored and can be used by the application to pass context information to the completion routine. A caller that passes a non-**NULL** *lpCompletionRoutine* and later calls WSAGetOverlappedResult for the same overlapped I/O request may not set the *fWait* parameter for that invocation of **WSAGetOverlappedResult** to **TRUE**. In this case the usage of the *hEvent* parameter is undefined, and attempting to wait on the *hEvent* parameter would produce unpredictable results.

The completion routine follows the same rules as stipulated for Windows file I/O completion routines. The completion routine will not be invoked until the thread is in an alertable wait state such as can occur when the function WSAWaitForMultipleEvents with the *fAlertable* parameter set to **TRUE** is invoked.

If an IO completion port is used and the *lpCompletionRoutine* parameter and the *hEvent* parameter are **NULL**, the result of the operation is schedule on the IO completion port. This happens for all successful operations, whether the operations complete immediately or not.

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

The prototype of the completion routine is as follows.

```
void CALLBACK CompletionROUTINE(
  IN DWORD dwError,
  IN DWORD cbTransferred,
  IN LPWSAOVERLAPPED lpOverlapped,
  IN DWORD dwFlags
);
```

The **CompletionRoutine** is a placeholder for an application-defined or library-defined function name. The *dwError* specifies the completion status for the overlapped operation as indicated by *lpOverlapped*. The *cbTransferred* specifies the number of bytes received. The *dwFlags* parameter contains information that would have appeared in *lpFlags* if the receive operation had completed immediately. This function does not return a value.

Returning from this function allows invocation of another pending completion routine for this socket. When using WSAWaitForMultipleEvents, all waiting completion routines are called before the alertable thread's wait is satisfied with a return code of WSA_IO_COMPLETION. The completion routines can be called in any order, not necessarily in the same order the overlapped operations are completed. However, the posted buffers are guaranteed to be filled in the same order they are specified.

Example Code

The following example demonstrates the use of the **WSARecvFrom** function.

```
#ifndef UNICODE
#define UNICODE
#endif

#define WIN32_LEAN_AND_MEAN

#include <winsock2.h>
#include <Ws2tcpip.h>
#include <stdio.h>

int __cdecl main()
{

    WSADATA wsaData;
    WSABUF DataBuf;
    WSAOVERLAPPED Overlapped;

    SOCKET RecvSocket = INVALID_SOCKET;
    struct sockaddr_in RecvAddr;
    struct sockaddr_in SenderAddr;

    int SenderAddrSize = sizeof (SenderAddr);
    u_short Port = 27015;

    char RecvBuf[1024];
    int BufLen = 1024;
    DWORD BytesRecv = 0;
    DWORD Flags = 0;

    int err = 0;
    int rc;
    int retval = 0;

    //-----------------------------------------------
    // Initialize Winsock
    rc = WSAStartup(MAKEWORD(2, 2), &wsaData);
    if (rc != 0) {
        /* Could not find a usable Winsock DLL */
        wprintf(L"WSAStartup failed with error: %ld\n", rc);
```

```
        return 1;
    }

    // Make sure the Overlapped struct is zeroed out
    SecureZeroMemory((PVOID) &Overlapped, sizeof(WSAOVERLAPPED) );

    // Create an event handle and setup the overlapped structure.
    Overlapped.hEvent = WSACreateEvent();
    if (Overlapped.hEvent == NULL) {
        wprintf(L"WSACreateEvent failed with error: %d\n", WSAGetLastError());
        WSACleanup();
        return 1;
    }
    //-----------------------------------------------
    // Create a receiver socket to receive datagrams
    RecvSocket = WSASocket(AF_INET,
                           SOCK_DGRAM,
                           IPPROTO_UDP, NULL, 0, WSA_FLAG_OVERLAPPED);

    if (RecvSocket == INVALID_SOCKET) {
        /* Could not open a socket */
        wprintf(L"WSASocket failed with error: %ld\n", WSAGetLastError());
        WSACloseEvent(Overlapped.hEvent);
        WSACleanup();
        return 1;
    }
    //-----------------------------------------------
    // Bind the socket to any address and the specified port.
    RecvAddr.sin_family = AF_INET;
    RecvAddr.sin_port = htons(Port);
    RecvAddr.sin_addr.s_addr = htonl(INADDR_ANY);

    rc = bind(RecvSocket, (SOCKADDR *) & RecvAddr, sizeof (RecvAddr));
    if (rc != 0) {
        /* Bind to the socket failed */
        wprintf(L"bind failed with error: %ld\n", WSAGetLastError());
        WSACloseEvent(Overlapped.hEvent);
        closesocket(RecvSocket);
        WSACleanup();
        return 1;
    }

    //-----------------------------------------------
    // Call the recvfrom function to receive datagrams
    // on the bound socket.
    DataBuf.len = BufLen;
    DataBuf.buf = RecvBuf;
    wprintf(L"Listening for incoming datagrams on port=%d\n", Port);
    rc = WSARecvFrom(RecvSocket,
                     &DataBuf,
                     1,
                     &BytesRecv,
                     &Flags,
                     (SOCKADDR *) & SenderAddr,
                     &SenderAddrSize, &Overlapped, NULL);

    if (rc != 0) {
        err = WSAGetLastError();
        if (err != WSA_IO_PENDING) {
            wprintf(L"WSARecvFrom failed with error: %ld\n", err);
            WSACloseEvent(Overlapped.hEvent);
            closesocket(RecvSocket);
            WSACleanup();
            return 1;
        }
        else {
            rc = WSAWaitForMultipleEvents(1, &Overlapped.hEvent, TRUE, INFINITE, TRUE);
            if (rc == WSA_WAIT_FAILED) {
                wprintf(L"WSAWaitForMultipleEvents failed with error: %d\n", WSAGetLastError());
```

```
                retval = 1;
            }

            rc = WSAGetOverlappedResult(RecvSocket, &Overlapped, &BytesRecv,
                                FALSE, &Flags);
            if (rc == FALSE) {
                wprintf(L"WSArecvFrom failed with error: %d\n", WSAGetLastError());
                retval = 1;
            }
            else
                wprintf(L"Number of received bytes = %d\n", BytesRecv);

            wprintf(L"Finished receiving. Closing socket.\n");
        }

    }
    //-----------------------------------------------
    // When the application is finished receiving, close the socket.

    WSACloseEvent(Overlapped.hEvent);
    closesocket(RecvSocket);
    wprintf(L"Exiting.\n");

    //-----------------------------------------------
    // Clean up and quit.
    WSACleanup();
    return (retval);
}
```

Windows Phone 8: This function is supported for Windows Phone Store apps on Windows Phone 8 and later.

Windows 8.1 and **Windows Server 2012 R2**: This function is supported for Windows Store apps on Windows 8.1, Windows Server 2012 R2, and later.

Requirements

Minimum supported client	Windows 8.1, Windows Vista [desktop apps | UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps | UWP apps]
Target Platform	Windows
Header	winsock2.h
Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

WSABUF

WSACloseEvent

WSACreateEvent

WSAGetOverlappedResult

WSAOVERLAPPED

WSASend

WSASendTo

WSASocket

WSAWaitForMultipleEvents

Winsock Functions

Winsock Reference

sendto

WSARemoveServiceClass function (winsock2.h)

10/22/2022 • 2 minutes to read • Edit Online

The **WSARemoveServiceClass** function permanently removes the service class schema from the registry.

Syntax

```
INT WSAAPI WSARemoveServiceClass(
  [in] LPGUID lpServiceClassId
);
```

Parameters

`[in] lpServiceClassId`

Pointer to the GUID for the service class you want to remove.

Return value

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

ERROR CODE	MEANING
WSATYPE_NOT_FOUND	The specified class was not found.
WSAEACCES	The calling routine does not have sufficient privileges to remove the Service.
WSAETOOMANYREFS	There are service instances that still reference the class. Removal of this class is not possible at this time.
WSANOTINITIALISED	The WS2_32.DLL has not been initialized. The application must first call WSAStartup before calling any Windows Sockets functions.
WSAEINVAL	The specified GUID was not valid.
WSA_NOT_ENOUGH_MEMORY	There was insufficient memory to perform the operation.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winsock2.h
Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

WSAGetLastError

WSAStartup

Winsock Functions

Winsock Reference

WSAResetEvent function (winsock2.h)

10/22/2022 • 2 minutes to read • Edit Online

The **WSAResetEvent** function resets the state of the specified event object to nonsignaled.

Syntax

```
BOOL WSAAPI WSAResetEvent(
  [in] WSAEVENT hEvent
);
```

Parameters

`[in] hEvent`

A handle that identifies an open event object handle.

Return value

If the **WSAResetEvent** function succeeds, the return value is **TRUE**. If the function fails, the return value is **FALSE**. To get extended error information, call WSAGetLastError.

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

Remarks

The **WSAResetEvent** function is used to set the state of the event object to nonsignaled.

The proper way to reset the state of an event object used with the WSAEventSelect function is to pass the handle of the event object to the WSAEnumNetworkEvents function in the *hEventObject* parameter. This will reset the event object and adjust the status of active FD events on the socket in an atomic fashion.

Windows Phone 8: This function is supported for Windows Phone Store apps on Windows Phone 8 and later.

Windows 8.1 and **Windows Server 2012 R2**: This function is supported for Windows Store apps on Windows 8.1, Windows Server 2012 R2, and later.

Requirements

Minimum supported client	Windows 8.1, Windows Vista [desktop apps | UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps | UWP apps]
Target Platform	Windows
Header	winsock2.h
Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

WSACloseEvent

WSACreateEvent

WSAEnumNetworkEvents

WSAEventSelect

WSASetEvent

Winsock Functions

Winsock Reference

WSASend function (winsock2.h)

10/22/2022 • 12 minutes to read • Edit Online

The **WSASend** function sends data on a connected socket.

Syntax

```
int WSAAPI WSASend(
  [in]  SOCKET                             s,
  [in]  LPWSABUF                           lpBuffers,
  [in]  DWORD                              dwBufferCount,
  [out] LPDWORD                            lpNumberOfBytesSent,
  [in]  DWORD                              dwFlags,
  [in]  LPWSAOVERLAPPED                    lpOverlapped,
  [in]  LPWSAOVERLAPPED_COMPLETION_ROUTINE lpCompletionRoutine
);
```

Parameters

`[in] s`

A descriptor that identifies a connected socket.

`[in] lpBuffers`

A pointer to an array of WSABUF structures. Each **WSABUF** structure contains a pointer to a buffer and the length, in bytes, of the buffer. For a Winsock application, once the **WSASend** function is called, the system owns these buffers and the application may not access them. This array must remain valid for the duration of the send operation.

`[in] dwBufferCount`

The number of WSABUF structures in the *lpBuffers* array.

`[out] lpNumberOfBytesSent`

A pointer to the number, in bytes, sent by this call if the I/O operation completes immediately.

Use **NULL** for this parameter if the *lpOverlapped* parameter is not **NULL** to avoid potentially erroneous results. This parameter can be **NULL** only if the *lpOverlapped* parameter is not **NULL**.

`[in] dwFlags`

The flags used to modify the behavior of the **WSASend** function call. For more information, see Using *dwFlags* in the Remarks section.

`[in] lpOverlapped`

A pointer to a WSAOVERLAPPED structure. This parameter is ignored for nonoverlapped sockets.

`[in] lpCompletionRoutine`

Type: _In_opt_ **LPWSAOVERLAPPED_COMPLETION_ROUTINE**

A pointer to the completion routine called when the send operation has been completed. This parameter is ignored for nonoverlapped sockets.

Return value

If no error occurs and the send operation has completed immediately, **WSASend** returns zero. In this case, the completion routine will have already been scheduled to be called once the calling thread is in the alertable state. Otherwise, a value of **SOCKET_ERROR** is returned, and a specific error code can be retrieved by calling WSAGetLastError. The error code WSA_IO_PENDING indicates that the overlapped operation has been successfully initiated and that completion will be indicated at a later time. Any other error code indicates that the overlapped operation was not successfully initiated and no completion indication will occur.

ERROR CODE	MEANING
WSAECONNABORTED	The virtual circuit was terminated due to a time-out or other failure.
WSAECONNRESET	For a stream socket, the virtual circuit was reset by the remote side. The application should close the socket as it is no longer usable. For a UDP datagram socket, this error would indicate that a previous send operation resulted in an ICMP "Port Unreachable" message.
WSAEFAULT	The *lpBuffers*, *lpNumberOfBytesSent*, *lpOverlapped*, *lpCompletionRoutine* parameter is not totally contained in a valid part of the user address space.
WSAEINTR	A blocking Windows Socket 1.1 call was canceled through WSACancelBlockingCall.
WSAEINPROGRESS	A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function.
WSAEINVAL	The socket has not been bound with bind or the socket is not created with the overlapped flag.
WSAEMSGSIZE	The socket is message oriented, and the message is larger than the maximum supported by the underlying transport.
WSAENETDOWN	The network subsystem has failed.
WSAENETRESET	For a stream socket, the connection has been broken due to keep-alive activity detecting a failure while the operation was in progress. For a datagram socket, this error indicates that the time to live has expired.
WSAENOBUFS	The Windows Sockets provider reports a buffer deadlock.
WSAENOTCONN	The socket is not connected.
WSAENOTSOCK	The descriptor is not a socket.

WSAEOPNOTSUPP	**MSG_OOB** was specified, but the socket is not stream-style such as type **SOCK_STREAM**, OOB data is not supported in the communication domain associated with this socket, **MSG_PARTIAL** is not supported, or the socket is unidirectional and supports only receive operations.
WSAESHUTDOWN	The socket has been shut down; it is not possible to WSASend on a socket after shutdown has been invoked with how set to **SD_SEND** or **SD_BOTH**.
WSAEWOULDBLOCK	**Windows NT:** Overlapped sockets: There are too many outstanding overlapped I/O requests. Nonoverlapped sockets: The socket is marked as nonblocking and the send operation cannot be completed immediately.
WSANOTINITIALISED	A successful WSAStartup call must occur before using this function.
WSA_IO_PENDING	An overlapped operation was successfully initiated and completion will be indicated at a later time.
WSA_OPERATION_ABORTED	The overlapped operation has been canceled due to the closure of the socket, the execution of the "SIO_FLUSH" command in WSAIoctl, or the thread that initiated the overlapped request exited before the operation completed. For more information, see the Remarks section.

Remarks

The **WSASend** function provides functionality over and above the standard send function in two important areas:

- It can be used in conjunction with overlapped sockets to perform overlapped send operations.
- It allows multiple send buffers to be specified making it applicable to the scatter/gather type of I/O.

The **WSASend** function is used to write outgoing data from one or more buffers on a connection-oriented socket specified by *s*. It can also be used, however, on connectionless sockets that have a stipulated default peer address established through the connect or WSAConnect function.

A socket created by the socket function will have the overlapped attribute as the default. A socket created by the WSASocket function with the *dwFlags* parameter passed to **WSASocket** with the **WSA_FLAG_OVERLAPPED** bit set will have the overlapped attribute. For sockets with the overlapped attribute, **WSASend** uses overlapped I/O unless both the *lpOverlapped* and *lpCompletionRoutine* parameters are **NULL**. In that case, the socket is treated as a non-overlapped socket. A completion indication will occur, invoking the completion of a routine or setting of an event object, when the buffer(s) have been consumed by the transport. If the operation does not complete immediately, the final completion status is retrieved through the completion routine or WSAGetOverlappedResult.

If both *lpOverlapped* and *lpCompletionRoutine* are **NULL**, the socket in this function will be treated as a non-overlapped socket.

For non-overlapped sockets, the last two parameters (*lpOverlapped*, *lpCompletionRoutine*) are ignored and **WSASend** adopts the same blocking semantics as send. Data is copied from the buffer(s) into the transport's buffer. If the socket is non-blocking and stream-oriented, and there is not sufficient space in the transport's buffer, **WSASend** will return with only part of the application's buffers having been consumed. Given the same

buffer situation and a blocking socket, **WSASend** will block until all of the application buffer contents have been consumed.

> **Note** The socket options **SO_RCVTIMEO** and **SO_SNDTIMEO** apply only to blocking sockets.

If this function is completed in an overlapped manner, it is the Winsock service provider's responsibility to capture the WSABUF structures before returning from this call. This enables applications to build stack-based **WSABUF** arrays pointed to by the *lpBuffers* parameter.

For message-oriented sockets, do not exceed the maximum message size of the underlying provider, which can be obtained by getting the value of socket option **SO_MAX_MSG_SIZE**. If the data is too long to pass atomically through the underlying protocol the error WSAEMSGSIZE is returned, and no data is transmitted.

Windows Me/98/95: The **WSASend** function does not support more than 16 buffers.

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

Using dwFlags

The *dwFlags* parameter can be used to influence the behavior of the function invocation beyond the options specified for the associated socket. That is, the semantics of this function are determined by the socket options and the *dwFlags* parameter. The latter is constructed by using the bitwise OR operator with any of any of the values listed in the following table.

VALUE	MEANING
MSG_DONTROUTE	Specifies that the data should not be subject to routing. A Windows Sockets service provider can choose to ignore this flag.
MSG_OOB	Send OOB data on a stream-style socket such as **SOCK_STREAM** only.
MSG_PARTIAL	Specifies that *lpBuffers* only contains a partial message. Be aware that the error code WSAEOPNOTSUPP will be returned by transports that do not support partial message transmissions.

> **Note** When issuing a blocking Winsock call such as **WSASend** with the *lpOverlapped* parameter set to NULL, Winsock may need to wait for a network event before the call can complete. Winsock performs an alertable wait in this situation, which can be interrupted by an asynchronous procedure call (APC) scheduled on the same thread. Issuing another blocking Winsock call inside an APC that interrupted an ongoing blocking Winsock call on the same thread will lead to undefined behavior, and must never be attempted by Winsock clients.

Overlapped Socket I/O

If an overlapped operation completes immediately, **WSASend** returns a value of zero and the *lpNumberOfBytesSent* parameter is updated with the number of bytes sent. If the overlapped operation is successfully initiated and will complete later, **WSASend** returns SOCKET_ERROR and indicates error code WSA_IO_PENDING. In this case, *lpNumberOfBytesSent* is not updated. When the overlapped operation completes

the amount of data transferred is indicated either through the *cbTransferred* parameter in the completion routine (if specified), or through the *lpcbTransfer* parameter in WSAGetOverlappedResult.

> **Note** All I/O initiated by a given thread is canceled when that thread exits. For overlapped sockets, pending asynchronous operations can fail if the thread is closed before the operations complete. For more information, see ExitThread.

The **WSASend** function using overlapped I/O can be called from within the completion routine of a previous WSARecv, WSARecvFrom, **WSASend**, or WSASendTo function. This enables time-sensitive data transmissions to occur entirely within a preemptive context.

The *lpOverlapped* parameter must be valid for the duration of the overlapped operation. If multiple I/O operations are simultaneously outstanding, each must reference a separate WSAOVERLAPPED structure.

If the *lpCompletionRoutine* parameter is **NULL**, the *hEvent* parameter of *lpOverlapped* is signaled when the overlapped operation completes if it contains a valid event object handle. An application can use WSAWaitForMultipleEvents or WSAGetOverlappedResult to wait or poll on the event object.

If *lpCompletionRoutine* is not **NULL**, the *hEvent* parameter is ignored and can be used by the application to pass context information to the completion routine. A caller that passes a non-**NULL** *lpCompletionRoutine* and later calls WSAGetOverlappedResult for the same overlapped I/O request may not set the *fWait* parameter for that invocation of **WSAGetOverlappedResult** to **TRUE**. In this case the usage of the *hEvent* parameter is undefined, and attempting to wait on the *hEvent* parameter would produce unpredictable results.

The completion routine follows the same rules as stipulated for Windows file I/O completion routines. The completion routine will not be invoked until the thread is in an alertable wait state such as can occur when the function WSAWaitForMultipleEvents with the *fAlertable* parameter set to **TRUE** is invoked.

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

The following C++ code example is a prototype of the completion routine.

```
void CALLBACK CompletionROUTINE(
  IN DWORD dwError,
  IN DWORD cbTransferred,
  IN LPWSAOVERLAPPED lpOverlapped,
  IN DWORD dwFlags
);
```

The CompletionRoutine function is a placeholder for an application-defined or library-defined function name. The *dwError* parameter specifies the completion status for the overlapped operation as indicated by *lpOverlapped*. *cbTransferred* specifies the number of bytes sent. Currently there are no flag values defined and *dwFlags* will be zero. This function does not return a value.

Returning from this function allows invocation of another pending completion routine for this socket. All waiting completion routines are called before the alertable thread's wait is satisfied with a return code of **WSA_IO_COMPLETION**. The completion routines can be called in any order, not necessarily in the same order the overlapped operations are completed. However, the posted buffers are guaranteed to be sent in the same order they are specified.

The order of calls made to **WSASend** is also the order in which the buffers are transmitted to the transport layer. **WSASend** should not be called on the same stream-oriented socket concurrently from different threads, because some Winsock providers may split a large send request into multiple transmissions, and this may lead to unintended data interleaving from multiple concurrent send requests on the same stream-oriented socket.

Example Code

The following code example shows how to use the **WSASend** function in overlapped I/O mode.

```
#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>

// Need to link with Ws2_32.lib
#pragma comment(lib, "ws2_32.lib")

#define DATA_BUFSIZE 4096
#define SEND_COUNT   10

int __cdecl main()
{
    WSADATA wsd;

    struct addrinfo *result = NULL;
    struct addrinfo hints;
    WSAOVERLAPPED SendOverlapped;

    SOCKET ListenSocket = INVALID_SOCKET;
    SOCKET AcceptSocket = INVALID_SOCKET;

    WSABUF DataBuf;
    DWORD SendBytes;
    DWORD Flags;

    char buffer[DATA_BUFSIZE];

    int err = 0;
    int rc, i;

    // Load Winsock
    rc = WSAStartup(MAKEWORD(2, 2), &wsd);
    if (rc != 0) {
        printf("Unable to load Winsock: %d\n", rc);
        return 1;
    }

    // Make sure the hints struct is zeroed out
    SecureZeroMemory((PVOID) & hints, sizeof(struct addrinfo));

    // Initialize the hints to obtain the
    // wildcard bind address for IPv4
    hints.ai_family = AF_INET;
    hints.ai_socktype = SOCK_STREAM;
    hints.ai_protocol = IPPROTO_TCP;
    hints.ai_flags = AI_PASSIVE;

    rc = getaddrinfo(NULL, "27015", &hints, &result);
    if (rc != 0) {
        printf("getaddrinfo failed with error: %d\n", rc);
        return 1;
    }

    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);
```

```
        freeaddrinfo(result);
        return 1;
    }

    rc = bind(ListenSocket, result->ai_addr, (int) result->ai_addrlen);
    if (rc == SOCKET_ERROR) {
        printf("bind failed with error: %d\n", WSAGetLastError());
        freeaddrinfo(result);
        closesocket(ListenSocket);
        return 1;
    }

    rc = listen(ListenSocket, 1);
    if (rc == SOCKET_ERROR) {
        printf("listen failed with error: %d\n", WSAGetLastError());
        freeaddrinfo(result);
        closesocket(ListenSocket);
        return 1;
    }
    // Accept an incoming connection request
    AcceptSocket = accept(ListenSocket, NULL, NULL);
    if (AcceptSocket == INVALID_SOCKET) {
        printf("accept failed with error: %d\n", WSAGetLastError());
        freeaddrinfo(result);
        closesocket(ListenSocket);
        return 1;
    }

    printf("Client Accepted...\n");

    // Make sure the SendOverlapped struct is zeroed out
    SecureZeroMemory((PVOID) & SendOverlapped, sizeof (WSAOVERLAPPED));

    // Create an event handle and setup the overlapped structure.
    SendOverlapped.hEvent = WSACreateEvent();
    if (SendOverlapped.hEvent == NULL) {
        printf("WSACreateEvent failed with error: %d\n", WSAGetLastError());
        freeaddrinfo(result);
        closesocket(ListenSocket);
        closesocket(AcceptSocket);
        return 1;
    }

    DataBuf.len = DATA_BUFSIZE;
    DataBuf.buf = buffer;

    for (i = 0; i < SEND_COUNT; i++) {

        rc = WSASend(AcceptSocket, &DataBuf, 1,
                     &SendBytes, 0, &SendOverlapped, NULL);
        if ((rc == SOCKET_ERROR) &&
            (WSA_IO_PENDING != (err = WSAGetLastError()))) {
            printf("WSASend failed with error: %d\n", err);
            break;
        }

        rc = WSAWaitForMultipleEvents(1, &SendOverlapped.hEvent, TRUE, INFINITE,
                                      TRUE);
        if (rc == WSA_WAIT_FAILED) {
            printf("WSAWaitForMultipleEvents failed with error: %d\n",
                   WSAGetLastError());
            break;
        }

        rc = WSAGetOverlappedResult(AcceptSocket, &SendOverlapped, &SendBytes,
                                    FALSE, &Flags);
        if (rc == FALSE) {
            printf("WSASend failed with error: %d\n", WSAGetLastError());
            break;
```

```
        }

        printf("Wrote %d bytes\n", SendBytes);

        WSAResetEvent(SendOverlapped.hEvent);

    }

    WSACloseEvent(SendOverlapped.hEvent);
    closesocket(AcceptSocket);
    closesocket(ListenSocket);
    freeaddrinfo(result);

    WSACleanup();

    return 0;
}
```

Windows Phone 8: This function is supported for Windows Phone Store apps on Windows Phone 8 and later.

Windows 8.1 and **Windows Server 2012 R2**: This function is supported for Windows Store apps on Windows 8.1, Windows Server 2012 R2, and later.

Requirements

Minimum supported client	Windows 8.1, Windows Vista [desktop apps | UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps | UWP apps]
Target Platform	Windows
Header	winsock2.h
Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

WSABUF

WSACloseEvent

WSAConnect

WSACreateEvent

WSAGetOverlappedResult

WSAOVERLAPPED

WSASocket

WSAWaitForMultipleEvents

Winsock Functions

Winsock Reference

connect

send

socket

WSASendDisconnect function (winsock2.h)

10/22/2022 • 2 minutes to read • Edit Online

The **WSASendDisconnect** function initiates termination of the connection for the socket and sends disconnect data.

Syntax

```
int WSAAPI WSASendDisconnect(
  [in] SOCKET   s,
  [in] LPWSABUF lpOutboundDisconnectData
);
```

Parameters

`[in] s`

Descriptor identifying a socket.

`[in] lpOutboundDisconnectData`

A pointer to the outgoing disconnect data.

Return value

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

ERROR CODE	MEANING
WSANOTINITIALISED	A successful WSAStartup call must occur before using this function.
WSAENETDOWN	The network subsystem has failed.
WSAENOPROTOOPT	The parameter *lpOutboundDisconnectData* is not **NULL**, and the disconnect data is not supported by the service provider.
WSAEINPROGRESS	A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function.
WSAENOTCONN	The socket is not connected (connection-oriented sockets only).
WSAENOTSOCK	The descriptor is not a socket.

WSAEFAULT	The *lpOutboundDisconnectData* parameter is not completely contained in a valid part of the user address space.

Remarks

The **WSASendDisconnect** function is used on connection-oriented sockets to disable transmission and to initiate termination of the connection along with the transmission of disconnect data, if any. This is equivalent to a shutdown (SD_SEND), except that **WSASendDisconnect** also allows sending disconnect data (in protocols that support it).

After this function has been successfully issued, subsequent sends are disallowed.

The *lpOutboundDisconnectData* parameter, if not **NULL**, points to a buffer containing the outgoing disconnect data to be sent to the remote party for retrieval by using WSARecvDisconnect.

> **Note** The native implementation of TCP/IP on Windows does not support disconnect data. Disconnect data is only supported with Windows Sockets providers that have the XP1_DISCONNECT_DATA flag in their WSAPROTOCOL_INFO structure. Use the WSAEnumProtocols function to obtain **WSAPROTOCOL_INFO** structures for all installed providers.

The **WSASendDisconnect** function does not close the socket, and resources attached to the socket will not be freed until closesocket is invoked.
The **WSASendDisconnect** function does not block regardless of the SO_LINGER setting on the socket.

An application should not rely on being able to reuse a socket after calling **WSASendDisconnect**. In particular, a Windows Sockets provider is not required to support the use of connect/WSAConnect on such a socket.

> **Note** When issuing a blocking Winsock call such as **WSASendDisconnect**, Winsock may need to wait for a network event before the call can complete. Winsock performs an alertable wait in this situation, which can be interrupted by an asynchronous procedure call (APC) scheduled on the same thread. Issuing another blocking Winsock call inside an APC that interrupted an ongoing blocking Winsock call on the same thread will lead to undefined behavior, and must never be attempted by Winsock clients.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winsock2.h
Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

Winsock Functions

Winsock Reference

connect

socket

WSASendMsg function (winsock2.h)

10/22/2022 • 11 minutes to read • Edit Online

The **WSASendMsg** function sends data and optional control information from connected and unconnected sockets.

> **Note** This function is a Microsoft-specific extension to the Windows Sockets specification.

Syntax

```
int WSAAPI WSASendMsg(
  [in]  SOCKET                             Handle,
  [in]  LPWSAMSG                           lpMsg,
  [in]  DWORD                              dwFlags,
  [out] LPDWORD                            lpNumberOfBytesSent,
  [in]  LPWSAOVERLAPPED                    lpOverlapped,
  [in]  LPWSAOVERLAPPED_COMPLETION_ROUTINE lpCompletionRoutine
);
```

Parameters

`[in] Handle`

A descriptor identifying the socket.

`[in] lpMsg`

A WSAMSG structure storing the Posix.1g **msghdr** structure.

`[in] dwFlags`

The flags used to modify the behavior of the **WSASendMsg** function call. For more information, see Using *dwFlags* in the Remarks section.

`[out] lpNumberOfBytesSent`

A pointer to the number, in bytes, sent by this call if the I/O operation completes immediately.

Use **NULL** for this parameter if the *lpOverlapped* parameter is not **NULL** to avoid potentially erroneous results. This parameter can be **NULL** only if the *lpOverlapped* parameter is not **NULL**.

`[in] lpOverlapped`

A pointer to a WSAOVERLAPPED structure. Ignored for non-overlapped sockets.

`[in] lpCompletionRoutine`

Type: _In_opt_ **LPWSAOVERLAPPED_COMPLETION_ROUTINE**

A pointer to the completion routine called when the send operation completes. Ignored for non-overlapped sockets.

Return value

Returns zero when successful and immediate completion occurs. When zero is returned, the specified completion routine is called when the calling thread is in the alertable state.

A return value of **SOCKET_ERROR**, and subsequent call to WSAGetLastError that returns WSA_IO_PENDING, indicates the overlapped operation has successfully initiated; completion is then indicated through other means, such as through events or completion ports.

Upon failure, returns **SOCKET_ERROR** and a subsequent call to WSAGetLastError returns a value other than **WSA_IO_PENDING**. The following table lists error codes.

ERROR CODE	MEANING
WSAEACCES	The requested address is a broadcast address, but the appropriate flag was not set.
WSAECONNRESET	For a UDP datagram socket, this error would indicate that a previous send operation resulted in an ICMP "Port Unreachable" message.
WSAEFAULT	The *lpMsg*, *lpNumberOfBytesSent*, *lpOverlapped*, or *lpCompletionRoutine* parameter is not totally contained in a valid part of the user address space. This error is also returned if a **name** member of the WSAMSG structure pointed to by the *lpMsg* parameter was a **NULL** pointer and the **namelen** member of the **WSAMSG** structure was not set to zero. This error is also returned if a **Control.buf** member of the **WSAMSG** structure pointed to by the *lpMsg* parameter was a **NULL** pointer and the **Control.len** member of the **WSAMSG** structure was not set to zero.
WSAEINPROGRESS	A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function.
WSAEINTR	A blocking Windows Socket 1.1 call was canceled through WSACancelBlockingCall.
WSAEINVAL	The socket has not been bound with bind, or the socket was not created with the overlapped flag.
WSAEMSGSIZE	The socket is message oriented, and the message is larger than the maximum supported by the underlying transport.
WSAENETDOWN	The network subsystem has failed.
WSAENETRESET	For a datagram socket, this error indicates that the time to live has expired.
WSAENETUNREACH	The network is unreachable.

WSAENOBUFS	The Windows Sockets provider reports a buffer deadlock.
WSAENOTCONN	The socket is not connected.
WSAENOTSOCK	The descriptor is not a socket.
WSAEOPNOTSUPP	The socket operation is not supported. This error is returned if the **dwFlags** member of the WSAMSG structure pointed to by the *lpMsg* parameter includes any control flags invalid for WSASendMsg.
WSAESHUTDOWN	The socket has been shut down; it is not possible to call the WSASendMsg function on a socket after shutdown has been invoked with *how* set to SD_SEND or SD_BOTH.
WSAETIMEDOUT	The socket timed out. This error is returned if the socket had a wait timeout specified using the **SO_SNDTIMEO** socket option and the timeout was exceeded.
WSAEWOULDBLOCK	Overlapped sockets: There are too many outstanding overlapped I/O requests. Nonoverlapped sockets: The socket is marked as nonblocking and the send operation cannot be completed immediately.
WSANOTINITIALISED	A successful WSAStartup call must occur before using this function.
WSA_IO_PENDING	An overlapped operation was successfully initiated and completion will be indicated at a later time.
WSA_OPERATION_ABORTED	The overlapped operation has been canceled due to the closure of the socket or due to the execution of the **SIO_FLUSH** command in WSAIoctl.

Remarks

The **WSASendMsg** function can be used in place of the WSASend and WSASendTo functions. The **WSASendMsg** function can only be used with datagrams and raw sockets. The socket descriptor in the *s* parameter must be opened with the socket type set to **SOCK_DGRAM** or **SOCK_RAW**.

The *dwFlags* parameter can only contain a combination of the following control flags: **MSG_DONTROUTE**, **MSG_PARTIAL**, and **MSG_OOB**. The **dwFlags** member of the WSAMSG structure pointed to by the *lpMsg* parameter is ignored on input and not used on output.

> **Note** The function pointer for the **WSASendMsg** function must be obtained at run time by making a call to the WSAIoctl function with the **SIO_GET_EXTENSION_FUNCTION_POINTER** opcode specified. The input buffer passed to the **WSAIoctl** function must contain **WSAID_WSASENDMSG**, a globally unique identifier (GUID) whose value identifies the **WSASendMsg** extension function. On success, the output returned by the **WSAIoctl** function contains a pointer to the **WSASendMsg** function. The **WSAID_WSASENDMSG** GUID is defined in the *Mswsock.h* header file.

Overlapped sockets are created with a WSASocket function call that has the **WSA_FLAG_OVERLAPPED** flag set. For overlapped sockets, sending information uses overlapped I/O unless both *lpOverlapped* and *lpCompletionRoutine* are **NULL**; when *lpOverlapped* and *lpCompletionRoutine* are **NULL**, the socket is treated as a nonoverlapped socket. A completion indication occurs with overlapped sockets; once the buffer or buffers have been consumed by the transport, a completion routine is triggered or an event object is set. If the operation does not complete immediately, the final completion status is retrieved through the completion routine or by calling the WSAGetOverlappedResult function.

For nonoverlapped sockets, the *lpOverlapped* and *lpCompletionRoutine* parameters are ignored and **WSASendMsg** adopts the same blocking semantics as the send function: data is copied from the buffer or buffers into the transport's buffer. If the socket is nonblocking and stream oriented, and there is insufficient space in the transport's buffer, **WSASendMsg** returns with only part of the application's buffers having been consumed. In contrast, this buffer situation on a blocking socket results in **WSASendMsg** blocking until all of the application's buffer contents have been consumed.

If this function is completed in an overlapped manner, it is the Winsock service provider's responsibility to capture this WSABUF structure before returning from this call. This enables applications to build stack-based **WSABUF** arrays pointed to by the **lpBuffers** member of the WSAMSG structure pointed to by the *lpMsg* parameter.

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

On an IPv4 socket of type **SOCK_DGRAM** or **SOCK_RAW**, an application can specific the local IP source address to use for sending with the **WSASendMsg** function. One of the control data objects passed in the WSAMSG structure to the **WSASendMsg** function may contain an in_pktinfo structure used to specify the local IPv4 source address to use for sending.

On an IPv6 socket of type **SOCK_DGRAM** or **SOCK_RAW**, an application can specific the local IP source address to use for sending with the **WSASendMsg** function. One of the control data objects passed in the WSAMSG structure to the **WSASendMsg** function may contain an in6_pktinfo structure used to specify the local IPv6 source address to use for sending.

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.

> **Note** The **SO_SNDTIMEO** socket option applies only to blocking sockets.

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

> **Note** When issuing a blocking Winsock call such as **WSASendMsg** with the *lpOverlapped* parameter set to NULL, Winsock may need to wait for a network event before the call can complete. Winsock performs an alertable wait in this situation, which can be interrupted by an asynchronous procedure call (APC) scheduled on the same thread. Issuing another blocking Winsock call inside an APC that interrupted an ongoing blocking Winsock call on the same thread will lead to undefined behavior, and must never be attempted by Winsock clients.

dwFlags

The *dwFlags* input parameter can be used to influence the behavior of the function invocation beyond the options specified for the associated socket. That is, the semantics of this function are determined by the socket options and the *dwFlags* parameter. The latter is constructed by using the bitwise OR operator with any of the following values.

VALUE	MEANING
MSG_DONTROUTE	Specifies that the data should not be subject to routing. A Windows Sockets service provider can choose to ignore this flag.
MSG_PARTIAL	Specifies that *lpMsg->lpBuffers* contains only a partial message. Note that the error code WSAEOPNOTSUPP will be returned by transports that do not support partial message transmissions.

The possible values for *dwFlags* parameter are defined in the *Winsock2.h* header file.

On output, the **dwFlags** member of the WSAMSG structure pointed to by the *lpMsg* parameter is not used.

Overlapped Socket I/O

If an overlapped operation completes immediately, **WSASendMsg** returns a value of zero and the *lpNumberOfBytesSent* parameter is updated with the number of bytes sent. If the overlapped operation is successfully initiated and will complete later, **WSASendMsg** returns SOCKET_ERROR and indicates error code WSA_IO_PENDING. In this case, *lpNumberOfBytesSent* is not updated. When the overlapped operation completes, the amount of data transferred is indicated either through the *cbTransferred* parameter in the completion routine (if specified) or through the *lpcbTransfer* parameter in WSAGetOverlappedResult.

> **Note** All I/O initiated by a given thread is canceled when that thread exits. For overlapped sockets, pending asynchronous operations can fail if the thread is closed before the operations complete. See ExitThread for more information.

The **WSASendMsg** function using overlapped I/O can be called from within the completion routine of a previous , WSARecv, WSARecvFrom, LPFN_WSARECVMSG (WSARecvMsg), WSASend, **WSASendMsg**, or WSASendTo function. This permits time-sensitive data transmissions to occur entirely within a preemptive context.

The *lpOverlapped* parameter must be valid for the duration of the overlapped operation. If multiple I/O operations are simultaneously outstanding, each must reference a separate WSAOVERLAPPED structure.

If the *lpCompletionRoutine* parameter is **NULL**, the *hEvent* parameter of *lpOverlapped* is signaled when the overlapped operation completes if it contains a valid event object handle. An application can use WSAWaitForMultipleEvents or WSAGetOverlappedResult to wait or poll on the event object.

If *lpCompletionRoutine* is not **NULL**, the *hEvent* parameter is ignored and can be used by the application to pass context information to the completion routine. A caller that passes a non-**NULL** *lpCompletionRoutine* and later calls WSAGetOverlappedResult for the same overlapped I/O request may not set the *fWait* parameter for that

invocation of **WSAGetOverlappedResult** to **TRUE**. In this case, the usage of the *hEvent* parameter is undefined, and attempting to wait on the *hEvent* parameter would produce unpredictable results.

The completion routine follows the same rules as stipulated for Windows file I/O completion routines. The completion routine will not be invoked until the thread is in an alertable wait state, for example, with WSAWaitForMultipleEvents called with the *fAlertable* parameter set to **TRUE**.

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

The prototype of the completion routine is as follows.

```
void CALLBACK CompletionRoutine(
  IN DWORD dwError,
  IN DWORD cbTransferred,
  IN LPWSAOVERLAPPED lpOverlapped,
  IN DWORD dwFlags
);
```

The **CompletionRoutine** function is a placeholder for an application-defined or library-defined function name. The *dwError* parameter specifies the completion status for the overlapped operation as indicated by the *lpOverlapped* parameter. The *cbTransferred* parameter indicates the number of bytes sent. Currently there are no flag values defined and the *dwFlags* parameter will be zero. The **CompletionRoutine** function does not return a value.

Returning from this function allows invocation of another pending completion routine for the socket. All waiting completion routines are called before the alertable thread's wait is satisfied with a return code of WSA_IO_COMPLETION. The completion routines can be called in any order, not necessarily in the same order the overlapped operations are completed. However, the posted buffers are guaranteed to be sent in the same order they are specified.

Windows 8.1 and **Windows Server 2012 R2**: This function is supported for Windows Store apps on Windows 8.1, Windows Server 2012 R2, and later.

Requirements

Minimum supported client	Windows 8.1, Windows Vista [desktop apps | UWP apps]
Minimum supported server	Windows Server 2008 [desktop apps | UWP apps]
Target Platform	Windows
Header	winsock2.h (include Mswsock.h)
Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

ExitThread

IPV6_PKTINFO

IP_PKTINFO

WSABUF

WSACancelBlockingCall

WSAGetLastError

WSAGetOverlappedResult

WSAIoctl

WSAMSG

WSAOVERLAPPED

WSASend

WSASendTo

WSASocket

WSAStartup

WSAWaitForMultipleEvents

Winsock Functions

Winsock Reference

bind

in6_pktinfo

in_pktinfo

send

shutdown

WSASendTo function (winsock2.h)

10/22/2022 • 13 minutes to read • Edit Online

The **WSASendTo** function sends data to a specific destination, using overlapped I/O where applicable.

Syntax

```
int WSAAPI WSASendTo(
  [in]  SOCKET                             s,
  [in]  LPWSABUF                           lpBuffers,
  [in]  DWORD                              dwBufferCount,
  [out] LPDWORD                            lpNumberOfBytesSent,
  [in]  DWORD                              dwFlags,
  [in]  const sockaddr                     *lpTo,
  [in]  int                                iTolen,
  [in]  LPWSAOVERLAPPED                    lpOverlapped,
  [in]  LPWSAOVERLAPPED_COMPLETION_ROUTINE lpCompletionRoutine
);
```

Parameters

`[in] s`

A descriptor identifying a (possibly connected) socket.

`[in] lpBuffers`

A pointer to an array of WSABUF structures. Each **WSABUF** structure contains a pointer to a buffer and the length of the buffer, in bytes. For a Winsock application, once the **WSASendTo** function is called, the system owns these buffers and the application may not access them. This array must remain valid for the duration of the send operation.

`[in] dwBufferCount`

The number of WSABUF structures in the *lpBuffers* array.

`[out] lpNumberOfBytesSent`

A pointer to the number of bytes sent by this call if the I/O operation completes immediately.

Use **NULL** for this parameter if the *lpOverlapped* parameter is not **NULL** to avoid potentially erroneous results. This parameter can be **NULL** only if the *lpOverlapped* parameter is not **NULL**.

`[in] dwFlags`

The flags used to modify the behavior of the **WSASendTo** function call.

`[in] lpTo`

An optional pointer to the address of the target socket in the SOCKADDR structure.

`[in] iTolen`

The size, in bytes, of the address in the *lpTo* parameter.

`[in] lpOverlapped`

A pointer to a WSAOVERLAPPED structure (ignored for nonoverlapped sockets).

`[in] lpCompletionRoutine`

Type: _In_opt_ **LPWSAOVERLAPPED_COMPLETION_ROUTINE**

A pointer to the completion routine called when the send operation has been completed (ignored for nonoverlapped sockets).

Return value

If no error occurs and the send operation has completed immediately, **WSASendTo** returns zero. In this case, the completion routine will have already been scheduled to be called once the calling thread is in the alertable state. Otherwise, a value of **SOCKET_ERROR** is returned, and a specific error code can be retrieved by calling WSAGetLastError. The error code WSA_IO_PENDING indicates that the overlapped operation has been successfully initiated and that completion will be indicated at a later time. Any other error code indicates that the overlapped operation was not successfully initiated and no completion indication will occur.

ERROR CODE	MEANING
WSAEACCES	The requested address is a broadcast address, but the appropriate flag was not set.
WSAEADDRNOTAVAIL	The remote address is not a valid address (such as ADDR_ANY).
WSAEAFNOSUPPORT	Addresses in the specified family cannot be used with this socket.
WSAECONNRESET	For a UDP datagram socket, this error would indicate that a previous send operation resulted in an ICMP "Port Unreachable" message.
WSAEDESTADDRREQ	A destination address is required.
WSAEFAULT	The *lpBuffers*, *lpTo*, *lpOverlapped*, *lpNumberOfBytesSent*, or *lpCompletionRoutine* parameters are not part of the user address space, or the *lpTo* parameter is too small.
WSAEHOSTUNREACH	A socket operation was attempted to an unreachable host.
WSAEINPROGRESS	A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function.
WSAEINTR	A blocking Windows Socket 1.1 call was canceled through WSACancelBlockingCall.
WSAEINVAL	The socket has not been bound with bind, or the socket is not created with the overlapped flag.

WSAEMSGSIZE	The socket is message oriented, and the message is larger than the maximum supported by the underlying transport.
WSAENETDOWN	The network subsystem has failed.
WSAENETRESET	For a datagram socket, this error indicates that the time to live has expired.
WSAENETUNREACH	The network cannot be reached from this host at this time.
WSAENOBUFS	The Windows Sockets provider reports a buffer deadlock.
WSAENOTCONN	The socket is not connected (connection-oriented sockets only).
WSAENOTSOCK	The descriptor is not a socket.
WSAESHUTDOWN	The socket has been shut down; it is not possible to WSASendTo on a socket after shutdown has been invoked with *how* set to SD_SEND or SD_BOTH.
WSAEWOULDBLOCK	**Windows NT:** Overlapped sockets: there are too many outstanding overlapped I/O requests. Nonoverlapped sockets: The socket is marked as nonblocking and the send operation cannot be completed immediately.
WSANOTINITIALISED	A successful WSAStartup call must occur before using this function.
WSA_IO_PENDING	An overlapped operation was successfully initiated and completion will be indicated at a later time.
WSA_OPERATION_ABORTED	The overlapped operation has been canceled due to the closure of the socket, or the execution of the SIO_FLUSH command in WSAIoctl.

Remarks

The **WSASendTo** function provides enhanced features over the standard sendto function in two important areas:

- It can be used in conjunction with overlapped sockets to perform overlapped send operations.
- It allows multiple send buffers to be specified making it applicable to the scatter/gather type of I/O.

The **WSASendTo** function is normally used on a connectionless socket specified by *s* to send a datagram contained in one or more buffers to a specific peer socket identified by the *lpTo* parameter. Even if the connectionless socket has been previously connected using the connect function to a specific address, *lpTo* overrides the destination

address for that particular datagram only. On a connection-oriented socket, the *lpTo* and *iToLen* parameters are ignored; in this case, the **WSASendTo** is equivalent to WSASend.

For overlapped sockets (created using WSASocket with flag **WSA_FLAG_OVERLAPPED**) sending data uses overlapped I/O, unless both *lpOverlapped* and *lpCompletionRoutine* are **NULL** in which case the socket is treated as a nonoverlapped socket. A completion indication will occur (invoking the completion routine or setting of an event object) when the buffer(s) have been consumed by the transport. If the operation does not complete immediately, the final completion status is retrieved through the completion routine or WSAGetOverlappedResult.

> **Note** If a socket is opened, a setsockopt call is made, and then a sendto call is made, Windows Sockets performs an implicit bind function call.

If both *lpOverlapped* and *lpCompletionRoutine* are **NULL**, the socket in this function will be treated as a nonoverlapped socket.

For nonoverlapped sockets, the last two parameters (*lpOverlapped*, *lpCompletionRoutine*) are ignored and **WSASendTo** adopts the same blocking semantics as send. Data is copied from the buffer(s) into the transport buffer. If the socket is nonblocking and stream oriented, and there is not sufficient space in the transport's buffer, **WSASendTo** returns with only part of the application's buffers having been consumed. Given the same buffer situation and a blocking socket, **WSASendTo** will block until all of the application's buffer contents have been consumed.

If this function is completed in an overlapped manner, it is the Winsock service provider's responsibility to capture the WSABUF structures before returning from this call. This enables applications to build stack-based **WSABUF** arrays pointed to by the *lpBuffers* parameter.

For message-oriented sockets, care must be taken not to exceed the maximum message size of the underlying transport, which can be obtained by getting the value of socket option **SO_MAX_MSG_SIZE**. If the data is too long to pass atomically through the underlying protocol the error WSAEMSGSIZE is returned, and no data is transmitted.

If the socket is unbound, unique values are assigned to the local association by the system, and the socket is then marked as bound.

If the socket is connected, the getsockname function can be used to determine the local IP address and port associated with the socket.

If the socket is not connected, the getsockname function can be used to determine the local port number associated with the socket but the IP address returned is set to the wildcard address for the given protocol (for example, INADDR_ANY or "0.0.0.0" for IPv4 and IN6ADDR_ANY_INIT or "::" for IPv6).

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

The *dwFlags* parameter can be used to influence the behavior of the function invocation beyond the options specified for the associated socket. That is, the semantics of this function are determined by the socket options and the *dwFlags* parameter. The latter is constructed by using the bitwise OR operator with any of any of the values listed in the following table.

VALUE	MEANING
MSG_DONTROUTE	Specifies that the data should not be subject to routing. A Windows Socket service provider may choose to ignore this flag.

MSG_OOB	Send OOB data (stream-style socket such as **SOCK_STREAM** only).
MSG_PARTIAL	Specifies that *lpBuffers* only contains a partial message. Be aware that the error code WSAEOPNOTSUPP will be returned by transports that do not support partial message transmissions.

> **Note** When issuing a blocking Winsock call such as **WSASendTo** with the *lpOverlapped* parameter set to **NULL**, Winsock may need to wait for a network event before the call can complete. Winsock performs an alertable wait in this situation, which can be interrupted by an asynchronous procedure call (APC) scheduled on the same thread. Issuing another blocking Winsock call inside an APC that interrupted an ongoing blocking Winsock call on the same thread will lead to undefined behavior, and must never be attempted by Winsock clients.

Overlapped Socket I/O

If an overlapped operation completes immediately, **WSASendTo** returns a value of zero and the *lpNumberOfBytesSent* parameter is updated with the number of bytes sent. If the overlapped operation is successfully initiated and will complete later, **WSASendTo** returns **SOCKET_ERROR** and indicates error code WSA_IO_PENDING. In this case, *lpNumberOfBytesSent* is not updated. When the overlapped operation completes the amount of data transferred is indicated either through the *cbTransferred* parameter in the completion routine (if specified), or through the *lpcbTransfer* parameter in WSAGetOverlappedResult.

> **Note** All I/O initiated by a given thread is canceled when that thread exits. For overlapped sockets, pending asynchronous operations can fail if the thread is closed before the operations complete. See ExitThread for more information.

The **WSASendTo** function using overlapped I/O can be called from within the completion routine of a previous WSARecv, WSARecvFrom, WSASend, or **WSASendTo** function. This permits time-sensitive data transmissions to occur entirely within a preemptive context.

The *lpOverlapped* parameter must be valid for the duration of the overlapped operation. If multiple I/O operations are simultaneously outstanding, each must reference a separate WSAOVERLAPPED structure.

If the *lpCompletionRoutine* parameter is **NULL**, the *hEvent* parameter of *lpOverlapped* is signaled when the overlapped operation completes if it contains a valid event object handle. An application can use WSAWaitForMultipleEvents or WSAGetOverlappedResult to wait or poll on the event object.

If *lpCompletionRoutine* is not **NULL**, the *hEvent* parameter is ignored and can be used by the application to pass context information to the completion routine. A caller that passes a non-**NULL** *lpCompletionRoutine* and later calls WSAGetOverlappedResult for the same overlapped I/O request may not set the *fWait* parameter for that invocation of **WSAGetOverlappedResult** to **TRUE**. In this case the usage of the *hEvent* parameter is undefined, and attempting to wait on the *hEvent* parameter would produce unpredictable results.

The completion routine follows the same rules as stipulated for Windows file I/O completion routines. The completion routine will not be invoked until the thread is in an alertable wait state such as can occur when the function WSAWaitForMultipleEvents with the *fAlertable* parameter set to **TRUE** is invoked.

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

The prototype of the completion routine is as follows.

```
void CALLBACK CompletionROUTINE(
  IN DWORD dwError,
  IN DWORD cbTransferred,
  IN LPWSAOVERLAPPED lpOverlapped,
  IN DWORD dwFlags
);
```

The CompletionRoutine function is a placeholder for an application-defined or library-defined function name. The *dwError* parameter specifies the completion status for the overlapped operation as indicated by *lpOverlapped*. The *cbTransferred* parameter specifies the number of bytes sent. Currently there are no flag values defined and *dwFlags* will be zero. This function does not return a value.

Returning from this function allows invocation of another pending completion routine for this socket. All waiting completion routines are called before the alertable thread's wait is satisfied with a return code of WSA_IO_COMPLETION. The completion routines can be called in any order, not necessarily in the same order in which the overlapped operations are completed. However, the posted buffers are guaranteed to be sent in the same order they are specified.

Example Code

The following example demonstrates the use of the **WSASendTo** function using an event object.

```
#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")

int __cdecl main(int argc, char **argv)
{

    //-----------------------------------------------
    // Declare and initialize variables
    WSADATA wsaData;
    WSABUF DataBuf;

    WSAOVERLAPPED Overlapped;
    SOCKET SendToSocket = INVALID_SOCKET;

    struct sockaddr_in RecvAddr;
    struct sockaddr_in LocalAddr;
    int RecvAddrSize = sizeof (RecvAddr);
    int LocalAddrSize = sizeof (LocalAddr);

    u_short Port = 27777;
    struct hostent *localHost;
    char *ip;

    char *targetip;
    char *targetport;

    char SendBuf[1024] = "Data buffer to send";
    int BufLen = 1024;
    DWORD BytesSent = 0;
    DWORD Flags = 0;

    int rc, err;
    int retval = 0;
```

```
    // Validate the parameters
    if (argc != 3) {
        printf("usage: %s targetip port\n", argv[0]);
        printf("  to sendto the localhost on port 27777\n");
        printf("       %s 127.0.0.1 27777\n", argv[0]);
        return 1;
    }

    targetip = argv[1];
    targetport = argv[2];

    //-----------------------------------------------
    // Initialize Winsock
    // Load Winsock
    rc = WSAStartup(MAKEWORD(2, 2), &wsaData);
    if (rc != 0) {
        printf("Unable to load Winsock: %d\n", rc);
        return 1;
    }

    // Make sure the Overlapped struct is zeroed out
    SecureZeroMemory((PVOID) &Overlapped, sizeof(WSAOVERLAPPED));

    // Create an event handle and setup the overlapped structure.
    Overlapped.hEvent = WSACreateEvent();
    if (Overlapped.hEvent == WSA_INVALID_EVENT) {
        printf("WSACreateEvent failed with error: %d\n", WSAGetLastError());
        WSACleanup();
        return 1;
    }
    //-----------------------------------------------
    // Create a socket for sending data
    SendToSocket =
        WSASocket(AF_INET, SOCK_DGRAM, IPPROTO_UDP, NULL, 0,
                  WSA_FLAG_OVERLAPPED);
    if (SendToSocket == INVALID_SOCKET) {
        printf("socket failed with error: %d\n", WSAGetLastError());
        WSACloseEvent(Overlapped.hEvent);
        WSACleanup();
        return 1;
    }
    //-----------------------------------------------
    // Set up the RecvAddr structure with the IP address of
    // the receiver (in this example case "123.123.123.1")
    // and the specified port number.
    RecvAddr.sin_family = AF_INET;

    RecvAddr.sin_addr.s_addr = inet_addr(targetip);
    if (RecvAddr.sin_addr.s_addr == INADDR_NONE)  {
        printf("The target ip address entered must be a legal IPv4 address\n");
        WSACloseEvent(Overlapped.hEvent);
        WSACleanup();
        return 1;
    }
    RecvAddr.sin_port = htons( (u_short) atoi(targetport));
    if(RecvAddr.sin_port == 0) {
        printf("The targetport must be a legal UDP port number\n");
        WSACloseEvent(Overlapped.hEvent);
        WSACleanup();
        return 1;
    }

    //-----------------------------------------------
    // Set up the LocalAddr structure with the local IP address
    // and the specified port number.
    localHost = gethostbyname("");
    ip = inet_ntoa(*(struct in_addr *) *localHost->h_addr_list);
```

```
    LocalAddr.sin_family = AF_INET;
    LocalAddr.sin_addr.s_addr = inet_addr(ip);
    LocalAddr.sin_port = htons(Port);

    //---------------------------------------------
    // Bind the sending socket to the LocalAddr structure
    // that has the internet address family, local IP address
    // and specified port number.
    rc = bind(SendToSocket, (struct sockaddr *) &LocalAddr, LocalAddrSize);
    if (rc == SOCKET_ERROR) {
        printf("bind failed with error: %d\n", WSAGetLastError());
        WSACloseEvent(Overlapped.hEvent);
        closesocket(SendToSocket);
        WSACleanup();
        return 1;
    }
    //---------------------------------------------
    // Send a datagram to the receiver
    printf("Sending datagram from IPv4 address = %s port=%d\n",
       inet_ntoa(LocalAddr.sin_addr), ntohs(LocalAddr.sin_port) );
    printf("   to IPv4 address = %s port=%d\n",
       inet_ntoa(RecvAddr.sin_addr), ntohs(RecvAddr.sin_port) );

//    printf("Sending a datagram...\n");
    DataBuf.len = BufLen;
    DataBuf.buf = SendBuf;
    rc = WSASendTo(SendToSocket, &DataBuf, 1,
                   &BytesSent, Flags, (SOCKADDR *) & RecvAddr,
                   RecvAddrSize, &Overlapped, NULL);

    if ((rc == SOCKET_ERROR) && (WSA_IO_PENDING != (err = WSAGetLastError()))) {
        printf("WSASendTo failed with error: %d\n", err);
        WSACloseEvent(Overlapped.hEvent);
        closesocket(SendToSocket);
        WSACleanup();
        return 1;
    }

    rc = WSAWaitForMultipleEvents(1, &Overlapped.hEvent, TRUE, INFINITE, TRUE);
    if (rc == WSA_WAIT_FAILED) {
        printf("WSAWaitForMultipleEvents failed with error: %d\n",
                WSAGetLastError());
        retval = 1;
    }

    rc = WSAGetOverlappedResult(SendToSocket, &Overlapped, &BytesSent,
                                FALSE, &Flags);
    if (rc == FALSE) {
        printf("WSASendTo failed with error: %d\n", WSAGetLastError());
        retval = 1;
    }
    else
        printf("Number of sent bytes = %d\n", BytesSent);

    //---------------------------------------------
    // When the application is finished sending, close the socket.
    printf("Finished sending. Closing socket.\n");
    WSACloseEvent(Overlapped.hEvent);
    closesocket(SendToSocket);
    printf("Exiting.\n");

    //---------------------------------------------
    // Clean up and quit.
    WSACleanup();
    return (retval);
}
```

Windows Phone 8: This function is supported for Windows Phone Store apps on Windows Phone 8 and later.

Windows 8.1 and **Windows Server 2012 R2**: This function is supported for Windows Store apps on Windows 8.1, Windows Server 2012 R2, and later.

Requirements

Minimum supported client	Windows 8.1, Windows Vista [desktop apps | UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps | UWP apps]
Target Platform	Windows
Header	winsock2.h
Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

WSACloseEvent

WSACreateEvent

WSAGetOverlappedResult

WSASocket

WSAWaitForMultipleEvents

Winsock Functions

Winsock Reference

WSASERVICECLASSINFOA structure (winsock2.h)

10/22/2022 • 2 minutes to read • Edit Online

The **WSASERVICECLASSINFO** structure contains information about a specified service class. For each service class in Windows Sockets 2, there is a single **WSASERVICECLASSINFO** structure.

Syntax

```
typedef struct _WSAServiceClassInfoA {
  LPGUID            lpServiceClassId;
  LPSTR             lpszServiceClassName;
  DWORD             dwCount;
  LPWSANSCLASSINFOA lpClassInfos;
} WSASERVICECLASSINFOA, *PWSASERVICECLASSINFOA, *LPWSASERVICECLASSINFOA;
```

Members

`lpServiceClassId`

Unique Identifier (GUID) for the service class.

`lpszServiceClassName`

Well known name associated with the service class.

`dwCount`

Number of entries in **lpClassInfos**.

`lpClassInfos`

Array of WSANSCLASSINFO structures that contains information about the service class.

Remarks

> **NOTE**
>
> The winsock2.h header defines WSASERVICECLASSINFO as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see Conventions for Function Prototypes.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	winsock2.h

See also

NSPGetServiceClassInfo

NSPLookupServiceBegin

WSASERVICECLASSINFOW structure (winsock2.h)

10/22/2022 • 2 minutes to read • Edit Online

The **WSASERVICECLASSINFO** structure contains information about a specified service class. For each service class in Windows Sockets 2, there is a single **WSASERVICECLASSINFO** structure.

Syntax

```
typedef struct _WSAServiceClassInfoW {
  LPGUID            lpServiceClassId;
  LPWSTR            lpszServiceClassName;
  DWORD             dwCount;
  LPWSANSCLASSINFOW lpClassInfos;
} WSASERVICECLASSINFOW, *PWSASERVICECLASSINFOW, *LPWSASERVICECLASSINFOW;
```

Members

`lpServiceClassId`

Unique Identifier (GUID) for the service class.

`lpszServiceClassName`

Well known name associated with the service class.

`dwCount`

Number of entries in **lpClassInfos**.

`lpClassInfos`

Array of WSANSCLASSINFO structures that contains information about the service class.

Remarks

> **NOTE**
>
> The winsock2.h header defines WSASERVICECLASSINFO as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see Conventions for Function Prototypes.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	winsock2.h

See also

NSPGetServiceClassInfo

NSPLookupServiceBegin

WSASetBlockingHook function (winsock2.h)

10/22/2022 • 2 minutes to read • Edit Online

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

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

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

Syntax

```
FARPROC WSAAPI WSASetBlockingHook(
  FARPROC lpBlockFunc
);
```

Parameters

`lpBlockFunc`

Return value

None

Requirements

Target Platform	Windows
Header	winsock2.h

See also

Winsock Functions

Winsock Reference

WSASetEvent function (winsock2.h)

10/22/2022 • 2 minutes to read • Edit Online

The **WSASetEvent** function sets the state of the specified event object to signaled.

Syntax

```
BOOL WSAAPI WSASetEvent(
  [in] WSAEVENT hEvent
);
```

Parameters

`[in] hEvent`

Handle that identifies an open event object.

Return value

If the function succeeds, the return value is **TRUE**.

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

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

Remarks

The **WSASetEvent** function sets the state of the event object to be signaled.

Windows Phone 8: This function is supported for Windows Phone Store apps on Windows Phone 8 and later.

Windows 8.1 and **Windows Server 2012 R2**: This function is supported for Windows Store apps on Windows 8.1, Windows Server 2012 R2, and later.

Requirements

Minimum supported client	Windows 8.1, Windows Vista [desktop apps | UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps | UWP apps]
Target Platform	Windows
Header	winsock2.h
Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

WSACloseEvent

WSACreateEvent

WSAResetEvent

Winsock Functions

Winsock Reference

WSASetLastError function (winsock2.h)

10/22/2022 • 2 minutes to read • Edit Online

The **WSASetLastError** function sets the error code that can be retrieved through the WSAGetLastError function.

Syntax

```
void WSAAPI WSASetLastError(
  [in] int iError
);
```

Parameters

`[in] iError`

Integer that specifies the error code to be returned by a subsequent WSAGetLastError call.

Return value

This function generates no return values.

ERROR CODE	MEANING
WSANOTINITIALISED	A successful WSAStartup call must occur before using this function.

Remarks

The **WSASetLastError** function allows an application to set the error code to be returned by a subsequent WSAGetLastError call for the current thread. Note that any subsequent Windows Sockets routine called by the application will override the error code as set by this routine.

The error code set by **WSASetLastError** is different from the error code reset by calling the function getsockopt with SO_ERROR.

The Windows Sockets error codes used by this function are listed under Windows Sockets Error Codes.

Windows Phone 8: This function is supported for Windows Phone Store apps on Windows Phone 8 and later.

Windows 8.1 and **Windows Server 2012 R2**: This function is supported for Windows Store apps on Windows 8.1, Windows Server 2012 R2, and later.

Requirements

Minimum supported client	Windows 8.1, Windows Vista [desktop apps | UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps | UWP apps]

Target Platform	Windows
Header	winsock2.h (include Winsock2.h)
Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

WSAGetLastError

Windows Sockets Error Codes

Winsock Functions

Winsock Reference

getsockopt

WSASetServiceA function (winsock2.h)

10/22/2022 • 5 minutes to read • Edit Online

The **WSASetService** function registers or removes from the registry a service instance within one or more namespaces.

Syntax

```
INT WSAAPI WSASetServiceA(
  [in] LPWSAQUERYSETA   lpqsRegInfo,
  [in] WSAESETSERVICEOP essoperation,
  [in] DWORD            dwControlFlags
);
```

Parameters

`[in] lpqsRegInfo`

A pointer to the service information for registration or deregistration.

`[in] essoperation`

A value that determines that operation requested. This parameter can be one of the values from the WSAESETSERVICEOP enumeration type defined in the *Winsock2.h* header file.

VALUE	MEANING
RNRSERVICE_REGISTER	Register the service. For SAP, this means sending out a periodic broadcast. This is an NOP for the DNS namespace. For persistent data stores, this means updating the address information.
RNRSERVICE_DEREGISTER	Remove the service from the registry. For SAP, this means stop sending out the periodic broadcast. This is an NOP for the DNS namespace. For persistent data stores this means deleting address information.
RNRSERVICE_DELETE	Delete the service from dynamic name and persistent spaces. For services represented by multiple CSADDR_INFO structures (using the SERVICE_MULTIPLE flag), only the specified address will be deleted, and this must match exactly the corresponding **CSADDR_INFO** structure that was specified when the service was registered.

`[in] dwControlFlags`

Service install flags value that further controls the operation performed of the **WSASetService** function. The possible values for this parameter are defined in the *Winsock2.h* header file.

FLAG	MEANING

SERVICE_MULTIPLE	Controls scope of operation. When this flag is not set, service addresses are managed as a group. A register or removal from the registry invalidates all existing addresses before adding the given address set. When set, the action is only performed on the given address set. A register does not invalidate existing addresses and a removal from the registry only invalidates the given set of addresses.

Return value

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

ERROR CODE	MEANING
WSAEACCES	The calling routine does not have sufficient privileges to install the Service.
WSAEINVAL	One or more required parameters were invalid or missing.
WSANOTINITIALISED	The *Ws2_32.dll* has not been initialized. The application must first call WSAStartup before calling any Windows Sockets functions.
WSA_NOT_ENOUGH_MEMORY	There was insufficient memory to perform the operation.

Remarks

The **WSASetService** function can be used to affect a specific namespace provider, all providers associated with a specific namespace, or all providers across all namespaces.

The available values for *essOperation* and *dwControlFlags* combine to control operation of the **WSASetService** function as shown in the following table.

OPERATION	FLAGS	SERVICE ALREADY EXISTS	SERVICE DOES NOT EXIST
RNRSERVICE_REGISTER	None	Overwrites the object. Uses only addresses specified. The object is REGISTERED.	Creates a new object. Uses only addresses specified. Object is REGISTERED.
RNRSERVICE_REGISTER	SERVICE_MULTIPLE	Updates the object. Adds new addresses to the existing set. The object is REGISTERED.	Creates a new object. Uses all addresses specified. Object is REGISTERED.
RNRSERVICE_DEREGISTER	None	Removes all addresses, but does not remove the object from the namespace. The object is removed from the registry.	WSASERVICE_NOT_FOUND

RNRSERVICE_DEREGISTER	SERVICE_MULTIPLE	Updates the object. Removes only addresses that are specified. Only marks the object as DEREGISTERED if no addresses are present. Does not remove the object from the namespace.	WSASERVICE_NOT_FOUND
RNRSERVICE_DELETE	None	Removes the object from the namespace.	WSASERVICE_NOT_FOUND
RNRSERVICE_DELETE	SERVICE_MULTIPLE	Removes only addresses that are specified. Only removes object from the namespace if no addresses remain.	WSASERVICE_NOT_FOUND

Publishing services to directories, such as Active Directory Services, is restricted based on access control lists (ACLs). For more information, see Security Issues for Service Publication.

When the *dwControlFlags* parameter is set to **SERVICE_MULTIPLE**, an application can manage its addresses independently. This is useful when the application wants to manage its protocols individually or when the service resides on more than one computer. For instance, when a service uses more than one protocol, it may find that one listening socket aborts but the other sockets remain operational. In this case, the service could remove the aborted address from the registry without affecting the other addresses.

When the *dwControlFlags* parameter is set to **SERVICE_MULTIPLE**, an application must not let stale addresses remain in the object. This can happen if the application aborts without issuing a DEREGISTER request. When a service registers, it should store its addresses. On its next invocation, the service should explicitly remove these old stale addresses from the registry before registering new addresses.

> **Note** If ANSI character strings are used, there is a chance that the WSAQUERYSET data in *lpqsRegInfo* may not contain any results after this function returns. This is because the ANSI version of this method, **WSASetServiceA**, converts the ANSI data in **WSAQUERYSET** to Unicode internally, but does not convert the results back to ANSI. This primarily impacts transports that return a "service record handle" used to uniquely identify a record. To work around this issue, applications should use Unicode string data in **WSAQUERYSET** when calling this function.

Service Properties

The following table describes how service property data is represented in a WSAQUERYSET structure. Fields labeled as (Optional) can contain a null pointer.

WSAQUERYSET MEMBER	SERVICE PROPERTY DESCRIPTION
dwSize	Must be set to sizeof (WSAQUERYSET). This is a versioning mechanism.
dwOutputFlags	Not applicable and ignored.
lpszServiceInstanceName	Referenced string contains the service instance name.
lpServiceClassId	The GUID corresponding to this service class.

lpVersion	(Optional) Supplies service instance version number.
lpszComment	(Optional) An optional comment string.
dwNameSpace	See table that follows.
lpNSProviderId	See table that follows.
lpszContext	(Optional) Specifies the starting point of the query in a hierarchical namespace.
dwNumberOfProtocols	Ignored.
lpafpProtocols	Ignored.
lpszQueryString	Ignored.
dwNumberOfCsAddrs	The number of elements in the array of CSADDR_INFO structures referenced by **lpcsaBuffer**.
lpcsaBuffer	A pointer to an array of CSADDR_INFO structures that contain the address(es) that the service is listening on.
lpBlob	(Optional) This is a pointer to a provider-specific entity.

As illustrated in the following, the combination of the **dwNameSpace** and **lpNSProviderId** members determine that namespace providers are affected by this function.

DWNAMESPACE	LPNSPROVIDERID	SCOPE OF IMPACT
Ignored	Non-null	The specified name-space provider.
A valid name- space identifier	Null	All name-space providers that support the indicated namespace.
NS_ALL	Null	All name-space providers.

Windows Phone 8: The **WSASetServiceW** function is supported for Windows Phone Store apps on Windows Phone 8 and later.

Windows 8.1 and **Windows Server 2012 R2**: The **WSASetServiceW** function is supported for Windows Store apps on Windows 8.1, Windows Server 2012 R2, and later.

> **NOTE**
> The winsock2.h header defines WSASetService as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see Conventions for Function Prototypes.

Requirements

Minimum supported client	Windows 8.1, Windows Vista [desktop apps | UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps | UWP apps]
Target Platform	Windows
Header	winsock2.h
Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

Bluetooth and WSASetService

WSAGetLastError

WSAStartup

Winsock Functions

Winsock Reference

WSASetServiceW function (winsock2.h)

10/22/2022 • 5 minutes to read • Edit Online

The **WSASetService** function registers or removes from the registry a service instance within one or more namespaces.

Syntax

```
INT WSAAPI WSASetServiceW(
  [in] LPWSAQUERYSETW   lpqsRegInfo,
  [in] WSAESETSERVICEOP essoperation,
  [in] DWORD            dwControlFlags
);
```

Parameters

`[in] lpqsRegInfo`

A pointer to the service information for registration or deregistration.

`[in] essoperation`

A value that determines that operation requested. This parameter can be one of the values from the WSAESETSERVICEOP enumeration type defined in the *Winsock2.h* header file.

VALUE	MEANING
RNRSERVICE_REGISTER	Register the service. For SAP, this means sending out a periodic broadcast. This is an NOP for the DNS namespace. For persistent data stores, this means updating the address information.
RNRSERVICE_DEREGISTER	Remove the service from the registry. For SAP, this means stop sending out the periodic broadcast. This is an NOP for the DNS namespace. For persistent data stores this means deleting address information.
RNRSERVICE_DELETE	Delete the service from dynamic name and persistent spaces. For services represented by multiple CSADDR_INFO structures (using the SERVICE_MULTIPLE flag), only the specified address will be deleted, and this must match exactly the corresponding **CSADDR_INFO** structure that was specified when the service was registered.

`[in] dwControlFlags`

Service install flags value that further controls the operation performed of the **WSASetService** function. The possible values for this parameter are defined in the *Winsock2.h* header file.

FLAG	MEANING

SERVICE_MULTIPLE	Controls scope of operation. When this flag is not set, service addresses are managed as a group. A register or removal from the registry invalidates all existing addresses before adding the given address set. When set, the action is only performed on the given address set. A register does not invalidate existing addresses and a removal from the registry only invalidates the given set of addresses.

Return value

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

ERROR CODE	MEANING
WSAEACCES	The calling routine does not have sufficient privileges to install the Service.
WSAEINVAL	One or more required parameters were invalid or missing.
WSANOTINITIALISED	The *Ws2_32.dll* has not been initialized. The application must first call WSAStartup before calling any Windows Sockets functions.
WSA_NOT_ENOUGH_MEMORY	There was insufficient memory to perform the operation.

Remarks

The **WSASetService** function can be used to affect a specific namespace provider, all providers associated with a specific namespace, or all providers across all namespaces.

The available values for *essOperation* and *dwControlFlags* combine to control operation of the **WSASetService** function as shown in the following table.

OPERATION	FLAGS	SERVICE ALREADY EXISTS	SERVICE DOES NOT EXIST
RNRSERVICE_REGISTER	None	Overwrites the object. Uses only addresses specified. The object is REGISTERED.	Creates a new object. Uses only addresses specified. Object is REGISTERED.
RNRSERVICE_REGISTER	SERVICE_MULTIPLE	Updates the object. Adds new addresses to the existing set. The object is REGISTERED.	Creates a new object. Uses all addresses specified. Object is REGISTERED.
RNRSERVICE_DEREGISTER	None	Removes all addresses, but does not remove the object from the namespace. The object is removed from the registry.	WSASERVICE_NOT_FOUND

RNRSERVICE_DEREGISTER	SERVICE_MULTIPLE	Updates the object. Removes only addresses that are specified. Only marks the object as DEREGISTERED if no addresses are present. Does not remove the object from the namespace.	WSASERVICE_NOT_FOUND
RNRSERVICE_DELETE	None	Removes the object from the namespace.	WSASERVICE_NOT_FOUND
RNRSERVICE_DELETE	SERVICE_MULTIPLE	Removes only addresses that are specified. Only removes object from the namespace if no addresses remain.	WSASERVICE_NOT_FOUND

Publishing services to directories, such as Active Directory Services, is restricted based on access control lists (ACLs). For more information, see Security Issues for Service Publication.

When the *dwControlFlags* parameter is set to **SERVICE_MULTIPLE**, an application can manage its addresses independently. This is useful when the application wants to manage its protocols individually or when the service resides on more than one computer. For instance, when a service uses more than one protocol, it may find that one listening socket aborts but the other sockets remain operational. In this case, the service could remove the aborted address from the registry without affecting the other addresses.

When the *dwControlFlags* parameter is set to **SERVICE_MULTIPLE**, an application must not let stale addresses remain in the object. This can happen if the application aborts without issuing a DEREGISTER request. When a service registers, it should store its addresses. On its next invocation, the service should explicitly remove these old stale addresses from the registry before registering new addresses.

> **Note** If ANSI character strings are used, there is a chance that the WSAQUERYSET data in *lpqsRegInfo* may not contain any results after this function returns. This is because the ANSI version of this method, **WSASetServiceA**, converts the ANSI data in **WSAQUERYSET** to Unicode internally, but does not convert the results back to ANSI. This primarily impacts transports that return a "service record handle" used to uniquely identify a record. To work around this issue, applications should use Unicode string data in **WSAQUERYSET** when calling this function.

Service Properties

The following table describes how service property data is represented in a WSAQUERYSET structure. Fields labeled as (Optional) can contain a null pointer.

WSAQUERYSET MEMBER	SERVICE PROPERTY DESCRIPTION
dwSize	Must be set to sizeof (WSAQUERYSET). This is a versioning mechanism.
dwOutputFlags	Not applicable and ignored.
lpszServiceInstanceName	Referenced string contains the service instance name.
lpServiceClassId	The GUID corresponding to this service class.

lpVersion	(Optional) Supplies service instance version number.
lpszComment	(Optional) An optional comment string.
dwNameSpace	See table that follows.
lpNSProviderId	See table that follows.
lpszContext	(Optional) Specifies the starting point of the query in a hierarchical namespace.
dwNumberOfProtocols	Ignored.
lpafpProtocols	Ignored.
lpszQueryString	Ignored.
dwNumberOfCsAddrs	The number of elements in the array of CSADDR_INFO structures referenced by **lpcsaBuffer**.
lpcsaBuffer	A pointer to an array of CSADDR_INFO structures that contain the address(es) that the service is listening on.
lpBlob	(Optional) This is a pointer to a provider-specific entity.

As illustrated in the following, the combination of the **dwNameSpace** and **lpNSProviderId** members determine that namespace providers are affected by this function.

DWNAMESPACE	LPNSPROVIDERID	SCOPE OF IMPACT
Ignored	Non-null	The specified name-space provider.
A valid name- space identifier	Null	All name-space providers that support the indicated namespace.
NS_ALL	Null	All name-space providers.

Windows Phone 8: The **WSASetServiceW** function is supported for Windows Phone Store apps on Windows Phone 8 and later.

Windows 8.1 and **Windows Server 2012 R2**: The **WSASetServiceW** function is supported for Windows Store apps on Windows 8.1, Windows Server 2012 R2, and later.

> **NOTE**
> The winsock2.h header defines WSASetService as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see Conventions for Function Prototypes.

Requirements

Minimum supported client	Windows 8.1, Windows Vista [desktop apps | UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps | UWP apps]
Target Platform	Windows
Header	winsock2.h
Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

Bluetooth and WSASetService

WSAGetLastError

WSAStartup

Winsock Functions

Winsock Reference

WSASocketA function (winsock2.h)

10/22/2022 • 21 minutes to read • Edit Online

The **WSASocket** function creates a socket that is bound to a specific transport-service provider.

Syntax

```
SOCKET WSAAPI WSASocketA(
  [in] int                af,
  [in] int                type,
  [in] int                protocol,
  [in] LPWSAPROTOCOL_INFOA lpProtocolInfo,
  [in] GROUP              g,
  [in] DWORD              dwFlags
);
```

Parameters

`[in] af`

The address family specification. Possible values for the address family are defined in the *Winsock2.h* header file.

On the Windows SDK released for Windows Vista and later, the organization of header files has changed and the 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. Note that the values for the AF_ address family and PF_ protocol family constants are identical (for example, **AF_INET** and **PF_INET**), so either constant can be used.

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 including Windows 7, Windows Server 2008, Windows Vista, Windows Server 2003, or Windows XP. The Windows Sockets provider for NetBIOS only supports sockets where the *type* parameter is set to **SOCK_DGRAM**. The Windows Sockets provider for NetBIOS is not directly related to the NetBIOS 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 supported on Windows XP with SP2 or later if the computer has a Bluetooth adapter and driver installed.

`[in] type`

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 IP_HDRINCL socket option must be set on the socket. To manipulate the IPv6 header, the IPV6_HDRINCL 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 reliable multicast programming. This *type* 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.

In Windows Sockets 2, new socket types were introduced. An application can dynamically discover the attributes of each available transport protocol through the WSAEnumProtocols function. So an application can determine the possible socket type and protocol options for an address family and use this information when specifying this parameter. Socket type definitions in the *Winsock2.h* and *Ws2def.h* header files will be periodically updated as new socket types, address families, and protocols are defined.

In Windows Sockets 1.1, the only possible socket types are **SOCK_DGRAM** and **SOCK_STREAM**.

`[in] 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 are defined in the *Winsock2.h* and *Wsrm.h* header files.

On the Windows SDK released for Windows Vista and later,, the organization of header files has changed and this parameter can be one of the values from 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.

When the *af* parameter is AF_INET or AF_INET6 and the *type* is **SOCK_RAW**, the value specified for the *protocol* is set in the protocol field of the IPv6 or IPv4 packet header.

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 *af* parameter is **AF_UNSPEC**, **AF_INET**, or **AF_INET6** and the *type* parameter is **SOCK_RAW** or unspecified. This *protocol* value is supported on Windows XP and later.
IPPROTO_IGMP 2	The Internet Group Management Protocol (IGMP). This is a possible value when the *af* parameter is **AF_UNSPEC**, **AF_INET**, or **AF_INET6** and the *type* parameter is **SOCK_RAW** or unspecified. This *protocol* value is supported on Windows XP and later.
BTHPROTO_RFCOMM 3	The Bluetooth Radio Frequency Communications (Bluetooth RFCOMM) protocol. This is a possible value when the *af* parameter is **AF_BTH** and the *type* parameter is **SOCK_STREAM**. This *protocol* value is supported on Windows XP with SP2 or later.
IPPROTO_TCP 6	The Transmission Control Protocol (TCP). This is a possible value when the *af* parameter is **AF_INET** or **AF_INET6** and the *type* parameter is **SOCK_STREAM**.
IPPROTO_UDP 17	The User Datagram Protocol (UDP). This is a possible value when the *af* parameter is **AF_INET** or **AF_INET6** and the *type* parameter is **SOCK_DGRAM**.
IPPROTO_ICMPV6 58	The Internet Control Message Protocol Version 6 (ICMPv6). This is a possible value when the *af* parameter is **AF_UNSPEC**, **AF_INET**, or **AF_INET6** and the *type* parameter is **SOCK_RAW** or unspecified. This *protocol* value is supported on Windows XP and later.
IPPROTO_RM 113	The PGM protocol for reliable multicast. This is a possible value when the *af* parameter is **AF_INET** and the *type* parameter is **SOCK_RDM**. On the Windows SDK released for Windows Vista and later, this protocol is also called **IPPROTO_PGM**. This *protocol* value is only supported if the Reliable Multicast Protocol is installed.

`[in] lpProtocolInfo`

A pointer to a WSAPROTOCOL_INFO structure that defines the characteristics of the socket to be created. If this parameter is not **NULL**, the socket will be bound to the provider associated with the indicated **WSAPROTOCOL_INFO** structure.

`[in] g`

An existing socket group ID or an appropriate action to take when creating a new socket and a new socket group.

If *g* is an existing socket group ID, join the new socket to this socket group, provided all the requirements set by this group are met.

If *g* is not an existing socket group ID, then the following values are possible.

G	MEANING
0	No group operation is performed.
SG_UNCONSTRAINED_GROUP 0x01	Create an unconstrained socket group and have the new socket be the first member. For an unconstrained group, Winsock does not constrain all sockets in the socket group to have been created with the same value for the *type* and *protocol* parameters.
SG_CONSTRAINED_GROUP 0x02	Create a constrained socket group and have the new socket be the first member. For a constrained socket group, Winsock constrains all sockets in the socket group to have been created with the same value for the *type* and *protocol* parameters. A constrained socket group may consist only of connection-oriented sockets, and requires that connections on all grouped sockets be to the same address on the same host.

> **Note** The SG_UNCONSTRAINED_GROUP and SG_CONSTRAINED_GROUP constants are not currently defined in a public header file.

`[in] dwFlags`

A set of flags used to specify additional socket attributes.

A combination of these flags may be set, although some combinations are not allowed.

VALUE	MEANING
WSA_FLAG_OVERLAPPED 0x01	Create a socket that supports overlapped I/O operations. Most sockets should be created with this flag set. Overlapped sockets can utilize WSASend, WSASendTo, WSARecv, WSARecvFrom, and WSAIoctl for overlapped I/O operations, which allow multiple operations to be initiated and in progress simultaneously. All functions that allow overlapped operation (WSASend, WSARecv, WSASendTo, WSARecvFrom, WSAIoctl) also support nonoverlapped usage on an overlapped socket if the values for parameters related to overlapped operations are **NULL**.

WSA_FLAG_MULTIPOINT_C_ROOT 0x02	Create a socket that will be a c_root in a multipoint session. This attribute is only allowed if the WSAPROTOCOL_INFO structure for the transport provider that creates the socket supports a multipoint or multicast mechanism and the control plane for a multipoint session is rooted. This would be indicated by the **dwServiceFlags1** member of the **WSAPROTOCOL_INFO** structure with the **XP1_SUPPORT_MULTIPOINT** and **XP1_MULTIPOINT_CONTROL_PLANE** flags set. When the *lpProtocolInfo* parameter is not NULL, the WSAPROTOCOL_INFO structure for the transport provider is pointed to by the *lpProtocolInfo* parameter. When the *lpProtocolInfo* parameter is NULL, the **WSAPROTOCOL_INFO** structure is based on the transport provider selected by the values specified for the *af*, *type*, and *protocol* parameters. Refer to Multipoint and Multicast Semantics for additional information on a multipoint session.
WSA_FLAG_MULTIPOINT_C_LEAF 0x04	Create a socket that will be a c_leaf in a multipoint session. This attribute is only allowed if the WSAPROTOCOL_INFO structure for the transport provider that creates the socket supports a multipoint or multicast mechanism and the control plane for a multipoint session is non-rooted. This would be indicated by the **dwServiceFlags1** member of the **WSAPROTOCOL_INFO** structure with the **XP1_SUPPORT_MULTIPOINT** flag set and the **XP1_MULTIPOINT_CONTROL_PLANE** flag not set. When the *lpProtocolInfo* parameter is not NULL, the WSAPROTOCOL_INFO structure for the transport provider is pointed to by the *lpProtocolInfo* parameter. When the *lpProtocolInfo* parameter is NULL, the **WSAPROTOCOL_INFO** structure is based on the transport provider selected by the values specified for the *af*, *type*, and *protocol* parameters. Refer to Multipoint and Multicast Semantics for additional information on a multipoint session.

WSA_FLAG_MULTIPOINT_D_ROOT 0x08	Create a socket that will be a d_root in a multipoint session. This attribute is only allowed if the WSAPROTOCOL_INFO structure for the transport provider that creates the socket supports a multipoint or multicast mechanism and the data plane for a multipoint session is rooted. This would be indicated by the **dwServiceFlags1** member of the **WSAPROTOCOL_INFO** structure with the **XP1_SUPPORT_MULTIPOINT** and **XP1_MULTIPOINT_DATA_PLANE** flags set. When the *lpProtocolInfo* parameter is not NULL, the WSAPROTOCOL_INFO structure for the transport provider is pointed to by the *lpProtocolInfo* parameter. When the *lpProtocolInfo* parameter is NULL, the **WSAPROTOCOL_INFO** structure is based on the transport provider selected by the values specified for the *af*, *type*, and *protocol* parameters. Refer to Multipoint and Multicast Semantics for additional information on a multipoint session.
WSA_FLAG_MULTIPOINT_D_LEAF 0x10	Create a socket that will be a d_leaf in a multipoint session. This attribute is only allowed if the WSAPROTOCOL_INFO structure for the transport provider that creates the socket supports a multipoint or multicast mechanism and the data plane for a multipoint session is non-rooted. This would be indicated by the **dwServiceFlags1** member of the **WSAPROTOCOL_INFO** structure with the **XP1_SUPPORT_MULTIPOINT** flag set and the **XP1_MULTIPOINT_DATA_PLANE** flag not set. When the *lpProtocolInfo* parameter is not NULL, the WSAPROTOCOL_INFO structure for the transport provider is pointed to by the *lpProtocolInfo* parameter. When the *lpProtocolInfo* parameter is NULL, the **WSAPROTOCOL_INFO** structure is based on the transport provider selected by the values specified for the *af*, *type*, and *protocol* parameters. Refer to Multipoint and Multicast Semantics for additional information on a multipoint session.
WSA_FLAG_ACCESS_SYSTEM_SECURITY 0x40	Create a socket that allows the the ability to set a security descriptor on the socket that contains a security access control list (SACL) as opposed to just a discretionary access control list (DACL). SACLs are used for generating audits and alarms when an access check occurs on the object. For a socket, an access check occurs to determine whether the socket should be allowed to bind to a specific address specified to the bind function. The **ACCESS_SYSTEM_SECURITY** access right controls the ability to get or set the SACL in an object's security descriptor. The system grants this access right only if the **SE_SECURITY_NAME** privilege is enabled in the access token of the requesting thread.

WSA_FLAG_NO_HANDLE_INHERIT 0x80	Create a socket that is non-inheritable. A socket handle created by the **WSASocket** or the socket function is inheritable by default. When this flag is set, the socket handle is non-inheritable. The GetHandleInformation function can be used to determine if a socket handle was created with the **WSA_FLAG_NO_HANDLE_INHERIT** flag set. The **GetHandleInformation** function will return that the **HANDLE_FLAG_INHERIT** value is set. This flag is supported on Windows 7 with SP1, Windows Server 2008 R2 with SP1, and later

> **Important** For multipoint sockets, only one of **WSA_FLAG_MULTIPOINT_C_ROOT** or **WSA_FLAG_MULTIPOINT_C_LEAF** flags can be specified, and only one of **WSA_FLAG_MULTIPOINT_D_ROOT** or **WSA_FLAG_MULTIPOINT_D_LEAF** flags can be specified. Refer to Multipoint and Multicast Semantics for additional information.

Return value

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

> **Note** This error code description is Microsoft-specific.

ERROR CODE	MEANING
WSANOTINITIALISED	A successful WSAStartup call must occur before using this function.
WSAENETDOWN	The network subsystem has failed.
WSAEAFNOSUPPORT	The specified address family is not supported.
WSAEFAULT	The *lpProtocolInfo* parameter is not in a valid part of the process address space.
WSAEINPROGRESS	A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function.

WSAEINVAL	This value is true for any of the following conditions. • The parameter *g* specified is not valid. • The WSAPROTOCOL_INFO structure that *lpProtocolInfo* points to is incomplete, the contents are invalid or the **WSAPROTOCOL_INFO** structure has already been used in an earlier duplicate socket operation. • The values specified for members of the socket triple <*af*, *type*, and *protocol*> are individually supported, but the given combination is not.
WSAEINVALIDPROVIDER	The service provider returned a version other than 2.2.
WSAEINVALIDPROCTABLE	The service provider returned an invalid or incomplete procedure table to the WSPStartup.
WSAEMFILE	No more socket descriptors are available.
WSAENOBUFS	No buffer space is available. The socket cannot be created.
WSAEPROTONOSUPPORT	The specified protocol is not supported.
WSAEPROTOTYPE	The specified protocol is the wrong type for this socket.
WSAEPROVIDERFAILEDINIT	The service provider failed to initialize. This error is returned if a layered service provider (LSP) or namespace provider was improperly installed or the provider fails to operate correctly.
WSAESOCKTNOSUPPORT	The specified socket type is not supported in this address family.

Remarks

The **WSASocket** function causes a socket descriptor and any related resources to be allocated and associated with a transport-service provider. Most sockets should be created with the **WSA_FLAG_OVERLAPPED** attribute set in the *dwFlags* parameter. A socket created with this attribute supports the use of overlapped I/O operations which provide higher performance. By default, a socket created with the **WSASocket** function will not have this overlapped attribute set. In contrast, the socket function creates a socket that supports overlapped I/O operations as the default behavior.

If the *lpProtocolInfo* parameter is **NULL**, Winsock will utilize the first available transport-service provider that supports the requested combination of address family, socket type and protocol specified in the *af*, *type*, and *protocol* parameters.

If the *lpProtocolInfo* parameter is not **NULL**, the socket will be bound to the provider associated with the indicated WSAPROTOCOL_INFO structure. In this instance, the application can supply the manifest constant **FROM_PROTOCOL_INFO** as the value for any of *af*, *type*, or *protocol* parameters. This indicates that the corresponding values from the indicated **WSAPROTOCOL_INFO** structure (**iAddressFamily**, **iSocketType**,

iProtocol) are to be assumed. In any case, the values specified for *af*, *type*, and *protocol* are passed unmodified to the transport-service provider.

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

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

Applications are encouraged to use **AF_INET6** for the *af* parameter and create a dual-mode socket that can be used with both IPv4 and IPv6.

If a socket is created using the **WSASocket** function, then the *dwFlags* parameter must have the **WSA_FLAG_OVERLAPPED** attribute set for the **SO_RCVTIMEO** or **SO_SNDTIMEO** socket options to function properly. Otherwise the timeout never takes effect on the socket.

Connection-oriented sockets such as **SOCK_STREAM** provide full-duplex connections, and must be in a connected state before any data can be sent or received on them. A connection to a specified socket is established with a connect or WSAConnect function call. Once connected, data can be transferred using send/WSASend and recv/WSARecv calls. When a session has been completed, the closesocket function should be called to release the resources associated with the socket. For connection-oriented sockets, the shutdown function should be called to stop data transfer on the socket before calling the **closesocket** function.

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

Connectionless, message-oriented sockets allow sending and receiving of datagrams to and from arbitrary peers using sendto/WSASendTo and recvfrom/WSARecvFrom. If such a socket is connected to a specific peer, datagrams can be sent to that peer using send/WSASend and can be received from (only) this peer using recv/WSARecv.

Support for sockets with type **SOCK_RAW** is not required, but service providers are encouraged to support raw sockets whenever possible.

The **WSASocket** function can be used to create a socket to be used by a service so that if another socket tries to bind to the same port used by the service, and audit record is generated. To enable this option, an application would need to do the following:

- Call the AdjustTokenPrivileges function to enable the **SE_SECURITY_NAME** privilege in the access token for the process. This privilege is required to set the **ACCESS_SYSTEM_SECURITY** access rights on the security descriptor for an object.
- Call the **WSASocket** function to create a socket with *dwFlag* with the **WSA_FLAG_ACCESS_SYSTEM_SECURITY** option set. The **WSASocket** function will fail if the AdjustTokenPrivileges function is not called first to enable the **SE_SECURITY_NAME** privilege needed for this operation.
- Call the SetSecurityInfo function to set a security descriptor with a System Access Control List (SACL) on the socket. The socket handle returned by the **WSASocket** function is passed in the *handle* parameter. If the function succeeds, this will set the the **ACCESS_SYSTEM_SECURITY** access right on the security descriptor for the socket.
- Call the bind function to bind the socket to a specific port. If the **bind** function succeeds, then an audit entry

is generated if another socket tries to bind to the same port.

- Call the AdjustTokenPrivileges function to remove the **SE_SECURITY_NAME** privilege in the access token for the process, since this is no longer needed.

For more information on **ACCESS_SYSTEM_SECURITY**, see SACL Access Right and Audit Generation in the Authorization documentation.

Socket Groups

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

Applications that need to exchange multimedia streams over the network are an example where being able to establish a specific relationship among a set of sockets could be beneficial. It is up to the transport on how to treat socket groups.

The **WSASocket** and WSAAccept functions can be used to explicitly create and join a socket group when creating a new socket. The socket group ID for a socket can be retrieved by using the getsockopt function with *level* parameter set to SOL_SOCKET and the *optname* parameter set to **SO_GROUP_ID**. A socket group and its associated socket group ID remain valid until the last socket belonging to this socket group is closed. Socket group IDs are unique across all processes for a given service provider. A socket group of zero indicates that the socket is not member of a socket group.

The relative group priority of a socket group can be accessed by using the getsockopt function with the *level* parameter set to SOL_SOCKET and the *optname* parameter set to **SO_GROUP_PRIORITY**. The relative group priority of a socket group can be set by using setsockopt with the *level* parameter set to SOL_SOCKET and the *optname* parameter set to **SO_GROUP_PRIORITY**.

The Winsock provider included with Windows allows the creation of socket groups and it enforces the SG_CONSTRAINED_GROUP. All sockets in a constrained socket group must be created with the same value for the *type* and *protocol* parameters. A constrained socket group may consist only of connection-oriented sockets, and requires that connections on all grouped sockets be to the same address on the same host. This is the only restriction applied to a socket group by the Winsock provider included with Windows. The socket group priority is not currently used by the Winsock provider or the TCP/IP stack included with Windows.

Example Code

The following example demonstrates the use of the **WSASocket** function.

```
#ifndef UNICODE
#define UNICODE 1
#endif

// link with Ws2_32.lib
#pragma comment(lib,"Ws2_32.lib")

#include <winsock2.h>
#include <ws2tcpip.h>
#include <stdio.h>
#include <stdlib.h>   // Needed for _wtoi


int __cdecl wmain(int argc, wchar_t **argv)
{

    //-----------------------------------------
    // Declare and initialize variables
    WSADATA wsaData = {0};
    int iResult = 0;
```

```
//    int i = 1;

    SOCKET sock = INVALID_SOCKET;
    int iFamily = AF_UNSPEC;
    int iType = 0;
    int iProtocol = 0;
    DWORD dwFlags = 0;

    // Validate the parameters
    if (argc != 5) {
        wprintf(L"usage: %s <addressfamily> <type> <protocol> <flags>\n", argv[0]);
        wprintf(L"       opens a socket for the specified family, type, protocol, and flags\n");
        wprintf(L"       flags value must be in decimal, not hex\n");
        wprintf(L"%ws example usage\n", argv[0]);
        wprintf(L"   %ws 0 2 17 1\n", argv[0]);
        wprintf(L"   where AF_UNSPEC=0 SOCK_DGRAM=2 IPPROTO_UDP=17 OVERLAPPED\n", argv[0]);
        return 1;
    }

    iFamily = _wtoi(argv[1]);
    iType = _wtoi(argv[2]);
    iProtocol = _wtoi(argv[3]);
    dwFlags = _wtoi(argv[4]);

    // Initialize Winsock
    iResult = WSAStartup(MAKEWORD(2, 2), &wsaData);
    if (iResult != 0) {
        wprintf(L"WSAStartup failed: %d\n", iResult);
        return 1;
    }

    wprintf(L"Calling socket with following parameters:\n");
    wprintf(L"  Address Family = ");
    switch (iFamily) {
    case AF_UNSPEC:
        wprintf(L"Unspecified");
        break;
    case AF_INET:
        wprintf(L"AF_INET (IPv4)");
        break;
    case AF_INET6:
        wprintf(L"AF_INET6 (IPv6)");
        break;
    case AF_NETBIOS:
        wprintf(L"AF_NETBIOS (NetBIOS)");
        break;
    case AF_BTH:
        wprintf(L"AF_BTH (Bluetooth)");
        break;
    default:
        wprintf(L"Other");
        break;
    }
    wprintf(L" (%d)\n", iFamily);

    wprintf(L"  Socket type = ");
    switch (iType) {
    case 0:
        wprintf(L"Unspecified");
        break;
    case SOCK_STREAM:
        wprintf(L"SOCK_STREAM (stream)");
        break;
    case SOCK_DGRAM:
        wprintf(L"SOCK_DGRAM (datagram)");
        break;
    case SOCK_RAW:
        wprintf(L"SOCK_RAW (raw)");
        break;
```

```
    case SOCK_RDM:
        wprintf(L"SOCK_RDM (reliable message datagram)");
        break;
    case SOCK_SEQPACKET:
        wprintf(L"SOCK_SEQPACKET (pseudo-stream packet)");
        break;
    default:
        wprintf(L"Other");
        break;
    }
    wprintf(L" (%d)\n", iType);

    wprintf(L"  Protocol = %d = ", iProtocol);
    switch (iProtocol) {
    case 0:
        wprintf(L"Unspecified");
        break;
    case IPPROTO_ICMP:
        wprintf(L"IPPROTO_ICMP (ICMP)");
        break;
    case IPPROTO_IGMP:
        wprintf(L"IPPROTO_IGMP (IGMP)");
        break;
    case IPPROTO_TCP:
        wprintf(L"IPPROTO_TCP (TCP)");
        break;
    case IPPROTO_UDP:
        wprintf(L"IPPROTO_UDP (UDP)");
        break;
    case IPPROTO_ICMPV6:
        wprintf(L"IPPROTO_ICMPV6 (ICMP Version 6)");
        break;
    default:
        wprintf(L"Other");
        break;
    }
    wprintf(L" (%d)\n", iProtocol);

    wprintf(L"  Flags = ");
    if (dwFlags & WSA_FLAG_OVERLAPPED)
        wprintf(L"  WSA_FLAG_OVERLAPPED");
    if (dwFlags & WSA_FLAG_MULTIPOINT_C_ROOT)
        wprintf(L"  WSA_FLAG_MULTIPOINT_C_ROOT");
    if (dwFlags & WSA_FLAG_MULTIPOINT_C_LEAF)
        wprintf(L"  WSA_FLAG_MULTIPOINT_C_LEAF");
    if (dwFlags & WSA_FLAG_MULTIPOINT_D_ROOT)
        wprintf(L"  WSA_FLAG_MULTIPOINT_D_ROOT");
    if (dwFlags & WSA_FLAG_MULTIPOINT_D_LEAF)
        wprintf(L"  WSA_FLAG_MULTIPOINT_D_LEAF");
    if (dwFlags & WSA_FLAG_ACCESS_SYSTEM_SECURITY)
        wprintf(L"  WSA_FLAG_ACCESS_SYSTEM_SECURITY");
#ifdef WSA_FLAG_NO_HANDLE_INHERIT
    if (dwFlags & WSA_FLAG_NO_HANDLE_INHERIT)
        wprintf(L"  WSA_FLAG_NO_HANDLE_INHERIT");
#endif
    wprintf(L" (0x%x)\n" , dwFlags);

    sock = WSASocket(iFamily, iType, iProtocol, NULL, 0, dwFlags);
    if (sock == INVALID_SOCKET)
        wprintf(L"WSASocket function failed with error = %d\n", WSAGetLastError() );
    else {
        wprintf(L"WSASocket function succeeded\n");

        // Close the socket to release the resources associated
        // Normally an application calls shutdown() before closesocket
        //   to  disables sends or receives on a socket first
        // This isn't needed in this simple sample
        iResult = closesocket(sock);
        if (iResult == SOCKET_ERROR) {
```

```
            wprintf(L"closesocket function zfailed with error = %d\n", WSAGetLastError() );
            WSACleanup();
            return 1;
        }
    }
    WSACleanup();

    return 0;
}
```

Windows Phone 8: The **WSASocketW** function is supported for Windows Phone Store apps on Windows Phone 8 and later.

Windows 8.1 and **Windows Server 2012 R2**: The **WSASocketW** function is supported for Windows Store apps on Windows 8.1, Windows Server 2012 R2, and later.

> **NOTE**
> The winsock2.h header defines WSASocket as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see Conventions for Function Prototypes.

Requirements

Minimum supported client	Windows 8.1, Windows Vista [desktop apps | UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps | UWP apps]
Target Platform	Windows
Header	winsock2.h
Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

WSAPROTOCOL_INFO

Winsock Functions

Winsock Reference

accept

bind

closesocket

connect

getsockname

getsockopt

ioctlsocket

listen

recv

recvfrom

select

send

sendto

setsockopt

shutdown

socket

WSASocketW function (winsock2.h)

10/22/2022 • 21 minutes to read • Edit Online

The **WSASocket** function creates a socket that is bound to a specific transport-service provider.

Syntax

```
SOCKET WSAAPI WSASocketW(
  [in] int                af,
  [in] int                type,
  [in] int                protocol,
  [in] LPWSAPROTOCOL_INFOW lpProtocolInfo,
  [in] GROUP              g,
  [in] DWORD              dwFlags
);
```

Parameters

`[in] af`

The address family specification. Possible values for the address family are defined in the *Winsock2.h* header file.

On the Windows SDK released for Windows Vista and later, the organization of header files has changed and the 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. Note that the values for the AF_ address family and PF_ protocol family constants are identical (for example, **AF_INET** and **PF_INET**), so either constant can be used.

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 including Windows 7, Windows Server 2008, Windows Vista, Windows Server 2003, or Windows XP. The Windows Sockets provider for NetBIOS only supports sockets where the *type* parameter is set to **SOCK_DGRAM**. The Windows Sockets provider for NetBIOS is not directly related to the NetBIOS 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 supported on Windows XP with SP2 or later if the computer has a Bluetooth adapter and driver installed.

`[in] type`

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 IP_HDRINCL socket option must be set on the socket. To manipulate the IPv6 header, the IPV6_HDRINCL 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 reliable multicast programming. This *type* 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.

In Windows Sockets 2, new socket types were introduced. An application can dynamically discover the attributes of each available transport protocol through the WSAEnumProtocols function. So an application can determine the possible socket type and protocol options for an address family and use this information when specifying this parameter. Socket type definitions in the *Winsock2.h* and *Ws2def.h* header files will be periodically updated as new socket types, address families, and protocols are defined.

In Windows Sockets 1.1, the only possible socket types are **SOCK_DGRAM** and **SOCK_STREAM**.

`[in] 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 are defined in the *Winsock2.h* and *Wsrm.h* header files.

On the Windows SDK released for Windows Vista and later,, the organization of header files has changed and this parameter can be one of the values from 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.

When the *af* parameter is AF_INET or AF_INET6 and the *type* is **SOCK_RAW**, the value specified for the *protocol* is set in the protocol field of the IPv6 or IPv4 packet header.

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 *af* parameter is **AF_UNSPEC**, **AF_INET**, or **AF_INET6** and the *type* parameter is **SOCK_RAW** or unspecified. This *protocol* value is supported on Windows XP and later.
IPPROTO_IGMP 2	The Internet Group Management Protocol (IGMP). This is a possible value when the *af* parameter is **AF_UNSPEC**, **AF_INET**, or **AF_INET6** and the *type* parameter is **SOCK_RAW** or unspecified. This *protocol* value is supported on Windows XP and later.
BTHPROTO_RFCOMM 3	The Bluetooth Radio Frequency Communications (Bluetooth RFCOMM) protocol. This is a possible value when the *af* parameter is **AF_BTH** and the *type* parameter is **SOCK_STREAM**. This *protocol* value is supported on Windows XP with SP2 or later.
IPPROTO_TCP 6	The Transmission Control Protocol (TCP). This is a possible value when the *af* parameter is **AF_INET** or **AF_INET6** and the *type* parameter is **SOCK_STREAM**.
IPPROTO_UDP 17	The User Datagram Protocol (UDP). This is a possible value when the *af* parameter is **AF_INET** or **AF_INET6** and the *type* parameter is **SOCK_DGRAM**.
IPPROTO_ICMPV6 58	The Internet Control Message Protocol Version 6 (ICMPv6). This is a possible value when the *af* parameter is **AF_UNSPEC**, **AF_INET**, or **AF_INET6** and the *type* parameter is **SOCK_RAW** or unspecified. This *protocol* value is supported on Windows XP and later.
IPPROTO_RM 113	The PGM protocol for reliable multicast. This is a possible value when the *af* parameter is **AF_INET** and the *type* parameter is **SOCK_RDM**. On the Windows SDK released for Windows Vista and later, this protocol is also called **IPPROTO_PGM**. This *protocol* value is only supported if the Reliable Multicast Protocol is installed.

`[in] lpProtocolInfo`

A pointer to a WSAPROTOCOL_INFO structure that defines the characteristics of the socket to be created. If this parameter is not **NULL**, the socket will be bound to the provider associated with the indicated **WSAPROTOCOL_INFO** structure.

`[in] g`

An existing socket group ID or an appropriate action to take when creating a new socket and a new socket group.

If *g* is an existing socket group ID, join the new socket to this socket group, provided all the requirements set by this group are met.

If *g* is not an existing socket group ID, then the following values are possible.

G	MEANING
0	No group operation is performed.
SG_UNCONSTRAINED_GROUP 0x01	Create an unconstrained socket group and have the new socket be the first member. For an unconstrained group, Winsock does not constrain all sockets in the socket group to have been created with the same value for the *type* and *protocol* parameters.
SG_CONSTRAINED_GROUP 0x02	Create a constrained socket group and have the new socket be the first member. For a constrained socket group, Winsock constrains all sockets in the socket group to have been created with the same value for the *type* and *protocol* parameters. A constrained socket group may consist only of connection-oriented sockets, and requires that connections on all grouped sockets be to the same address on the same host.

> **Note** The SG_UNCONSTRAINED_GROUP and SG_CONSTRAINED_GROUP constants are not currently defined in a public header file.

`[in] dwFlags`

A set of flags used to specify additional socket attributes.

A combination of these flags may be set, although some combinations are not allowed.

VALUE	MEANING
WSA_FLAG_OVERLAPPED 0x01	Create a socket that supports overlapped I/O operations. Most sockets should be created with this flag set. Overlapped sockets can utilize WSASend, WSASendTo, WSARecv, WSARecvFrom, and WSAIoctl for overlapped I/O operations, which allow multiple operations to be initiated and in progress simultaneously. All functions that allow overlapped operation (WSASend, WSARecv, WSASendTo, WSARecvFrom, WSAIoctl) also support nonoverlapped usage on an overlapped socket if the values for parameters related to overlapped operations are **NULL**.

WSA_FLAG_MULTIPOINT_C_ROOT 0x02	Create a socket that will be a c_root in a multipoint session. This attribute is only allowed if the WSAPROTOCOL_INFO structure for the transport provider that creates the socket supports a multipoint or multicast mechanism and the control plane for a multipoint session is rooted. This would be indicated by the **dwServiceFlags1** member of the **WSAPROTOCOL_INFO** structure with the **XP1_SUPPORT_MULTIPOINT** and **XP1_MULTIPOINT_CONTROL_PLANE** flags set. When the *lpProtocolInfo* parameter is not NULL, the WSAPROTOCOL_INFO structure for the transport provider is pointed to by the *lpProtocolInfo* parameter. When the *lpProtocolInfo* parameter is NULL, the **WSAPROTOCOL_INFO** structure is based on the transport provider selected by the values specified for the *af*, *type*, and *protocol* parameters. Refer to Multipoint and Multicast Semantics for additional information on a multipoint session.
WSA_FLAG_MULTIPOINT_C_LEAF 0x04	Create a socket that will be a c_leaf in a multipoint session. This attribute is only allowed if the WSAPROTOCOL_INFO structure for the transport provider that creates the socket supports a multipoint or multicast mechanism and the control plane for a multipoint session is non-rooted. This would be indicated by the **dwServiceFlags1** member of the **WSAPROTOCOL_INFO** structure with the **XP1_SUPPORT_MULTIPOINT** flag set and the **XP1_MULTIPOINT_CONTROL_PLANE** flag not set. When the *lpProtocolInfo* parameter is not NULL, the WSAPROTOCOL_INFO structure for the transport provider is pointed to by the *lpProtocolInfo* parameter. When the *lpProtocolInfo* parameter is NULL, the **WSAPROTOCOL_INFO** structure is based on the transport provider selected by the values specified for the *af*, *type*, and *protocol* parameters. Refer to Multipoint and Multicast Semantics for additional information on a multipoint session.

WSA_FLAG_MULTIPOINT_D_ROOT 0x08	Create a socket that will be a d_root in a multipoint session. This attribute is only allowed if the WSAPROTOCOL_INFO structure for the transport provider that creates the socket supports a multipoint or multicast mechanism and the data plane for a multipoint session is rooted. This would be indicated by the **dwServiceFlags1** member of the **WSAPROTOCOL_INFO** structure with the **XP1_SUPPORT_MULTIPOINT** and **XP1_MULTIPOINT_DATA_PLANE** flags set. When the *lpProtocolInfo* parameter is not NULL, the WSAPROTOCOL_INFO structure for the transport provider is pointed to by the *lpProtocolInfo* parameter. When the *lpProtocolInfo* parameter is NULL, the **WSAPROTOCOL_INFO** structure is based on the transport provider selected by the values specified for the *af*, *type*, and *protocol* parameters. Refer to Multipoint and Multicast Semantics for additional information on a multipoint session.
WSA_FLAG_MULTIPOINT_D_LEAF 0x10	Create a socket that will be a d_leaf in a multipoint session. This attribute is only allowed if the WSAPROTOCOL_INFO structure for the transport provider that creates the socket supports a multipoint or multicast mechanism and the data plane for a multipoint session is non-rooted. This would be indicated by the **dwServiceFlags1** member of the **WSAPROTOCOL_INFO** structure with the **XP1_SUPPORT_MULTIPOINT** flag set and the **XP1_MULTIPOINT_DATA_PLANE** flag not set. When the *lpProtocolInfo* parameter is not NULL, the WSAPROTOCOL_INFO structure for the transport provider is pointed to by the *lpProtocolInfo* parameter. When the *lpProtocolInfo* parameter is NULL, the **WSAPROTOCOL_INFO** structure is based on the transport provider selected by the values specified for the *af*, *type*, and *protocol* parameters. Refer to Multipoint and Multicast Semantics for additional information on a multipoint session.
WSA_FLAG_ACCESS_SYSTEM_SECURITY 0x40	Create a socket that allows the the ability to set a security descriptor on the socket that contains a security access control list (SACL) as opposed to just a discretionary access control list (DACL). SACLs are used for generating audits and alarms when an access check occurs on the object. For a socket, an access check occurs to determine whether the socket should be allowed to bind to a specific address specified to the bind function. The **ACCESS_SYSTEM_SECURITY** access right controls the ability to get or set the SACL in an object's security descriptor. The system grants this access right only if the **SE_SECURITY_NAME** privilege is enabled in the access token of the requesting thread.

WSA_FLAG_NO_HANDLE_INHERIT 0x80	Create a socket that is non-inheritable. A socket handle created by the **WSASocket** or the socket function is inheritable by default. When this flag is set, the socket handle is non-inheritable. The GetHandleInformation function can be used to determine if a socket handle was created with the **WSA_FLAG_NO_HANDLE_INHERIT** flag set. The **GetHandleInformation** function will return that the **HANDLE_FLAG_INHERIT** value is set. This flag is supported on Windows 7 with SP1, Windows Server 2008 R2 with SP1, and later

> **Important** For multipoint sockets, only one of **WSA_FLAG_MULTIPOINT_C_ROOT** or **WSA_FLAG_MULTIPOINT_C_LEAF** flags can be specified, and only one of **WSA_FLAG_MULTIPOINT_D_ROOT** or **WSA_FLAG_MULTIPOINT_D_LEAF** flags can be specified. Refer to Multipoint and Multicast Semantics for additional information.

Return value

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

> **Note** This error code description is Microsoft-specific.

ERROR CODE	MEANING
WSANOTINITIALISED	A successful WSAStartup call must occur before using this function.
WSAENETDOWN	The network subsystem has failed.
WSAEAFNOSUPPORT	The specified address family is not supported.
WSAEFAULT	The *lpProtocolInfo* parameter is not in a valid part of the process address space.
WSAEINPROGRESS	A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function.

WSAEINVAL	This value is true for any of the following conditions. • The parameter *g* specified is not valid. • The WSAPROTOCOL_INFO structure that *lpProtocolInfo* points to is incomplete, the contents are invalid or the **WSAPROTOCOL_INFO** structure has already been used in an earlier duplicate socket operation. • The values specified for members of the socket triple <*af*, *type*, and *protocol*> are individually supported, but the given combination is not.
WSAEINVALIDPROVIDER	The service provider returned a version other than 2.2.
WSAEINVALIDPROCTABLE	The service provider returned an invalid or incomplete procedure table to the WSPStartup.
WSAEMFILE	No more socket descriptors are available.
WSAENOBUFS	No buffer space is available. The socket cannot be created.
WSAEPROTONOSUPPORT	The specified protocol is not supported.
WSAEPROTOTYPE	The specified protocol is the wrong type for this socket.
WSAEPROVIDERFAILEDINIT	The service provider failed to initialize. This error is returned if a layered service provider (LSP) or namespace provider was improperly installed or the provider fails to operate correctly.
WSAESOCKTNOSUPPORT	The specified socket type is not supported in this address family.

Remarks

The **WSASocket** function causes a socket descriptor and any related resources to be allocated and associated with a transport-service provider. Most sockets should be created with the **WSA_FLAG_OVERLAPPED** attribute set in the *dwFlags* parameter. A socket created with this attribute supports the use of overlapped I/O operations which provide higher performance. By default, a socket created with the **WSASocket** function will not have this overlapped attribute set. In contrast, the socket function creates a socket that supports overlapped I/O operations as the default behavior.

If the *lpProtocolInfo* parameter is **NULL**, Winsock will utilize the first available transport-service provider that supports the requested combination of address family, socket type and protocol specified in the *af*, *type*, and *protocol* parameters.

If the *lpProtocolInfo* parameter is not **NULL**, the socket will be bound to the provider associated with the indicated WSAPROTOCOL_INFO structure. In this instance, the application can supply the manifest constant **FROM_PROTOCOL_INFO** as the value for any of *af*, *type*, or *protocol* parameters. This indicates that the corresponding values from the indicated **WSAPROTOCOL_INFO** structure (**iAddressFamily**, **iSocketType**,

iProtocol) are to be assumed. In any case, the values specified for *af*, *type*, and *protocol* are passed unmodified to the transport-service provider.

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

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

Applications are encouraged to use **AF_INET6** for the *af* parameter and create a dual-mode socket that can be used with both IPv4 and IPv6.

If a socket is created using the **WSASocket** function, then the *dwFlags* parameter must have the **WSA_FLAG_OVERLAPPED** attribute set for the **SO_RCVTIMEO** or **SO_SNDTIMEO** socket options to function properly. Otherwise the timeout never takes effect on the socket.

Connection-oriented sockets such as **SOCK_STREAM** provide full-duplex connections, and must be in a connected state before any data can be sent or received on them. A connection to a specified socket is established with a connect or WSAConnect function call. Once connected, data can be transferred using send/WSASend and recv/WSARecv calls. When a session has been completed, the closesocket function should be called to release the resources associated with the socket. For connection-oriented sockets, the shutdown function should be called to stop data transfer on the socket before calling the **closesocket** function.

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

Connectionless, message-oriented sockets allow sending and receiving of datagrams to and from arbitrary peers using sendto/WSASendTo and recvfrom/WSARecvFrom. If such a socket is connected to a specific peer, datagrams can be sent to that peer using send/WSASend and can be received from (only) this peer using recv/WSARecv.

Support for sockets with type **SOCK_RAW** is not required, but service providers are encouraged to support raw sockets whenever possible.

The **WSASocket** function can be used to create a socket to be used by a service so that if another socket tries to bind to the same port used by the service, and audit record is generated. To enable this option, an application would need to do the following:

- Call the AdjustTokenPrivileges function to enable the **SE_SECURITY_NAME** privilege in the access token for the process. This privilege is required to set the **ACCESS_SYSTEM_SECURITY** access rights on the security descriptor for an object.
- Call the **WSASocket** function to create a socket with *dwFlag* with the **WSA_FLAG_ACCESS_SYSTEM_SECURITY** option set. The **WSASocket** function will fail if the AdjustTokenPrivileges function is not called first to enable the **SE_SECURITY_NAME** privilege needed for this operation.
- Call the SetSecurityInfo function to set a security descriptor with a System Access Control List (SACL) on the socket. The socket handle returned by the **WSASocket** function is passed in the *handle* parameter. If the function succeeds, this will set the the **ACCESS_SYSTEM_SECURITY** access right on the security descriptor for the socket.
- Call the bind function to bind the socket to a specific port. If the **bind** function succeeds, then an audit entry

is generated if another socket tries to bind to the same port.

- Call the AdjustTokenPrivileges function to remove the **SE_SECURITY_NAME** privilege in the access token for the process, since this is no longer needed.

For more information on **ACCESS_SYSTEM_SECURITY**, see SACL Access Right and Audit Generation in the Authorization documentation.

Socket Groups

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

Applications that need to exchange multimedia streams over the network are an example where being able to establish a specific relationship among a set of sockets could be beneficial. It is up to the transport on how to treat socket groups.

The **WSASocket** and WSAAccept functions can be used to explicitly create and join a socket group when creating a new socket. The socket group ID for a socket can be retrieved by using the getsockopt function with *level* parameter set to SOL_SOCKET and the *optname* parameter set to **SO_GROUP_ID**. A socket group and its associated socket group ID remain valid until the last socket belonging to this socket group is closed. Socket group IDs are unique across all processes for a given service provider. A socket group of zero indicates that the socket is not member of a socket group.

The relative group priority of a socket group can be accessed by using the getsockopt function with the *level* parameter set to SOL_SOCKET and the *optname* parameter set to **SO_GROUP_PRIORITY**. The relative group priority of a socket group can be set by using setsockopt with the *level* parameter set to SOL_SOCKET and the *optname* parameter set to **SO_GROUP_PRIORITY**.

The Winsock provider included with Windows allows the creation of socket groups and it enforces the SG_CONSTRAINED_GROUP. All sockets in a constrained socket group must be created with the same value for the *type* and *protocol* parameters. A constrained socket group may consist only of connection-oriented sockets, and requires that connections on all grouped sockets be to the same address on the same host. This is the only restriction applied to a socket group by the Winsock provider included with Windows. The socket group priority is not currently used by the Winsock provider or the TCP/IP stack included with Windows.

Example Code

The following example demonstrates the use of the **WSASocket** function.

```
#ifndef UNICODE
#define UNICODE 1
#endif

// link with Ws2_32.lib
#pragma comment(lib,"Ws2_32.lib")

#include <winsock2.h>
#include <ws2tcpip.h>
#include <stdio.h>
#include <stdlib.h>   // Needed for _wtoi


int __cdecl wmain(int argc, wchar_t **argv)
{

    //-----------------------------------------
    // Declare and initialize variables
    WSADATA wsaData = {0};
    int iResult = 0;
```

```
//    int i = 1;

    SOCKET sock = INVALID_SOCKET;
    int iFamily = AF_UNSPEC;
    int iType = 0;
    int iProtocol = 0;
    DWORD dwFlags = 0;

    // Validate the parameters
    if (argc != 5) {
        wprintf(L"usage: %s <addressfamily> <type> <protocol> <flags>\n", argv[0]);
        wprintf(L"       opens a socket for the specified family, type, protocol, and flags\n");
        wprintf(L"       flags value must be in decimal, not hex\n");
        wprintf(L"%ws example usage\n", argv[0]);
        wprintf(L"   %ws 0 2 17 1\n", argv[0]);
        wprintf(L"   where AF_UNSPEC=0 SOCK_DGRAM=2 IPPROTO_UDP=17 OVERLAPPED\n", argv[0]);
        return 1;
    }

    iFamily = _wtoi(argv[1]);
    iType = _wtoi(argv[2]);
    iProtocol = _wtoi(argv[3]);
    dwFlags = _wtoi(argv[4]);

    // Initialize Winsock
    iResult = WSAStartup(MAKEWORD(2, 2), &wsaData);
    if (iResult != 0) {
        wprintf(L"WSAStartup failed: %d\n", iResult);
        return 1;
    }

    wprintf(L"Calling socket with following parameters:\n");
    wprintf(L"  Address Family = ");
    switch (iFamily) {
    case AF_UNSPEC:
        wprintf(L"Unspecified");
        break;
    case AF_INET:
        wprintf(L"AF_INET (IPv4)");
        break;
    case AF_INET6:
        wprintf(L"AF_INET6 (IPv6)");
        break;
    case AF_NETBIOS:
        wprintf(L"AF_NETBIOS (NetBIOS)");
        break;
    case AF_BTH:
        wprintf(L"AF_BTH (Bluetooth)");
        break;
    default:
        wprintf(L"Other");
        break;
    }
    wprintf(L" (%d)\n", iFamily);

    wprintf(L"  Socket type = ");
    switch (iType) {
    case 0:
        wprintf(L"Unspecified");
        break;
    case SOCK_STREAM:
        wprintf(L"SOCK_STREAM (stream)");
        break;
    case SOCK_DGRAM:
        wprintf(L"SOCK_DGRAM (datagram)");
        break;
    case SOCK_RAW:
        wprintf(L"SOCK_RAW (raw)");
        break;
```

```
    case SOCK_RDM:
        wprintf(L"SOCK_RDM (reliable message datagram)");
        break;
    case SOCK_SEQPACKET:
        wprintf(L"SOCK_SEQPACKET (pseudo-stream packet)");
        break;
    default:
        wprintf(L"Other");
        break;
    }
    wprintf(L" (%d)\n", iType);

    wprintf(L"  Protocol = %d = ", iProtocol);
    switch (iProtocol) {
    case 0:
        wprintf(L"Unspecified");
        break;
    case IPPROTO_ICMP:
        wprintf(L"IPPROTO_ICMP (ICMP)");
        break;
    case IPPROTO_IGMP:
        wprintf(L"IPPROTO_IGMP (IGMP)");
        break;
    case IPPROTO_TCP:
        wprintf(L"IPPROTO_TCP (TCP)");
        break;
    case IPPROTO_UDP:
        wprintf(L"IPPROTO_UDP (UDP)");
        break;
    case IPPROTO_ICMPV6:
        wprintf(L"IPPROTO_ICMPV6 (ICMP Version 6)");
        break;
    default:
        wprintf(L"Other");
        break;
    }
    wprintf(L" (%d)\n", iProtocol);

    wprintf(L"  Flags = ");
    if (dwFlags & WSA_FLAG_OVERLAPPED)
        wprintf(L"  WSA_FLAG_OVERLAPPED");
    if (dwFlags & WSA_FLAG_MULTIPOINT_C_ROOT)
        wprintf(L"  WSA_FLAG_MULTIPOINT_C_ROOT");
    if (dwFlags & WSA_FLAG_MULTIPOINT_C_LEAF)
        wprintf(L"  WSA_FLAG_MULTIPOINT_C_LEAF");
    if (dwFlags & WSA_FLAG_MULTIPOINT_D_ROOT)
        wprintf(L"  WSA_FLAG_MULTIPOINT_D_ROOT");
    if (dwFlags & WSA_FLAG_MULTIPOINT_D_LEAF)
        wprintf(L"  WSA_FLAG_MULTIPOINT_D_LEAF");
    if (dwFlags & WSA_FLAG_ACCESS_SYSTEM_SECURITY)
        wprintf(L"  WSA_FLAG_ACCESS_SYSTEM_SECURITY");
#ifdef WSA_FLAG_NO_HANDLE_INHERIT
    if (dwFlags & WSA_FLAG_NO_HANDLE_INHERIT)
        wprintf(L"  WSA_FLAG_NO_HANDLE_INHERIT");
#endif
    wprintf(L" (0x%x)\n" , dwFlags);

    sock = WSASocket(iFamily, iType, iProtocol, NULL, 0, dwFlags);
    if (sock == INVALID_SOCKET)
        wprintf(L"WSASocket function failed with error = %d\n", WSAGetLastError() );
    else {
        wprintf(L"WSASocket function succeeded\n");

        // Close the socket to release the resources associated
        // Normally an application calls shutdown() before closesocket
        //   to  disables sends or receives on a socket first
        // This isn't needed in this simple sample
        iResult = closesocket(sock);
        if (iResult == SOCKET_ERROR) {
```

```
            wprintf(L"closesocket function zfailed with error = %d\n", WSAGetLastError() );
            WSACleanup();
            return 1;
        }
    }
    WSACleanup();

    return 0;
}
```

Windows Phone 8: The **WSASocketW** function is supported for Windows Phone Store apps on Windows Phone 8 and later.

Windows 8.1 and **Windows Server 2012 R2**: The **WSASocketW** function is supported for Windows Store apps on Windows 8.1, Windows Server 2012 R2, and later.

> **NOTE**
> The winsock2.h header defines WSASocket as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see Conventions for Function Prototypes.

Requirements

Minimum supported client	Windows 8.1, Windows Vista [desktop apps | UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps | UWP apps]
Target Platform	Windows
Header	winsock2.h
Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

WSAPROTOCOL_INFO

Winsock Functions

Winsock Reference

accept

bind

closesocket

connect

getsockname

getsockopt

ioctlsocket

listen

recv

recvfrom

select

send

sendto

setsockopt

shutdown

socket

WSAStartup function (winsock2.h)

10/22/2022 • 7 minutes to read • Edit Online

The **WSAStartup** function initiates use of the Winsock DLL by a process.

Syntax

```
int WSAAPI WSAStartup(
  [in]  WORD      wVersionRequested,
  [out] LPWSADATA lpWSAData
);
```

Parameters

`[in] wVersionRequested`

The highest version of Windows Sockets specification that the caller can use. The high-order byte specifies the minor version number; the low-order byte specifies the major version number.

`[out] lpWSAData`

A pointer to the WSADATA data structure that is to receive details of the Windows Sockets implementation.

Return value

If successful, the **WSAStartup** function returns zero. Otherwise, it returns one of the error codes listed below.

The **WSAStartup** function directly returns the extended error code in the return value for this function. A call to the WSAGetLastError function is not needed and should not be used.

ERROR CODE	MEANING
WSASYSNOTREADY	The underlying network subsystem is not ready for network communication.
WSAVERNOTSUPPORTED	The version of Windows Sockets support requested is not provided by this particular Windows Sockets implementation.
WSAEINPROGRESS	A blocking Windows Sockets 1.1 operation is in progress.
WSAEPROCLIM	A limit on the number of tasks supported by the Windows Sockets implementation has been reached.
WSAEFAULT	The *lpWSAData* parameter is not a valid pointer.

Remarks

The **WSAStartup** function must be the first Windows Sockets function called by an application or DLL. It allows an application or DLL to specify the version of Windows Sockets required and retrieve details of the specific Windows Sockets implementation. The application or DLL can only issue further Windows Sockets functions after successfully calling **WSAStartup**.

In order to support various Windows Sockets implementations and applications that can have functional differences from the latest version of Windows Sockets specification, a negotiation takes place in **WSAStartup**. The caller of **WSAStartup** passes in the *wVersionRequested* parameter the highest version of the Windows Sockets specification that the application supports. The Winsock DLL indicates the highest version of the Windows Sockets specification that it can support in its response. The Winsock DLL also replies with version of the Windows Sockets specification that it expects the caller to use.

When an application or DLL calls the **WSAStartup** function, the Winsock DLL examines the version of the Windows Sockets specification requested by the application passed in the *wVersionRequested* parameter. If the version requested by the application is equal to or higher than the lowest version supported by the Winsock DLL, the call succeeds and the Winsock DLL returns detailed information in the WSADATA structure pointed to by the *lpWSAData* parameter. The **wHighVersion** member of the **WSADATA** structure indicates the highest version of the Windows Sockets specification that the Winsock DLL supports. The **wVersion** member of the **WSADATA** structure indicates the version of the Windows Sockets specification that the Winsock DLL expects the caller to use.

If the **wVersion** member of the WSADATA structure is unacceptable to the caller, the application or DLL should call WSACleanup to release the Winsock DLL resources and fail to initialize the Winsock application. In order to support this application or DLL, it will be necessary to search for an updated version of the Winsock DLL to install on the platform.

The current version of the Windows Sockets specification is version 2.2. The current Winsock DLL, *Ws2_32.dll*, supports applications that request any of the following versions of Windows Sockets specification:

- 1.0
- 1.1
- 2.0
- 2.1
- 2.2

To get full access to the new syntax of a higher version of the Windows Sockets specification, the application must negotiate for this higher version. In this case, the *wVersionRequested* parameter should be set to request version 2.2. The application must also fully conform to that higher version of the Windows Socket specification, such as compiling against the appropriate header file, linking with a new library, or other special cases. The *Winsock2.h header* file for Winsock 2 support is included with the Microsoft Windows Software Development Kit (SDK).

Windows Sockets version 2.2 is supported on Windows Server 2008, Windows Vista, Windows Server 2003, Windows XP, Windows 2000, Windows NT 4.0 with Service Pack 4 (SP4) and later, Windows Me, Windows 98, and Windows 95 OSR2. Windows Sockets version 2.2 is also supported on
Windows 95 with the Windows Socket 2 Update. Applications on these platforms should normally request Winsock 2.2 by setting the *wVersionRequested* parameter accordingly.

On Windows 95 and versions of Windows NT 3.51 and earlier, Windows Sockets version 1.1 is the highest version of the Windows Sockets specification supported.

It is legal and possible for an application or DLL written to use a lower version of the Windows Sockets specification that is supported by the Winsock DLL to successfully negotiate this lower version using the **WSAStartup** function. For example, an application can request version 1.1 in the *wVersionRequested* parameter passed to the **WSAStartup** function on a platform with the Winsock 2.2 DLL. In this case, the

application should only rely on features that fit within the version requested. New Ioctl codes, new behavior of existing functions, and new functions should not be used. The version negotiation provided by the **WSAStartup** was primarily used to allow older Winsock 1.1 applications developed for Windows 95 and Windows NT 3.51 and earlier to run with the same behavior on later versions of Windows. The *Winsock.h header* file for Winsock 1.1 support is included with the Windows SDK.

This negotiation in the **WSAStartup** function allows both the application or DLL that uses Windows Sockets and the Winsock DLL to support a range of Windows Sockets versions. An application or DLL can use the Winsock DLL if there is any overlap in the version ranges. Detailed information on the Windows Sockets implementation is provided in the WSADATA structure returned by the **WSAStartup** function.

The following table shows how **WSAStartup** works with different applications and Winsock DLL versions.

CALLER VERSION SUPPORT	WINSOCK DLL VERSION SUPPORT	WVERSION REQUESTED	WVERSION RETURNED	WHIGHVERSION RETURNED	END RESULT
1.1	1.1	1.1	1.1	1.1	use 1.1
1.0 1.1	1.0	1.1	1.0	1.0	use 1.0
1.0	1.0 1.1	1.0	1.0	1.1	use 1.0
1.1	1.0 1.1	1.1	1.1	1.1	use 1.1
1.1	1.0	1.1	1.0	1.0	Application fails
1.0	1.1	1.0	—	—	WSAVERNOTSUPPORTED
1.0 1.1	1.0 1.1	1.1	1.1	1.1	use 1.1
1.1 2.0	1.0 1.1	2.0	1.1	1.1	use 1.1
2.0	1.0 1.1 2.0	2.0	2.0	2.0	use 2.0
2.0 2.2	1.0 1.1 2.0	2.2	2.0	2.0	use 2.0
2.2	1.0 1.1 2.0 2.1 2.2	2.2	2.2	2.2	use 2.2

Once an application or DLL has made a successful **WSAStartup** call, it can proceed to make other Windows Sockets calls as needed. When it has finished using the services of the Winsock DLL, the application must call WSACleanup to allow the Winsock DLL to free internal Winsock resources used by the application.

An application can call **WSAStartup** more than once if it needs to obtain the WSADATA structure information more than once. On each such call, the application can specify any version number supported by the Winsock DLL.

The **WSAStartup** function typically leads to protocol-specific helper DLLs being loaded. As a result, the **WSAStartup** function should not be called from the DllMain function in a application DLL. This can potentially cause deadlocks. For more information, please see the DLL Main Function.

An application must call the WSACleanup function for every successful time the **WSAStartup** function is called. This means, for example, that if an application calls **WSAStartup** three times, it must call **WSACleanup** three

times. The first two calls to **WSACleanup** do nothing except decrement an internal counter; the final **WSACleanup** call for the task does all necessary resource deallocation for the task.

> **Note** An application can call the WSAGetLastError function to determine the extended error code for other Windows sockets functions as is normally done in Windows Sockets even if the **WSAStartup** function fails or the **WSAStartup** function was not called to properly initialize Windows Sockets before calling a Windows Sockets function. The **WSAGetLastError** function is one of the only functions in the Winsock 2.2 DLL that can be called in the case of a **WSAStartup** failure.

Windows Phone 8: This function is supported for Windows Phone Store apps on Windows Phone 8 and later.

Windows 8.1 and **Windows Server 2012 R2**: This function is supported for Windows Store apps on Windows 8.1, Windows Server 2012 R2, and later.

Examples

The following code fragment demonstrates how an application that supports only version 2.2 of Windows Sockets makes a **WSAStartup** call:

```
#define WIN32_LEAN_AND_MEAN

#include <windows.h>
#include <winsock2.h>
#include <ws2tcpip.h>
#include <stdio.h>

// Need to link with Ws2_32.lib
#pragma comment(lib, "ws2_32.lib")


int __cdecl main()
{

    WORD wVersionRequested;
    WSADATA wsaData;
    int err;

/* Use the MAKEWORD(lowbyte, highbyte) macro declared in Windef.h */
    wVersionRequested = MAKEWORD(2, 2);

    err = WSAStartup(wVersionRequested, &wsaData);
    if (err != 0) {
        /* Tell the user that we could not find a usable */
        /* Winsock DLL.                                  */
        printf("WSAStartup failed with error: %d\n", err);
        return 1;
    }

/* Confirm that the WinSock DLL supports 2.2.*/
/* Note that if the DLL supports versions greater    */
/* than 2.2 in addition to 2.2, it will still return */
/* 2.2 in wVersion since that is the version we      */
/* requested.                                        */

    if (LOBYTE(wsaData.wVersion) != 2 || HIBYTE(wsaData.wVersion) != 2) {
        /* Tell the user that we could not find a usable */
        /* WinSock DLL.                                  */
        printf("Could not find a usable version of Winsock.dll\n");
        WSACleanup();
        return 1;
    }
    else
        printf("The Winsock 2.2 dll was found okay\n");


/* The Winsock DLL is acceptable. Proceed to use it. */

/* Add network programming using Winsock here */

/* then call WSACleanup when done using the Winsock dll */

    WSACleanup();

}
```

Requirements

Minimum supported client	Windows 8.1, Windows Vista [desktop apps | UWP apps]

Minimum supported server	Windows Server 2003 [desktop apps | UWP apps]
Target Platform	Windows
Header	winsock2.h (include Winsock2.h)
Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

MAKEWORD

WSACleanup

WSAGetLastError

Winsock Functions

Winsock Reference

send

sendto

WSAStringToAddressA function (winsock2.h)

10/22/2022 • 2 minutes to read • Edit Online

The **WSAStringToAddress** function converts a network address in its standard text presentation form into its numeric binary form in a sockaddr structure, suitable for passing to Windows Sockets routines that take such a structure.

Syntax

```
INT WSAAPI WSAStringToAddressA(
  [in]           LPSTR               AddressString,
  [in]           INT                 AddressFamily,
  [in, optional] LPWSAPROTOCOL_INFOA lpProtocolInfo,
  [out]          LPSOCKADDR          lpAddress,
  [in, out]      LPINT               lpAddressLength
);
```

Parameters

`[in] AddressString`

A pointer to the zero-terminated string that contains the network address in standard text form to convert.

`[in] AddressFamily`

The address family of the network address pointed to by the *AddressString* parameter.

`[in, optional] lpProtocolInfo`

The WSAPROTOCOL_INFO structure associated with the provider to be used. If this is **NULL**, the call is routed to the provider of the first protocol supporting the indicated *AddressFamily*.

`[out] lpAddress`

A pointer to a buffer that is filled with a sockaddr structure for the address string if the function succeeds.

`[in, out] lpAddressLength`

A pointer to the length, in bytes, of the buffer pointed to by the *lpAddress* parameter. If the function call is successful, this parameter returns a pointer to the size of the sockaddr structure returned in the *lpAddress* parameter. If the specified buffer is not large enough, the function fails with a specific error of WSAEFAULT and this parameter is updated with the required size in bytes.

Return value

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

ERROR CODE	MEANING
WSAEFAULT	The buffer pointed to by the *lpAddress* parameter is too small. Pass in a larger buffer.

WSAEINVAL	The functions was unable to translate the string into a sockaddr. See the following Remarks section for more information.
WSANOTINITIALISED	The WS2_32.DLL has not been initialized. The application must first call WSAStartup before calling any Windows Socket functions.
WSA_NOT_ENOUGH_MEMORY	There was insufficient memory to perform the operation.

Remarks

The **WSAStringToAddress** function converts a network address in standard text form into its numeric binary form in a sockaddr structure.

Any missing components of the address will be defaulted to a reasonable value, if possible. For example, a missing port number will default to zero. If the caller wants the translation to be done by a particular provider, it should supply the corresponding WSAPROTOCOL_INFO structure in the *lpProtocolInfo* parameter.

The **WSAStringToAddress** function fails (and returns WSAEINVAL) if the **sin_family** member of the SOCKADDR_IN structure, which is passed in the *lpAddress* parameter in the form of a **sockaddr** structure, is not set to AF_INET or AF_INET6.

Support for IPv6 addresses using the **WSAStringToAddress** function was added on Windows XP with Service Pack 1 (SP1)and later. IPv6 must also be installed on the local computer for the **WSAStringToAddress** function to support IPv6 addresses.

Windows Phone 8: This function is supported for Windows Phone Store apps on Windows Phone 8 and later.

Windows 8.1 and **Windows Server 2012 R2**: This function is supported for Windows Store apps on Windows 8.1, Windows Server 2012 R2, and later.

> **NOTE**
> The winsock2.h header defines WSAStringToAddress as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see Conventions for Function Prototypes.

Requirements

Minimum supported client	Windows 8.1, Windows Vista [desktop apps | UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps | UWP apps]
Target Platform	Windows
Header	winsock2.h
Library	Ws2_32.lib

DLL	Ws2_32.dll

See also

InetNtop

InetPton

RtlIpv4AddressToString

RtlIpv4AddressToStringEx

RtlIpv4StringToAddress

RtlIpv4StringToAddressEx

RtlIpv6AddressToString

RtlIpv6AddressToStringEx

RtlIpv6StringToAddress

RtlIpv6StringToAddressEx

WSAAddressToString

WSAPROTOCOL_INFO

WSAStartup

inet_addr

inet_ntoa

sockaddr

WSAStringToAddressW function (winsock2.h)

10/22/2022 • 2 minutes to read • Edit Online

The **WSAStringToAddress** function converts a network address in its standard text presentation form into its numeric binary form in a sockaddr structure, suitable for passing to Windows Sockets routines that take such a structure.

Syntax

```
INT WSAAPI WSAStringToAddressW(
  [in]           LPWSTR              AddressString,
  [in]           INT                 AddressFamily,
  [in, optional] LPWSAPROTOCOL_INFOW lpProtocolInfo,
  [out]          LPSOCKADDR          lpAddress,
  [in, out]      LPINT               lpAddressLength
);
```

Parameters

`[in] AddressString`

A pointer to the zero-terminated string that contains the network address in standard text form to convert.

`[in] AddressFamily`

The address family of the network address pointed to by the *AddressString* parameter.

`[in, optional] lpProtocolInfo`

The WSAPROTOCOL_INFO structure associated with the provider to be used. If this is **NULL**, the call is routed to the provider of the first protocol supporting the indicated *AddressFamily*.

`[out] lpAddress`

A pointer to a buffer that is filled with a sockaddr structure for the address string if the function succeeds.

`[in, out] lpAddressLength`

A pointer to the length, in bytes, of the buffer pointed to by the *lpAddress* parameter. If the function call is successful, this parameter returns a pointer to the size of the sockaddr structure returned in the *lpAddress* parameter. If the specified buffer is not large enough, the function fails with a specific error of WSAEFAULT and this parameter is updated with the required size in bytes.

Return value

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

ERROR CODE	MEANING
WSAEFAULT	The buffer pointed to by the *lpAddress* parameter is too small. Pass in a larger buffer.

WSAEINVAL	The functions was unable to translate the string into a sockaddr. See the following Remarks section for more information.
WSANOTINITIALISED	The WS2_32.DLL has not been initialized. The application must first call WSAStartup before calling any Windows Socket functions.
WSA_NOT_ENOUGH_MEMORY	There was insufficient memory to perform the operation.

Remarks

The **WSAStringToAddress** function converts a network address in standard text form into its numeric binary form in a sockaddr structure.

Any missing components of the address will be defaulted to a reasonable value, if possible. For example, a missing port number will default to zero. If the caller wants the translation to be done by a particular provider, it should supply the corresponding WSAPROTOCOL_INFO structure in the *lpProtocolInfo* parameter.

The **WSAStringToAddress** function fails (and returns WSAEINVAL) if the **sin_family** member of the SOCKADDR_IN structure, which is passed in the *lpAddress* parameter in the form of a **sockaddr** structure, is not set to AF_INET or AF_INET6.

Support for IPv6 addresses using the **WSAStringToAddress** function was added on Windows XP with Service Pack 1 (SP1)and later. IPv6 must also be installed on the local computer for the **WSAStringToAddress** function to support IPv6 addresses.

Windows Phone 8: This function is supported for Windows Phone Store apps on Windows Phone 8 and later.

Windows 8.1 and **Windows Server 2012 R2**: This function is supported for Windows Store apps on Windows 8.1, Windows Server 2012 R2, and later.

> **NOTE**
> The winsock2.h header defines WSAStringToAddress as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see Conventions for Function Prototypes.

Requirements

Minimum supported client	Windows 8.1, Windows Vista [desktop apps | UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps | UWP apps]
Target Platform	Windows
Header	winsock2.h
Library	Ws2_32.lib

DLL	Ws2_32.dll

See also

InetNtop

InetPton

RtlIpv4AddressToString

RtlIpv4AddressToStringEx

RtlIpv4StringToAddress

RtlIpv4StringToAddressEx

RtlIpv6AddressToString

RtlIpv6AddressToStringEx

RtlIpv6StringToAddress

RtlIpv6StringToAddressEx

WSAAddressToString

WSAPROTOCOL_INFO

WSAStartup

inet_addr

inet_ntoa

sockaddr

WSAUnhookBlockingHook function (winsock2.h)

10/22/2022 • 2 minutes to read • Edit Online

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

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

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

Syntax

```
int WSAAPI WSAUnhookBlockingHook();
```

Return value

None

Requirements

Target Platform	Windows
Header	winsock2.h

See also

Winsock Functions

Winsock Reference

WSAVERSION structure (winsock2.h)

10/22/2022 • 2 minutes to read • Edit Online

The **WSAVERSION** structure provides version comparison in Windows Sockets.

Syntax

```
typedef struct _WSAVersion {
  DWORD          dwVersion;
  WSAECOMPARATOR ecHow;
} WSAVERSION, *PWSAVERSION, *LPWSAVERSION;
```

Members

`dwVersion`

Version of Windows Sockets.

`ecHow`

WSAECOMPARATOR enumeration, used in the comparison.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	winsock2.h

See also

WSAECOMPARATOR

WSAWaitForMultipleEvents function (winsock2.h)

10/22/2022 • 8 minutes to read • Edit Online

The **WSAWaitForMultipleEvents** function returns when one or all of the specified event objects are in the signaled state, when the time-out interval expires, or when an I/O completion routine has executed.

Syntax

```
DWORD WSAAPI WSAWaitForMultipleEvents(
  [in] DWORD          cEvents,
  [in] const WSAEVENT *lphEvents,
  [in] BOOL           fWaitAll,
  [in] DWORD          dwTimeout,
  [in] BOOL           fAlertable
);
```

Parameters

`[in] cEvents`

The number of event object handles in the array pointed to by *lphEvents*. The maximum number of event object handles is **WSA_MAXIMUM_WAIT_EVENTS**. One or more events must be specified.

`[in] lphEvents`

A pointer to an array of event object handles. The array can contain handles of objects of different types. It may not contain multiple copies of the same handle if the *fWaitAll* parameter is set to **TRUE**. If one of these handles is closed while the wait is still pending, the behavior of **WSAWaitForMultipleEvents** is undefined.

The handles must have the **SYNCHRONIZE** access right. For more information, see Standard Access Rights.

`[in] fWaitAll`

A value that specifies the wait type. If **TRUE**, the function returns when the state of all objects in the *lphEvents* array is signaled. If **FALSE**, the function returns when any of the event objects is signaled. In the latter case, the return value minus **WSA_WAIT_EVENT_0** indicates the index of the event object whose state caused the function to return. If more than one event object became signaled during the call, this is the array index to the signaled event object with the smallest index value of all the signaled event objects.

`[in] dwTimeout`

The time-out interval, in milliseconds. **WSAWaitForMultipleEvents** returns if the time-out interval expires, even if conditions specified by the *fWaitAll* parameter are not satisfied. If the *dwTimeout* parameter is zero, **WSAWaitForMultipleEvents** tests the state of the specified event objects and returns immediately. If *dwTimeout* is **WSA_INFINITE**, **WSAWaitForMultipleEvents** waits forever; that is, the time-out interval never expires.

`[in] fAlertable`

A value that specifies whether the thread is placed in an alertable wait state so the system can execute I/O completion routines. If **TRUE**, the thread is placed in an alertable wait state and **WSAWaitForMultipleEvents** can return when the system executes an I/O completion routine. In this case, **WSA_WAIT_IO_COMPLETION** is returned and the event that was being waited on is not signaled yet. The application must call the

WSAWaitForMultipleEvents function again. If **FALSE**, the thread is not placed in an alertable wait state and I/O completion routines are not executed.

Return value

If the **WSAWaitForMultipleEvents** function succeeds, the return value upon success is one of the following values.

RETURN VALUE	MEANING
WSA_WAIT_EVENT_0 to (WSA_WAIT_EVENT_0 + cEvents - 1)	If the *fWaitAll* parameter is **TRUE**, the return value indicates that all specified event objects is signaled. If the *fWaitAll* parameter is **FALSE**, the return value minus **WSA_WAIT_EVENT_0** indicates the *lphEvents* array index of the signaled event object that satisfied the wait. If more than one event object became signaled during the call, the return value indicates the *lphEvents* array index of the signaled event object with the smallest index value of all the signaled event objects.
WSA_WAIT_IO_COMPLETION	The wait was ended by one or more I/O completion routines that were executed. The event that was being waited on is not signaled yet. The application must call the WSAWaitForMultipleEvents function again. This return value can only be returned if the *fAlertable* parameter is **TRUE**.
WSA_WAIT_TIMEOUT	The time-out interval elapsed and the conditions specified by the *fWaitAll* parameter were not satisfied. No I/O completion routines were executed.

If the **WSAWaitForMultipleEvents** function fails, the return value is **WSA_WAIT_FAILED**. The following table lists values that can be used with WSAGetLastError to get extended error information.

ERROR CODE	MEANING
WSANOTINITIALISED	A successful WSAStartup call must occur before using this function.
WSAENETDOWN	The network subsystem has failed.
WSAEINPROGRESS	A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function.
WSA_NOT_ENOUGH_MEMORY	Not enough free memory was available to complete the operation.
WSA_INVALID_HANDLE	One or more of the values in the *lphEvents* array is not a valid event object handle.
WSA_INVALID_PARAMETER	The *cEvents* parameter does not contain a valid handle count.

Remarks

The **WSAWaitForMultipleEvents** function determines whether the wait criteria have been met. If the criteria

have not been met, the calling thread enters the wait state. It uses no processor time while waiting for the criteria to be met.

The **WSAWaitForMultipleEvents** function returns when any one or all of the specified objects are in the signaled state, or when the time-out interval elapses.

When the *bWaitAll* parameter is **TRUE**, the wait operation is completed only when the states of all objects have been set to signaled. The function does not modify the states of the specified objects until the states of all objects have been set to signaled.

When *bWaitAll* parameter is **FALSE**, **WSAWaitForMultipleEvents** checks the handles in the *lphEvents* array in order starting with index 0, until one of the objects is signaled. If multiple objects become signaled, the function returns the index of the first handle in the *lphEvents* array whose object was signaled.

This function is also used to perform an alertable wait by setting the *fAlertable* parameter to **TRUE**. This enables the function to return when the system executes an I/O completion routine by the calling thread.

A thread must be in an alertable wait state in order for the system to execute I/O completion routines (asynchronous procedure calls or APCs). So if an application calls **WSAWaitForMultipleEvents** when there are pending asynchronous operations that have I/O completion routines and the *fAlertable* parameter is **FALSE**, then those I/O completion routines will not be executed even if those I/O operations are completed.

If the *fAlertable* parameter is **TRUE** and one of the pending operations completes, the APC is executed and **WSAWaitForMultipleEvents** will return **WSA_IO_COMPLETION**. The pending event is not signaled yet. The application must call the **WSAWaitForMultipleEvents** function again.

Applications that require an alertable wait state without waiting for any event objects to be signaled should use the Windows SleepEx function.

The current implementation of **WSAWaitForMultipleEvents** calls the WaitForMultipleObjectsEx function.

> **Note** Use caution when calling the **WSAWaitForMultipleEvents** with code that directly or indirectly creates windows. If a thread creates any windows, it must process messages. Message broadcasts are sent to all windows in the system. A thread that uses **WSAWaitForMultipleEvents** with no time-out limit (the *dwTimeout* parameter set to **WSA_INFINITE**) may cause the system to become deadlocked.

Example Code

The following code example shows how to use the **WSAWaitForMultipleEvents** function.

```
#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")

#define DATA_BUFSIZE 4096

int main()
{
    //-----------------------------------------
    // Declare and initialize variables
```

```
    WSADATA wsaData = { 0 };
    int iResult = 0;
    BOOL bResult = TRUE;

    WSABUF DataBuf;
    char buffer[DATA_BUFSIZE];

    DWORD EventTotal = 0;
    DWORD RecvBytes = 0;
    DWORD Flags = 0;
    DWORD BytesTransferred = 0;

    WSAEVENT EventArray[WSA_MAXIMUM_WAIT_EVENTS];
    WSAOVERLAPPED AcceptOverlapped;
    SOCKET ListenSocket = INVALID_SOCKET;
    SOCKET AcceptSocket = INVALID_SOCKET;

    DWORD Index;

    //-----------------------------------------
    // Initialize Winsock
    // Initialize Winsock
    iResult = WSAStartup(MAKEWORD(2, 2), &wsaData);
    if (iResult != 0) {
        wprintf(L"WSAStartup failed: %d\n", iResult);
        return 1;
    }
    //-----------------------------------------
    // Create a listening socket bound to a local
    // IP address and the port specified
    ListenSocket = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
    if (ListenSocket == INVALID_SOCKET) {
        wprintf(L"socket failed with error = %d\n", WSAGetLastError());
        WSACleanup();
        return 1;
    }

    u_short port = 27015;
    char *ip;
    sockaddr_in service;
    service.sin_family = AF_INET;
    service.sin_port = htons(port);
    hostent *thisHost;

    thisHost = gethostbyname("");
    if (thisHost == NULL) {
        wprintf(L"gethostbyname failed with error = %d\n", WSAGetLastError());
        closesocket(ListenSocket);
        WSACleanup();
        return 1;
    }

    ip = inet_ntoa(*(struct in_addr *) *thisHost->h_addr_list);

    service.sin_addr.s_addr = inet_addr(ip);

    //-----------------------------------------
    // Bind the listening socket to the local IP address
    // and port number
    iResult = bind(ListenSocket, (SOCKADDR *) & service, sizeof (SOCKADDR));
    if (iResult != 0) {
        wprintf(L"bind failed with error = %d\n", WSAGetLastError());
        closesocket(ListenSocket);
        WSACleanup();
        return 1;
    }
    //-----------------------------------------
    // Set the socket to listen for incoming
    // connection requests
```

```
    iResult = listen(ListenSocket, 1);
    if (iResult != 0) {
        wprintf(L"listen failed with error = %d\n", WSAGetLastError());
        closesocket(ListenSocket);
        WSACleanup();
        return 1;
    }
    wprintf(L"Listening...\n");

    //-----------------------------------------
    // Accept and incoming connection request
    AcceptSocket = accept(ListenSocket, NULL, NULL);
    if (AcceptSocket == INVALID_SOCKET) {
        wprintf(L"accept failed with error = %d\n", WSAGetLastError());
        closesocket(ListenSocket);
        WSACleanup();
        return 1;
    }
    wprintf(L"Client Accepted...\n");

    //-----------------------------------------
    // Create an event handle and setup an overlapped structure.
    EventArray[EventTotal] = WSACreateEvent();
    if (EventArray[EventTotal] == WSA_INVALID_EVENT) {
        wprintf(L"WSACreateEvent failed with error = %d\n", WSAGetLastError());
        closesocket(AcceptSocket);
        closesocket(ListenSocket);
        WSACleanup();
        return 1;
    }

    ZeroMemory(&AcceptOverlapped, sizeof (WSAOVERLAPPED));
    AcceptOverlapped.hEvent = EventArray[EventTotal];

    DataBuf.len = DATA_BUFSIZE;
    DataBuf.buf = buffer;

    EventTotal++;

    //-----------------------------------------
    // Call WSARecv to receive data into DataBuf on
    // the accepted socket in overlapped I/O mode
    if (WSARecv(AcceptSocket, &DataBuf, 1, &RecvBytes, &Flags, &AcceptOverlapped, NULL) ==
        SOCKET_ERROR) {
        iResult = WSAGetLastError();
        if (iResult != WSA_IO_PENDING)
            wprintf(L"WSARecv failed with error = %d\n", iResult);
    }
    //-----------------------------------------
    // Process overlapped receives on the socket
    while (1) {

        //-----------------------------------------
        // Wait for the overlapped I/O call to complete
        Index = WSAWaitForMultipleEvents(EventTotal, EventArray, FALSE, WSA_INFINITE, FALSE);

        //-----------------------------------------
        // Reset the signaled event
        bResult = WSAResetEvent(EventArray[Index - WSA_WAIT_EVENT_0]);
        if (bResult == FALSE) {
            wprintf(L"WSAResetEvent failed with error = %d\n", WSAGetLastError());
        }
        //-----------------------------------------
        // Determine the status of the overlapped event
        bResult =
            WSAGetOverlappedResult(AcceptSocket, &AcceptOverlapped, &BytesTransferred, FALSE,
                                   &Flags);
        if (bResult == FALSE) {
            wprintf(L"WSAGetOverlappedResult failed with error = %d\n", WSAGetLastError());
```

```
        }
        //----------------------------------------
        // If the connection has been closed, close the accepted socket
        if (BytesTransferred == 0) {
            wprintf(L"Closing accept Socket %d\n", AcceptSocket);
            closesocket(ListenSocket);
            closesocket(AcceptSocket);
            WSACloseEvent(EventArray[Index - WSA_WAIT_EVENT_0]);
            WSACleanup();
            return 1;
        }
        //----------------------------------------
        // If data has been received, echo the received data
        // from DataBuf back to the client
        iResult =
            WSASend(AcceptSocket, &DataBuf, 1, &RecvBytes, Flags, &AcceptOverlapped, NULL);
        if (iResult != 0) {
            wprintf(L"WSASend failed with error = %d\n", WSAGetLastError());
        }
        //----------------------------------------
        // Reset the changed flags and overlapped structure
        Flags = 0;
        ZeroMemory(&AcceptOverlapped, sizeof (WSAOVERLAPPED));

        AcceptOverlapped.hEvent = EventArray[Index - WSA_WAIT_EVENT_0];

        //----------------------------------------
        // Reset the data buffer
        DataBuf.len = DATA_BUFSIZE;
        DataBuf.buf = buffer;
    }

    closesocket(ListenSocket);
    closesocket(AcceptSocket);
    WSACleanup();

    return 0;
}
```

Windows Phone 8: This function is supported for Windows Phone Store apps on Windows Phone 8 and later.

Windows 8.1 and **Windows Server 2012 R2**: This function is supported for Windows Store apps on Windows 8.1, Windows Server 2012 R2, and later.

Requirements

Minimum supported client	Windows 8.1, Windows Vista [desktop apps | UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps | UWP apps]
Target Platform	Windows
Header	winsock2.h
Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

Standard Access Rights

WSACloseEvent

WSACreateEvent

WaitForMultipleObjectsEx

Winsock Functions

Winsock Reference

ws2atm.h header

10/22/2022 • 2 minutes to read • Edit Online

This header is used by Windows Sockets 2. For more information, see:

- Windows Sockets 2

ws2atm.h contains the following programming interfaces:

Structures

ATM_ADDRESS The ATM_ADDRESS structure holds ATM address data for ATM-based sockets.
ATM_BHLI The ATM_BHLI structure is used to identify B-HLI information for an associated ATM socket.
ATM_BLLI The ATM_BLLI structure is used to identify B-LLI information for an associated ATM socket.
sockaddr_atm The Windows Sockets sockaddr_atm structure stores socket address information for ATM sockets.

ATM_ADDRESS structure (ws2atm.h)

10/22/2022 • 2 minutes to read • Edit Online

The **ATM_ADDRESS** structure holds ATM address data for ATM-based sockets.

Syntax

```
typedef struct {
  DWORD AddressType;
  DWORD NumofDigits;
  UCHAR Addr[ATM_ADDR_SIZE];
} ATM_ADDRESS;
```

Members

`AddressType`

Type of end-system ATM address.

`NumofDigits`

Number of digits in the **Addr** parameter.

`Addr[ATM_ADDR_SIZE]`

Array representing the ATM address.

Remarks

For ATM_E164, enter the numbered digits in the same order in which they would be entered on a numeric keypad; that is, the number digit that would be entered first is located in **addr**. Digits are coded in IA5 characters. Bit 8 is set to zero.

For ATM_NSAP, code the address using Binary Coded Decimal (BCD) as defined in the ATM Forum UNI 3.1. The **NumofDigits** field is ignored in this case, and the NSAP-style address always contains 20 bytes.

A value of SAP_FIELD_ANY in **AddressType** indicates that the **satm_number** field is a wildcard. There are two more specialized wildcard values: SAP_FIELD_ANY_AESA_SEL and SAP_FIELD_ANY_AESA_REST. SAP_FIELD_ANY_AESA_SEL means that this is an NSAP-style ATM Endsystem Address and the selector octet is set as a wildcard. SAP_FIELD_ANY_AESA_REST means that this is an NSAP-style ATM Endsystem Address and all the octets except for the selector octet are set as wildcards.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	ws2atm.h

See also

sockaddr_atm

ATM_BHLI structure (ws2atm.h)

10/22/2022 • 2 minutes to read • Edit Online

The **ATM_BHLI** structure is used to identify B-HLI information for an associated ATM socket.

Syntax

```
typedef struct {
  DWORD HighLayerInfoType;
  DWORD HighLayerInfoLength;
  UCHAR HighLayerInfo[8];
} ATM_BHLI;
```

Members

`HighLayerInfoType`

Identifies the **high layer information type** field in the B-LLI information element. Note that the type **BHLI_HighLayerProfile** has been eliminated in UNI 3.1. A value of SAP_FIELD_ABSENT indicates that B-HLI is not present, and a value of SAP_FIELD_ANY means wildcard.

`HighLayerInfoLength`

Identifies the number of bytes from one to eight in the **HighLayerInfo** array. Valid values include eight for the cases of BHLI_ISO and BHLI_UserSpecific, four for BHLI_HighLayerProfile, and seven for BHLI_VendorSpecificAppId.

`HighLayerInfo[8]`

Identifies the **high layer information** field in the B-LLI information element. In the case of **HighLayerInfoType** being BHLI_VendorSpecificAppId, the first 3 bytes consist of a globally-administered organizationally unique identifier (OUI), (according to IEEE standard 802-1990), followed by a 4-byte application identifier, which is administered by the vendor identified by the OUI. Value for the case of BHLI_UserSpecific is user defined and requires bilateral agreement between two end users.

Remarks

The following are the manifest constants associated with the **ATM_BHLI** structure:

```
#include <windows.h>
/*
 *  values used for the HighLayerInfoType field in struct ATM_BHLI
 */

#define BHLI_ISO                    0x00   /* ISO                                  */
#define BHLI_UserSpecific           0x01   /* User Specific                        */
#define BHLI_HighLayerProfile       0x02   /* High layer profile (only in UNI3.0) */
#define BHLI_VendorSpecificAppId    0x03   /* Vendor-Specific Application ID       */
```

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	ws2atm.h

See also

ATM_ADDRESS

ATM_BLLI

sockaddr_atm

ATM_BLLI structure (ws2atm.h)

10/22/2022 • 2 minutes to read • Edit Online

The **ATM_BLLI** structure is used to identify B-LLI information for an associated ATM socket.

Syntax

```
typedef struct {
  DWORD Layer2Protocol;
  DWORD Layer2UserSpecifiedProtocol;
  DWORD Layer3Protocol;
  DWORD Layer3UserSpecifiedProtocol;
  DWORD Layer3IPI;
  UCHAR SnapID[5];
} ATM_BLLI;
```

Members

`Layer2Protocol`

Identifies the layer-two protocol. Corresponds to the *User information layer 2 protocol* field in the B-LLI information element. A value of SAP_FIELD_ABSENT indicates that this field is not used, and a value of SAP_FIELD_ANY means wildcard.

`Layer2UserSpecifiedProtocol`

Identifies the user-specified layer-two protocol. Only used if the **Layer2Protocol** parameter is set to BLLI_L2_USER_SPECIFIED. The valid values range from zero–127. Corresponds to the *User specified layer 2 protocol information* field in the B-LLI information element.

`Layer3Protocol`

Identifies the layer-three protocol. Corresponds to the *User information layer 3 protocol* field in the B-LLI information element. A value of SAP_FIELD_ABSENT indicates that this field is not used, and a value of SAP_FIELD_ANY means wildcard.

`Layer3UserSpecifiedProtocol`

Identifies the user-specified layer-three protocol. Only used if the **Layer3Protocol** parameter is set to BLLI_L3_USER_SPECIFIED. The valid values range from zero–127. Corresponds to the *User specified layer 3 protocol information* field in the B-LLI information element.

`Layer3IPI`

Identifies the layer-three Initial Protocol Identifier. Only used if the **Layer3Protocol** parameter is set to BLLI_L3_ISO_TR9577. Corresponds to the *ISO/IEC TR 9577 Initial Protocol Identifier* field in the B-LLI information element.

`SnapID[5]`

Identifies the 802.1 SNAP identifier. Only used if the **Layer3Protocol** parameter is set to BLLI_L3_ISO_TR9577 and **Layer3IPI** is set to BLLI_L3_IPI_SNAP, indicating an IEEE 802.1 SNAP identifier. Corresponds to the *OUI* and *PID* fields in the B-LLI information element.

Remarks

The following are the manifest constants associated with the **ATM_BLLI** structure:

```
#include <windows.h>

/*
 *  values used for Layer2Protocol in struct B-LLI
 */
#define BLLI_L2_ISO_1745           0x01   /* Basic mode ISO 1745    */
#define BLLI_L2_Q921               0x02   /* CCITT Rec. Q.921       */
#define BLLI_L2_X25L               0x06   /* CCITT Rec. X.25, link layer              */
#define BLLI_L2_X25M               0x07   /* CCITT Rec. X.25, multilink               */
#define BLLI_L2_ELAPB              0x08   /* Extended LAPB; for half duplex operation */
#define BLLI_L2_HDLC_NRM           0x09   /* HDLC NRM (ISO 4335)                      */
#define BLLI_L2_HDLC_ABM           0x0A   /* HDLC ABM (ISO 4335)                      */
#define BLLI_L2_HDLC_ARM           0x0B   /* HDLC ARM (ISO 4335)                      */
#define BLLI_L2_LLC                0x0C   /* LAN logical link control (ISO 8802/2)    */
#define BLLI_L2_X75                0x0D   /* CCITT Rec. X.75, single link procedure   */
#define BLLI_L2_Q922               0x0E   /* CCITT Rec. Q.922                         */
#define BLLI_L2_USER_SPECIFIED     0x10   /* User Specified                           */
#define BLLI_L2_ISO_7776           0x11   /* ISO 7776 DTE-DTE operation               */

/*
 *  values used for Layer3Protocol in struct B-LLI
 */
#define BLLI_L3_X25                0x06   /* CCITT Rec. X.25, packet layer            */
#define BLLI_L3_ISO_8208           0x07   /* ISO/IEC 8208 (X.25 packet layer for DTE  */
#define BLLI_L3_X223               0x08   /* X.223/ISO 8878                           */
#define BLLI_L3_SIO_8473           0x09   /* ISO/IEC 8473 (OSI connectionless)        */
#define BLLI_L3_T70                0x0A   /* CCITT Rec. T.70 min. network layer       */
#define BLLI_L3_ISO_TR9577         0x0B   /* ISO/IEC TR 9577 Network Layer Protocol ID*/
#define BLLI_L3_USER_SPECIFIED     0x10   /* User Specified                           */

/*
 *  values used for Layer3IPI in struct B-LLI
 */
#define BLLI_L3_IPI_SNAP           0x80   /* IEEE 802.1 SNAP identifier               */
#define BLLI_L3_IPI_IP             0xCC   /* Internet Protocol (IP) identifier        */
```

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	ws2atm.h

See also

ATM_ADDRESS

ATM_BHLI

sockaddr_atm

sockaddr_atm structure (ws2atm.h)

10/22/2022 • 3 minutes to read • Edit Online

The Windows Sockets **sockaddr_atm** structure stores socket address information for ATM sockets.

Syntax

```
typedef struct sockaddr_atm {
  u_short     satm_family;
  ATM_ADDRESS satm_number;
  ATM_BLLI    satm_blli;
  ATM_BHLI    satm_bhli;
} sockaddr_atm, SOCKADDR_ATM, *PSOCKADDR_ATM, *LPSOCKADDR_ATM;
```

Members

`satm_family`

Identifies the address family, which is AF_ATM in this case.

`satm_number`

Identifies the ATM address that could be either in E.164 or NSAP-style ATM End Systems Address format. This field will be mapped to the called party number information element (IE) if it is specified in bind and WSPBind for a listening socket, or in connect, WSAConnect, WSPConnect, WSAJoinLeaf, or WSPJoinLeaf for a connecting socket. It will be mapped to the Calling Party Number IE if specified in **bind** and **WSPBind** for a connecting socket.

`satm_blli`

Identifies the fields in the B-LLI information element that are used along with **satm_bhli** to identify an application. See ATM_BLLI for more details. Note that the B-LLI layer two information is treated as not present if its **Layer2Protocol** field contains SAP_FIELD_ABSENT, or as a wildcard if it contains SAP_FIELD_ANY. Similarly, the B-LLI layer three information is treated as not present if its **Layer3Protocol** field contains SAP_FIELD_ABSENT, or as a wildcard if it contains SAP_FIELD_ANY.

`satm_bhli`

Identifies the fields in the B-HLI information element that are used along with **satm_blli** to identify an application. See ATM_BHLI for information about the **ATM_BHLI** structure.

> **Note** **satm_bhli** is treated as not present if its **HighLayerInfoType** field contains SAP_FIELD_ABSENT, or as a wildcard if it contains SAP_FIELD_ANY.

Remarks

For listening sockets, the **sockaddr_atm** structure is used in bind/WSPBind to register a Service Access Point (SAP) to receive incoming connection requests destined to this SAP. SAP registration is used to match against the SAP specified in an incoming connection request to determine which listening socket is to receive this

request. In the current specification, overlapping registration is not allowed. Overlapping registration is defined as having more than one registered SAP to potentially match the SAP specified in any incoming connection request. Listen and WSPListen will return the error code WSAEADDRINUSE if the SAP associated with the listening socket overlaps with any currently registered SAPs in the system.

The fields in a SAP to be registered must contain either a valid value, or one of two special manifest constants: SAP_FIELD_ABSENT or SAP_FIELD_ANY.

SAP_FIELD_ABSENT simply means that this field is not presented as part of a SAP. SAP_FIELD_ANY means using wildcards.

Note that the requirement of nonoverlapping registration does not preclude using wildcards. For example, it is possible to have two registered SAPs that both contain SAP_FIELD_ANY in some fields and different values in other fields.

> **Note** The called party ATM number is mandatory, thus the **satm_number** field cannot contain SAP_FIELD_ABSENT.

For connecting sockets, the **sockaddr_atm** structure is used to specify the destination SAP in connect/WSAConnect/WSPConnect for point-to-point connections, and WSAJoinLeaf/WSPJoinLeaf for point-to-multipoint connections. The fields in the destination SAP of a connecting socket must contain either a valid value or SAP_FIELD_ABSENT, that is, SAP_FIELD_ANY is not allowed.

Furthermore, SAP_FIELD_ABSENT is not allowed for the **satm_number** field. The destination SAP is used to match against all the registered SAPs in the destination computer to determine the forwarding destination for this connection request. If each and every field of the destination SAP of an incoming request either equals the corresponding field of a registered SAP, or the corresponding field contains the SAP_FIELD_ANY, the listening socket associated with this registered SAP will receive the incoming connection request.

If bind and/or WSPBind are used on a connecting socket to specify the calling party ATM address, the **satm_blli** and **satm_bhli** fields should be ignored and the ones specified in connect, WSAConnect, or WSPConnect will be used.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	ws2atm.h

See also

ATM_ADDRESS

ATM_BHLI

ATM_BLLI

WSAConnect

WSAJoinLeaf

WSPBind

WSPConnect

WSPJoinLeaf

WSPListen

bind

connect

listen

ws2def.h header

10/22/2022 • 2 minutes to read • Edit Online

This header is used by multiple technologies. For more information, see:

- IP Helper
- Network Drivers, Windows Vista and Later
- Windows Sockets 2

ws2def.h contains the following programming interfaces:

Structures

ADDRINFO_DNS_SERVER Represents a custom Domain Name System (DNS) server, used in the Winsock APIs.
ADDRINFOA Used by the getaddrinfo function to hold host address information.
ADDRINFOEX2A Used by the GetAddrInfoEx function to hold host address information when both a canonical name and a fully qualified domain name have been requested. (ANSI)
ADDRINFOEX2W Used by the GetAddrInfoEx function to hold host address information when both a canonical name and a fully qualified domain name have been requested. (Unicode)
ADDRINFOEX3 Used by the GetAddrInfoEx function to hold host address information when a specific network interface has been requested. (addrinfoex3W)
ADDRINFOEX4 Used by the GetAddrInfoEx function to hold host address information when a specific network interface has been requested. (addrinfoex4W)
ADDRINFOEX5 Used by the GetAddrInfoExW function to hold host address information when a specific network interface has been requested. (ADDRINFOEX5)
ADDRINFOEX6 Used by the GetAddrInfoExW function to hold host address information when a specific network interface has been requested. (ADDRINFOEX6)

ADDRINFOEXA

Used by the GetAddrInfoEx function to hold host address information. (ANSI)

ADDRINFOEXW

Used by the GetAddrInfoEx function to hold host address information. (Unicode)

ADDRINFOW

Used by the GetAddrInfoW function to hold host address information.

CSADDR_INFO

The CSADDR_INFO (ws2def.h) structure contains Windows Sockets address information for a socket, network service, or namespace provider.

SOCKADDR

The SOCKADDR structure is a generic structure that specifies a transport address.

SOCKADDR_IN

The SOCKADDR_IN structure specifies a transport address and port for the AF_INET address family.

SOCKADDR_STORAGE_LH

The SOCKADDR_STORAGE structure is a generic structure that specifies a transport address. (SOCKADDR_STORAGE_LH)

SOCKADDR_STORAGE_XP

The SOCKADDR_STORAGE structure is a generic structure that specifies a transport address.S

SOCKET_ADDRESS

SOCKET_ADDRESS structure stores protocol-specific address information.

SOCKET_ADDRESS_LIST

The SOCKET_ADDRESS_LIST structure defines a variable-sized list of transport addresses.

SOCKET_PROCESSOR_AFFINITY

Contains the association between a socket and an RSS processor core and NUMA node.

WSABUF

The WSABUF structure enables the creation or manipulation of a data buffer used by some Winsock functions.

WSACMSGHDR

The CMSGHDR structure defines the header for a control data object that is associated with a datagram.

WSAMSG	Used with the WSARecvMsg and WSASendMsg functions to store address and optional control information about connected and unconnected sockets as well as an array of buffers used to store message data.

Enumerations

SCOPE_LEVEL	The SCOPE_LEVEL enumeration is used with the IP_ADAPTER_ADDRESSES structure to identify scope levels for IPv6 addresses.

ADDRINFO_DNS_SERVER structure (ws2def.h)

10/22/2022 • 2 minutes to read • Edit Online

Represents a custom Domain Name System (DNS) server, used in the Winsock APIs.

Syntax

```
typedef struct addrinfo_dns_server {
  unsigned int     ai_servertype;
  unsigned __int64 ai_flags;
  unsigned int     ai_addrlen;
  struct sockaddr  *ai_addr;
  union {
    PWSTR ai_template;
  };
} ADDRINFO_DNS_SERVER;
```

Members

`ai_servertype`

The type of DNS server. Can be one of the following values.

CONSTANT	VALUE	MEANING
AI_DNS_SERVER_TYPE_UDP	1	A regular DNS server.
AI_DNS_SERVER_TYPE_DOH	2	A *DNS-over-HTTPS* server.

`ai_flags`

A bitmap containing any of the following options.

CONSTANT	VALUE	MEANING
AI_DNS_SERVER_UDP_FALLBACK	0x1	This server can also be used for non-secure name resolution.

`ai_addrlen`

The length in bytes of the socket address structure that *ai_addr* points to.

`ai_addr`

A pointer to a socket address structure containing the address of the custom server. Only **SOCKADDR_IN** and **SOCKADDR_IN6** structures are supported. The *sa_family* member must be set to **AF_INET** or **AF_INET6**. The rest of the structure must be zeroed out, with the exception of the **SOCKADDR_IN::sin_addr** member for IPv4, or **SOCKADDR_IN6::sin6_addr** for IPv6.

`ai_template`

If *ai_servertype* is set to **AI_DNS_SERVER_TYPE_DOH**, then this member must point to a **NULL**-terminated wide string representing the *DNS-over-HTTPS* template for this server.

If *ai_servertype* is set to **AI_DNS_SERVER_TYPE_UDP**, then this field must be **NULL**.

Requirements

Minimum supported client	Windows Build 22000
Minimum supported server	Windows Build 22000
Header	ws2def.h

See also

- ADDRINFOEX6

ADDRINFOA structure (ws2def.h)

10/22/2022 • 11 minutes to read • Edit Online

The **addrinfo** structure is used by the getaddrinfo function to hold host address information.

Syntax

```
typedef struct addrinfo {
  int             ai_flags;
  int             ai_family;
  int             ai_socktype;
  int             ai_protocol;
  size_t          ai_addrlen;
  char            *ai_canonname;
  struct sockaddr *ai_addr;
  struct addrinfo *ai_next;
} ADDRINFOA, *PADDRINFOA;
```

Members

`ai_flags`

Type: **int**

Flags that indicate options used in the getaddrinfo function.

Supported values for the **ai_flags** member are defined in the *Ws2def.h* header file on the Windows SDK for Windows 7 and later. These values are defined in the *Ws2tcpip.h* header file on the Windows SDK for Windows Server 2008 and Windows Vista. These values are defined in the *Ws2tcpip.h* header file on the Platform SDK for Windows Server 2003, and Windows XP. Supported values for the **ai_flags** member can be a combination of the following options.

VALUE	MEANING
AI_PASSIVE 0x01	The socket address will be used in a call to the bind function.
AI_CANONNAME 0x02	The canonical name is returned in the first **ai_canonname** member.
AI_NUMERICHOST 0x04	The *nodename* parameter passed to the getaddrinfo function must be a numeric string.
AI_ALL 0x0100	If this bit is set, a request is made for IPv6 addresses and IPv4 addresses with **AI_V4MAPPED**. This option is supported on Windows Vista and later.

AI_ADDRCONFIG 0x0400	The getaddrinfo will resolve only if a global address is configured. The IPv6 and IPv4 loopback address is not considered a valid global address. This option is supported on Windows Vista and later.
AI_V4MAPPED 0x0800	If the getaddrinfo request for IPv6 addresses fails, a name service request is made for IPv4 addresses and these addresses are converted to IPv4-mapped IPv6 address format. This option is supported on Windows Vista and later.
AI_NON_AUTHORITATIVE 0x04000	The address information can be from a non-authoritative namespace provider. This option is only supported on Windows Vista and later for the **NS_EMAIL** namespace.
AI_SECURE 0x08000	The address information is from a secure channel. This option is only supported on Windows Vista and later for the **NS_EMAIL** namespace.
AI_RETURN_PREFERRED_NAMES 0x010000	The address information is for a preferred name for a user. This option is only supported on Windows Vista and later for the **NS_EMAIL** namespace.
AI_FQDN 0x00020000	If a flat name (single label) is specified, getaddrinfo will return the fully qualified domain name that the name eventually resolved to. The fully qualified domain name is returned in the **ai_canonname** member. This is different than **AI_CANONNAME** bit flag that returns the canonical name registered in DNS which may be different than the fully qualified domain name that the flat name resolved to. Only one of the **AI_FQDN** and **AI_CANONNAME** bits can be set. The getaddrinfo function will fail if both flags are present with **EAI_BADFLAGS**. This option is supported on Windows 7, Windows Server 2008 R2, and later.
AI_FILESERVER 0x00040000	A hint to the namespace provider that the hostname being queried is being used in a file share scenario. The namespace provider may ignore this hint. This option is supported on Windows 7, Windows Server 2008 R2, and later.

`ai_family`

Type: **int**

The address family. Possible values for the address family are defined in the *Winsock2.h* header file.

On the Windows SDK released for Windows Vista and later, the organization of header files has changed and the 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. Note that the values for the AF_ address family and PF_ protocol family constants are identical (for example, **AF_UNSPEC** and **PF_UNSPEC**), so either constant can be used.

The following table lists common values for the address family although many other values are possible.

VALUE	MEANING
AF_UNSPEC 0	The address family is unspecified.
AF_INET 2	The Internet Protocol version 4 (IPv4) address family.
AF_NETBIOS 17	The NetBIOS address family. This address family is only supported if a Windows Sockets provider for NetBIOS is installed.
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 a Bluetooth adapter is installed on Windows Server 2003 or later.

`ai_socktype`

Type: **int**

The socket type. Possible values for the socket type are defined in the *Winsock2.h* header file.

The following table lists the possible values for the socket type supported for Windows Sockets 2:

VALUE	MEANING
SOCK_STREAM 1	Provides sequenced, reliable, two-way, connection-based byte streams with an OOB data transmission mechanism. Uses the Transmission Control Protocol (TCP) for the Internet address family (**AF_INET** or **AF_INET6**). If the **ai_family** member is **AF_IRDA**, then **SOCK_STREAM** is the only supported socket type.
SOCK_DGRAM 2	Supports datagrams, which are connectionless, unreliable buffers of a fixed (typically small) maximum length. Uses the User Datagram Protocol (UDP) for the Internet address family (**AF_INET** or **AF_INET6**).

SOCK_RAW 3	Provides a raw socket that allows an application to manipulate the next upper-layer protocol header. To manipulate the IPv4 header, the IP_HDRINCL socket option must be set on the socket. To manipulate the IPv6 header, the IPV6_HDRINCL socket option must be set on the socket.
SOCK_RDM 4	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 reliable multicast programming.
SOCK_SEQPACKET 5	Provides a pseudo-stream packet based on datagrams.

In Windows Sockets 2, new socket types were introduced. An application can dynamically discover the attributes of each available transport protocol through the WSAEnumProtocols function. So an application can determine the possible socket type and protocol options for an address family and use this information when specifying this parameter. Socket type definitions in the *Winsock2.h* and *Ws2def.h* header files will be periodically updated as new socket types, address families, and protocols are defined.

In Windows Sockets 1.1, the only possible socket types are **SOCK_DATAGRAM** and **SOCK_STREAM**.

`ai_protocol`

Type: **int**

The protocol type. The possible options are specific to the address family and socket type specified. Possible values for the **ai_protocol** are defined in the *Winsock2.h* and *Wsrm.h* header files.

On the Windows SDK released for Windows Vista and later, the organization of header files has changed and this member can be one of the values from 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 for **ai_protocol**, the caller does not wish to specify a protocol and the service provider will choose the **ai_protocol** to use. For protocols other than IPv4 and IPv6, set **ai_protocol** to zero.

The following table lists common values for the **ai_protocol** member although many other values are possible.

VALUE	MEANING
IPPROTO_TCP 6	The Transmission Control Protocol (TCP). This is a possible value when the **ai_family** member is **AF_INET** or **AF_INET6** and the **ai_socktype** member is **SOCK_STREAM**.
IPPROTO_UDP 17	The User Datagram Protocol (UDP). This is a possible value when the **ai_family** member is **AF_INET** or **AF_INET6** and the *type* parameter is **SOCK_DGRAM**.
IPPROTO_RM 113	The PGM protocol for reliable multicast. This is a possible value when the **ai_family** member is **AF_INET** and the **ai_socktype** member is **SOCK_RDM**. On the Windows SDK released for Windows Vista and later, this value is also called **IPPROTO_PGM**.

If the **ai_family** member is **AF_IRDA**, then the **ai_protocol** must be 0.

`ai_addrlen`

Type: **size_t**

The length, in bytes, of the buffer pointed to by the **ai_addr** member.

`ai_canonname`

Type: **char***

The canonical name for the host.

`ai_addr`

Type: **struct sockaddr***

A pointer to a sockaddr structure. The **ai_addr** member in each returned **addrinfo** structure points to a filled-in socket address structure. The length, in bytes, of each returned **addrinfo** structure is specified in the **ai_addrlen** member.

`ai_next`

Type: **struct addrinfo***

A pointer to the next structure in a linked list. This parameter is set to **NULL** in the last **addrinfo** structure of a linked list.

Remarks

The **addrinfo** structure is used by the ANSI getaddrinfo function to hold host address information.

The addrinfoW structure is the version of this structure used by the Unicode GetAddrInfoW function.

Macros in the *Ws2tcpip.h* header file define a **ADDRINFOT** structure and a mixed-case function name of **GetAddrInfo**. The **GetAddrInfo** function should be called with the *nodename* and *servname* parameters of a pointer of type **TCHAR** and the *hints* and *res* parameters of a pointer of type **ADDRINFOT**. When UNICODE or _UNICODE is not defined, **ADDRINFOT** is defined to the **addrinfo** structure and **GetAddrInfo** is defined to getaddrinfo, the ANSI version of this function. When UNICODE or _UNICODE is defined, **ADDRINFOT** is defined to the addrinfoW structure and **GetAddrInfo** is defined to GetAddrInfoW, the Unicode version of this function.

Upon a successful call to getaddrinfo, a linked list of **addrinfo** structures is returned in the *res* parameter passed to the **getaddrinfo** function. The list can be processed by following the pointer provided in the **ai_next** member of each returned **addrinfo** structure until a **NULL** pointer is encountered. In each returned **addrinfo** structure, the **ai_family**, **ai_socktype**, and **ai_protocol** members correspond to respective arguments in a socket or WSASocket function call. Also, the **ai_addr** member in each returned **addrinfo** structure points to a filled-in socket address structure, the length of which is specified in its **ai_addrlen** member.

Support for getaddrinfo and the addrinfo struct on older versions of Windows

The getaddrinfo function that uses the **addrinfo** structure was added to the *Ws2_32.dll* on Windows XP and later. The **addrinfo** structure is defined in the Ws2tcpip.h header file included with the Platform SDK released for Windows XP and later and the Windows SDK released for Windows Vista and later.

To execute an application that uses the getaddrinfo function and the **addrinfo** structure on earlier versions of Windows (Windows 2000), then you need to include the *Ws2tcpip.h* and *Wspiapi.h* files. When the *Wspiapi.h* include file is added, the **getaddrinfo** function is defined to the WspiapiGetAddrInfo inline function in the *Wspiapi.h* file. At runtime, the WspiapiGetAddrInfo function is implemented in such a way that if the *Ws2_32.dll* or the *Wship6.dll* (the file containing **getaddrinfo** in the IPv6 Technology Preview for Windows 2000) does not include **getaddrinfo**, then a version of **getaddrinfo** is implemented inline based on code in the *Wspiapi.h*

header file. This inline code will be used on older Windows platforms that do not natively support the **getaddrinfo** function.

The IPv6 protocol is supported on Windows 2000 when the IPv6 Technology Preview for Windows 2000 is installed. Otherwise getaddrinfo support on versions of Windows earlier than Windows XP is limited to handling IPv4 name resolution.

The GetAddrInfoW function that uses the addrinfoW structure is the Unicode version of the getaddrinfo function and associated **addrinfo** structure. The **GetAddrInfoW** function was added to the *Ws2_32.dll* in Windows XP with Service Pack 2 (SP2). The **GetAddrInfoW** function and the **addrinfoW** structure cannot be used on versions of Windows earlier than Windows XP with SP2.

Examples

The following code example shows the use of the **addrinfo** structure.

```
#undef UNICODE

#include <winsock2.h>
#include <ws2tcpip.h>
#include <stdio.h>

// link with Ws2_32.lib
#pragma comment(lib, "Ws2_32.lib")

int __cdecl main(int argc, char **argv)
{

    //-----------------------------------------
    // Declare and initialize variables
    WSADATA wsaData;
    int iResult;
    INT iRetval;

    DWORD dwRetval;

    int i = 1;

    struct addrinfo *result = NULL;
    struct addrinfo *ptr = NULL;
    struct addrinfo hints;

    struct sockaddr_in  *sockaddr_ipv4;
//    struct sockaddr_in6 *sockaddr_ipv6;
    LPSOCKADDR sockaddr_ip;

    char ipstringbuffer[46];
    DWORD ipbufferlength = 46;

    // Validate the parameters
    if (argc != 3) {
        printf("usage: %s <hostname> <servicename>\n", argv[0]);
        printf("       provides protocol-independent translation\n");
        printf("       from an ANSI host name to an IP address\n");
        printf("%s example usage\n", argv[0]);
        printf("   %s www.contoso.com 0\n", argv[0]);
        return 1;
    }

    // Initialize Winsock
    iResult = WSAStartup(MAKEWORD(2, 2), &wsaData);
    if (iResult != 0) {
        printf("WSAStartup failed: %d\n", iResult);
        return 1;
    }
```

```
    //--------------------------------
    // Setup the hints address info structure
    // which is passed to the getaddrinfo() function
    ZeroMemory( &hints, sizeof(hints) );
    hints.ai_family = AF_UNSPEC;
    hints.ai_socktype = SOCK_STREAM;
    hints.ai_protocol = IPPROTO_TCP;

    printf("Calling getaddrinfo with following parameters:\n");
    printf("\tnodename = %s\n", argv[1]);
    printf("\tservname (or port) = %s\n\n", argv[2]);

//--------------------------------
// Call getaddrinfo(). If the call succeeds,
// the result variable will hold a linked list
// of addrinfo structures containing response
// information
    dwRetval = getaddrinfo(argv[1], argv[2], &hints, &result);
    if ( dwRetval != 0 ) {
        printf("getaddrinfo failed with error: %d\n", dwRetval);
        WSACleanup();
        return 1;
    }

    printf("getaddrinfo returned success\n");

    // Retrieve each address and print out the hex bytes
    for(ptr=result; ptr != NULL ;ptr=ptr->ai_next) {

        printf("getaddrinfo response %d\n", i++);
        printf("\tFlags: 0x%x\n", ptr->ai_flags);
        printf("\tFamily: ");
        switch (ptr->ai_family) {
            case AF_UNSPEC:
                printf("Unspecified\n");
                break;
            case AF_INET:
                printf("AF_INET (IPv4)\n");
                sockaddr_ipv4 = (struct sockaddr_in *) ptr->ai_addr;
                printf("\tIPv4 address %s\n",
                    inet_ntoa(sockaddr_ipv4->sin_addr) );
                break;
            case AF_INET6:
                printf("AF_INET6 (IPv6)\n");
                // the InetNtop function is available on Windows Vista and later
                // sockaddr_ipv6 = (struct sockaddr_in6 *) ptr->ai_addr;
                // printf("\tIPv6 address %s\n",
                //    InetNtop(AF_INET6, &sockaddr_ipv6->sin6_addr, ipstringbuffer, 46) );

                // We use WSAAddressToString since it is supported on Windows XP and later
                sockaddr_ip = (LPSOCKADDR) ptr->ai_addr;
                // The buffer length is changed by each call to WSAAddresstoString
                // So we need to set it for each iteration through the loop for safety
                ipbufferlength = 46;
                iRetval = WSAAddressToString(sockaddr_ip, (DWORD) ptr->ai_addrlen, NULL,
                    ipstringbuffer, &ipbufferlength );
                if (iRetval)
                    printf("WSAAddressToString failed with %u\n", WSAGetLastError() );
                else
                    printf("\tIPv6 address %s\n", ipstringbuffer);
                break;
            case AF_NETBIOS:
                printf("AF_NETBIOS (NetBIOS)\n");
                break;
            default:
                printf("Other %ld\n", ptr->ai_family);
                break;
        }
        printf("\tSocket type: ");
```

```
        printf("\tSocket type: ");
        switch (ptr->ai_socktype) {
            case 0:
                printf("Unspecified\n");
                break;
            case SOCK_STREAM:
                printf("SOCK_STREAM (stream)\n");
                break;
            case SOCK_DGRAM:
                printf("SOCK_DGRAM (datagram) \n");
                break;
            case SOCK_RAW:
                printf("SOCK_RAW (raw) \n");
                break;
            case SOCK_RDM:
                printf("SOCK_RDM (reliable message datagram)\n");
                break;
            case SOCK_SEQPACKET:
                printf("SOCK_SEQPACKET (pseudo-stream packet)\n");
                break;
            default:
                printf("Other %ld\n", ptr->ai_socktype);
                break;
        }
        printf("\tProtocol: ");
        switch (ptr->ai_protocol) {
            case 0:
                printf("Unspecified\n");
                break;
            case IPPROTO_TCP:
                printf("IPPROTO_TCP (TCP)\n");
                break;
            case IPPROTO_UDP:
                printf("IPPROTO_UDP (UDP) \n");
                break;
            default:
                printf("Other %ld\n", ptr->ai_protocol);
                break;
        }
        printf("\tLength of this sockaddr: %d\n", ptr->ai_addrlen);
        printf("\tCanonical name: %s\n", ptr->ai_canonname);
    }

    freeaddrinfo(result);
    WSACleanup();

    return 0;
}
```

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	ws2def.h

See also

GetAddrInfoW

WSAEnumProtocols

addrinfoW

bind

getaddrinfo

sockaddr

ADDRINFOEX2A structure (ws2def.h)

10/22/2022 • 9 minutes to read • Edit Online

The **addrinfoex2** structure is used by the GetAddrInfoEx function to hold host address information when both a canonical name and a fully qualified domain name have been requested.

Syntax

```
typedef struct addrinfoex2A {
  int                 ai_flags;
  int                 ai_family;
  int                 ai_socktype;
  int                 ai_protocol;
  size_t              ai_addrlen;
  char                *ai_canonname;
  struct sockaddr     *ai_addr;
  void                *ai_blob;
  size_t              ai_bloblen;
  LPGUID              ai_provider;
  struct addrinfoex2A *ai_next;
  int                 ai_version;
  char                *ai_fqdn;
} ADDRINFOEX2A, *PADDRINFOEX2A, *LPADDRINFOEX2A;
```

Members

`ai_flags`

Flags that indicate options used in the GetAddrInfoEx function.

Supported values for the **ai_flags** member are defined in the *Winsock2.h* include file and can be a combination of the following options.

VALUE	MEANING
AI_PASSIVE 0x01	The socket address will be used in a call to the bind function.
AI_CANONNAME 0x02	The canonical name is returned in the first **ai_canonname** member.
AI_NUMERICHOST 0x04	The *nodename* parameter passed to the GetAddrInfoEx function must be a numeric string.
AI_ALL 0x0100	If this bit is set, a request is made for IPv6 addresses and IPv4 addresses with **AI_V4MAPPED**. This option is supported on Windows Vista and later.

AI_ADDRCONFIG 0x0400	The GetAddrInfoEx will resolve only if a global address is configured. The IPv6 and IPv4 loopback address is not considered a valid global address. This option is supported on Windows Vista and later.
AI_V4MAPPED 0x0800	If the GetAddrInfoEx request for an IPv6 addresses fails, a name service request is made for IPv4 addresses and these addresses are converted to IPv4-mapped IPv6 address format. This option is supported on Windows Vista and later.
AI_NON_AUTHORITATIVE 0x04000	The address information is from non-authoritative results. When this option is set in the *pHints* parameter of GetAddrInfoEx, the **NS_EMAIL** namespace provider returns both authoritative and non-authoritative results. If this option is not set, then only authoritative results are returned. This option is only supported on Windows Vista and later for the **NS_EMAIL** namespace.
AI_SECURE 0x08000	The address information is from a secure channel. If the **AI_SECURE** bit is set, the **NS_EMAIL** namespace provider will return results that were obtained with enhanced security to minimize possible spoofing. When this option is set in the *pHints* parameter of GetAddrInfoEx, the **NS_EMAIL** namespace provider returns only results that were obtained with enhanced security to minimize possible spoofing. This option is only supported on Windows Vista and later for the **NS_EMAIL** namespace.
AI_RETURN_PREFERRED_NAMES 0x010000	The address information is for a preferred names for publication with a specific namespace. When this option is set in the *pHints* parameter of GetAddrInfoEx, no name should be provided in the *pName* parameter and the **NS_EMAIL** namespace provider will return preferred names for publication. This option is only supported on Windows Vista and later for the **NS_EMAIL** namespace.
AI_FQDN 0x00020000	The fully qualified domain name is returned in the first **ai_fqdn** member. When this option is set in the *pHints* parameter of GetAddrInfoEx and a flat name (single label) is specified in the *pName* parameter, the fully qualified domain name that the name eventually resolved to will be returned. This option is supported on Windows 7, Windows Server 2008 R2, and later.

AI_FILESERVER 0x00040000	A hint to the namespace provider that the hostname being queried is being used in a file share scenario. The namespace provider may ignore this hint. This option is supported on Windows 7, Windows Server 2008 R2, and later.
AI_DISABLE_IDN_ENCODING 0x00080000	Disable the automatic International Domain Name encoding using Punycode in the name resolution functions called by the GetAddrInfoEx function. This option is supported on Windows 8, Windows Server 2012, and later.

`ai_family`

The address family.

The 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. Note that the values for the AF_ address family and PF_ protocol family constants are identical (for example, **AF_INET** and **PF_INET**), so either constant can be used.

The table below lists common values for the address family although many other values are possible.

VALUE	MEANING
AF_UNSPEC 0	The address family is unspecified.
AF_INET 2	The Internet Protocol version 4 (IPv4) address family.
AF_NETBIOS 17	The NetBIOS address family. This address family is only supported if a Windows Sockets provider for NetBIOS is installed.
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 a Bluetooth adapter is installed.

`ai_socktype`

The socket type. Possible values for the socket type are defined in the *Winsock2.h* include file.

The following table lists the possible values for the socket type supported for Windows Sockets 2:

VALUE	MEANING
SOCK_STREAM 1	Provides sequenced, reliable, two-way, connection-based byte streams with an OOB data transmission mechanism. Uses the Transmission Control Protocol (TCP) for the Internet address family (**AF_INET** or **AF_INET6**). If the **ai_family** member is **AF_IRDA**, then **SOCK_STREAM** is the only supported socket type.
SOCK_DGRAM 2	Supports datagrams, which are connectionless, unreliable buffers of a fixed (typically small) maximum length. Uses the User Datagram Protocol (UDP) for the Internet address family (**AF_INET** or **AF_INET6**).
SOCK_RAW 3	Provides a raw socket that allows an application to manipulate the next upper-layer protocol header. To manipulate the IPv4 header, the IP_HDRINCL socket option must be set on the socket. To manipulate the IPv6 header, the IPV6_HDRINCL socket option must be set on the socket.
SOCK_RDM 4	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 reliable multicast programming.
SOCK_SEQPACKET 5	Provides a pseudo-stream packet based on datagrams.

In Windows Sockets 2, new socket types were introduced. An application can dynamically discover the attributes of each available transport protocol through the WSAEnumProtocols function. So an application can determine the possible socket type and protocol options for an address family and use this information when specifying this parameter. Socket type definitions in the *Winsock2.h* and *Ws2def.h* header files will be periodically updated as new socket types, address families, and protocols are defined.

In Windows Sockets 1.1, the only possible socket types are **SOCK_DATAGRAM** and **SOCK_STREAM**.

`ai_protocol`

The protocol type. The possible options are specific to the address family and socket type specified. Possible values for the **ai_protocol** are defined in *Winsock2.h* and the *Wsrm.h* header files.

On the Windows SDK released for Windows Vista and later,, the organization of header files has changed and this member can be one of the values from 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 for **ai_protocol**, the caller does not wish to specify a protocol and the service provider will choose the **ai_protocol** to use. For protocols other than IPv4 and IPv6, set **ai_protocol** to zero.

The following table lists common values for the **ai_protocol** member although many other values are possible.

VALUE	MEANING

IPPROTO_TCP 6	The Transmission Control Protocol (TCP). This is a possible value when the **ai_family** member is **AF_INET** or **AF_INET6** and the **ai_socktype** member is **SOCK_STREAM**.
IPPROTO_UDP 17	The User Datagram Protocol (UDP). This is a possible value when the **ai_family** member is **AF_INET** or **AF_INET6** and the *type* parameter is **SOCK_DGRAM**.
IPPROTO_RM 113	The PGM protocol for reliable multicast. This is a possible value when the **ai_family** member is **AF_INET** and the **ai_socktype** member is **SOCK_RDM**. On the Windows SDK released for Windows Vista and later, this value is also called **IPPROTO_PGM**.

If the **ai_family** member is **AF_IRDA**, then the **ai_protocol** must be 0.

`ai_addrlen`

The length, in bytes, of the buffer pointed to by the **ai_addr** member.

`ai_canonname`

The canonical name for the host.

`ai_addr`

A pointer to a sockaddr structure. The **ai_addr** member in each returned **addrinfoex2** structure points to a filled-in socket address structure. The length, in bytes, of each returned **addrinfoex2** structure is specified in the **ai_addrlen** member.

`ai_blob`

A pointer to data that is used to return provider-specific namespace information that is associated with the name beyond a list of addresses. The length, in bytes, of the buffer pointed to by **ai_blob** must be specified in the **ai_bloblen** member.

`ai_bloblen`

The length, in bytes, of the **ai_blob** member.

`ai_provider`

A pointer to the GUID of a specific namespace provider.

`ai_next`

A pointer to the next structure in a linked list. This parameter is set to **NULL** in the last **addrinfoex2** structure of a linked list.

`ai_version`

The version number of this structure. The value currently used for this version of the structure is 2.

`ai_fqdn`

The fully qualified domain name for the host.

Remarks

The **addrinfoex2** structure is supported on Windows 8 and Windows Server 2012

The **addrinfoex2** structure is used by the GetAddrInfoEx function to hold host address information when both the **AI_FQDN** and **AI_CANONNAME** bits are set in the **ai_flags** member of the optional addrinfoex structure provided in the *hints* parameter to the **GetAddrInfoEx** function. The **addrinfoex2** structure is an enhanced version of the **addrinfoex** structure that can return both the canonical name and the fully qualified domain name for the host. The extra structure members are for a version number of the structure and the fully qualified domain name for the host.

The **addrinfoex2** structure used with GetAddrInfoEx function is an enhanced version of the addrinfo and addrinfoW structures used with the getaddrinfo and GetAddrInfoW functions. The **GetAddrInfoEx** function allows specifying the namespace provider to resolve the query. For use with the IPv6 and IPv4 protocol, name resolution can be by the Domain Name System (DNS), a local *hosts* file, an email provider (the **NS_EMAIL** namespace), or by other naming mechanisms.

The blob data in tha **ai_blob** member is used to return additional provider-specific namespace information associated with a name. The format of data in the **ai_blob** member is specific to a particular namespace provider. Currently, blob data is used by the **NS_EMAIL** namespace provider to supply additional information.

When UNICODE or _UNICODE is defined, **addrinfoex2** is defined to **addrinfoex2W**, the Unicode version of this structure. The string parameters are defined to the **PWSTR** data type and the **addrinfoex2W** structure is used.

When UNICODE or _UNICODE is not defined, **addrinfoex2** is defined to **addrinfoex2A**, the ANSI version of this structure. The string parameters are of the **char *** data type and the **addrinfoex2A** structure is used.

Upon a successful call to GetAddrInfoEx, a linked list of **addrinfoex2** structures is returned in the *ppResult* parameter passed to the **GetAddrInfoEx** function. The list can be processed by following the pointer provided in the **ai_next** member of each returned **addrinfoex2** structure until a **NULL** pointer is encountered. In each returned **addrinfoex2** structure, the **ai_family**, **ai_socktype**, and **ai_protocol** members correspond to respective arguments in a socket or WSASocket function call. Also, the **ai_addr** member in each returned **addrinfoex2** structure points to a filled-in socket address structure, the length of which is specified in its **ai_addrlen** member.

Requirements

Minimum supported client	Windows 8 [desktop apps only]
Minimum supported server	Windows Server 2012 [desktop apps only]
Header	ws2def.h

See also

GetAddrInfoEx

addrinfo

addrinfoW

addrinfoex

ADDRINFOEX2W structure (ws2def.h)

10/22/2022 • 9 minutes to read • Edit Online

The **addrinfoex2** structure is used by the GetAddrInfoEx function to hold host address information when both a canonical name and a fully qualified domain name have been requested.

Syntax

```
typedef struct addrinfoex2W {
  int                 ai_flags;
  int                 ai_family;
  int                 ai_socktype;
  int                 ai_protocol;
  size_t              ai_addrlen;
  PWSTR               ai_canonname;
  struct sockaddr     *ai_addr;
  void                *ai_blob;
  size_t              ai_bloblen;
  LPGUID              ai_provider;
  struct addrinfoex2W *ai_next;
  int                 ai_version;
  PWSTR               ai_fqdn;
} ADDRINFOEX2W, *PADDRINFOEX2W, *LPADDRINFOEX2W;
```

Members

`ai_flags`

Flags that indicate options used in the GetAddrInfoEx function.

Supported values for the **ai_flags** member are defined in the *Winsock2.h* include file and can be a combination of the following options.

VALUE	MEANING
AI_PASSIVE 0x01	The socket address will be used in a call to the bind function.
AI_CANONNAME 0x02	The canonical name is returned in the first **ai_canonname** member.
AI_NUMERICHOST 0x04	The *nodename* parameter passed to the GetAddrInfoEx function must be a numeric string.
AI_ALL 0x0100	If this bit is set, a request is made for IPv6 addresses and IPv4 addresses with **AI_V4MAPPED**. This option is supported on Windows Vista and later.

AI_ADDRCONFIG 0x0400	The GetAddrInfoEx will resolve only if a global address is configured. The IPv6 and IPv4 loopback address is not considered a valid global address. This option is supported on Windows Vista and later.
AI_V4MAPPED 0x0800	If the GetAddrInfoEx request for an IPv6 addresses fails, a name service request is made for IPv4 addresses and these addresses are converted to IPv4-mapped IPv6 address format. This option is supported on Windows Vista and later.
AI_NON_AUTHORITATIVE 0x04000	The address information is from non-authoritative results. When this option is set in the *pHints* parameter of GetAddrInfoEx, the **NS_EMAIL** namespace provider returns both authoritative and non-authoritative results. If this option is not set, then only authoritative results are returned. This option is only supported on Windows Vista and later for the **NS_EMAIL** namespace.
AI_SECURE 0x08000	The address information is from a secure channel. If the **AI_SECURE** bit is set, the **NS_EMAIL** namespace provider will return results that were obtained with enhanced security to minimize possible spoofing. When this option is set in the *pHints* parameter of GetAddrInfoEx, the **NS_EMAIL** namespace provider returns only results that were obtained with enhanced security to minimize possible spoofing. This option is only supported on Windows Vista and later for the **NS_EMAIL** namespace.
AI_RETURN_PREFERRED_NAMES 0x010000	The address information is for a preferred names for publication with a specific namespace. When this option is set in the *pHints* parameter of GetAddrInfoEx, no name should be provided in the *pName* parameter and the **NS_EMAIL** namespace provider will return preferred names for publication. This option is only supported on Windows Vista and later for the **NS_EMAIL** namespace.
AI_FQDN 0x00020000	The fully qualified domain name is returned in the first **ai_fqdn** member. When this option is set in the *pHints* parameter of GetAddrInfoEx and a flat name (single label) is specified in the *pName* parameter, the fully qualified domain name that the name eventually resolved to will be returned. This option is supported on Windows 7, Windows Server 2008 R2, and later.

AI_FILESERVER 0x00040000	A hint to the namespace provider that the hostname being queried is being used in a file share scenario. The namespace provider may ignore this hint. This option is supported on Windows 7, Windows Server 2008 R2, and later.
AI_DISABLE_IDN_ENCODING 0x00080000	Disable the automatic International Domain Name encoding using Punycode in the name resolution functions called by the GetAddrInfoEx function. This option is supported on Windows 8, Windows Server 2012, and later.

`ai_family`

The address family.

The 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. Note that the values for the AF_ address family and PF_ protocol family constants are identical (for example, **AF_INET** and **PF_INET**), so either constant can be used.

The table below lists common values for the address family although many other values are possible.

VALUE	MEANING
AF_UNSPEC 0	The address family is unspecified.
AF_INET 2	The Internet Protocol version 4 (IPv4) address family.
AF_NETBIOS 17	The NetBIOS address family. This address family is only supported if a Windows Sockets provider for NetBIOS is installed.
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 a Bluetooth adapter is installed.

`ai_socktype`

The socket type. Possible values for the socket type are defined in the *Winsock2.h* include file.

The following table lists the possible values for the socket type supported for Windows Sockets 2:

VALUE	MEANING
SOCK_STREAM 1	Provides sequenced, reliable, two-way, connection-based byte streams with an OOB data transmission mechanism. Uses the Transmission Control Protocol (TCP) for the Internet address family (**AF_INET** or **AF_INET6**). If the **ai_family** member is **AF_IRDA**, then **SOCK_STREAM** is the only supported socket type.
SOCK_DGRAM 2	Supports datagrams, which are connectionless, unreliable buffers of a fixed (typically small) maximum length. Uses the User Datagram Protocol (UDP) for the Internet address family (**AF_INET** or **AF_INET6**).
SOCK_RAW 3	Provides a raw socket that allows an application to manipulate the next upper-layer protocol header. To manipulate the IPv4 header, the IP_HDRINCL socket option must be set on the socket. To manipulate the IPv6 header, the IPV6_HDRINCL socket option must be set on the socket.
SOCK_RDM 4	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 reliable multicast programming.
SOCK_SEQPACKET 5	Provides a pseudo-stream packet based on datagrams.

In Windows Sockets 2, new socket types were introduced. An application can dynamically discover the attributes of each available transport protocol through the WSAEnumProtocols function. So an application can determine the possible socket type and protocol options for an address family and use this information when specifying this parameter. Socket type definitions in the *Winsock2.h* and *Ws2def.h* header files will be periodically updated as new socket types, address families, and protocols are defined.

In Windows Sockets 1.1, the only possible socket types are **SOCK_DATAGRAM** and **SOCK_STREAM**.

`ai_protocol`

The protocol type. The possible options are specific to the address family and socket type specified. Possible values for the **ai_protocol** are defined in *Winsock2.h* and the *Wsrm.h* header files.

On the Windows SDK released for Windows Vista and later,, the organization of header files has changed and this member can be one of the values from 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 for **ai_protocol**, the caller does not wish to specify a protocol and the service provider will choose the **ai_protocol** to use. For protocols other than IPv4 and IPv6, set **ai_protocol** to zero.

The following table lists common values for the **ai_protocol** member although many other values are possible.

VALUE	MEANING

IPPROTO_TCP 6	The Transmission Control Protocol (TCP). This is a possible value when the **ai_family** member is **AF_INET** or **AF_INET6** and the **ai_socktype** member is **SOCK_STREAM**.
IPPROTO_UDP 17	The User Datagram Protocol (UDP). This is a possible value when the **ai_family** member is **AF_INET** or **AF_INET6** and the *type* parameter is **SOCK_DGRAM**.
IPPROTO_RM 113	The PGM protocol for reliable multicast. This is a possible value when the **ai_family** member is **AF_INET** and the **ai_socktype** member is **SOCK_RDM**. On the Windows SDK released for Windows Vista and later, this value is also called **IPPROTO_PGM**.

If the **ai_family** member is **AF_IRDA**, then the **ai_protocol** must be 0.

`ai_addrlen`

The length, in bytes, of the buffer pointed to by the **ai_addr** member.

`ai_canonname`

The canonical name for the host.

`ai_addr`

A pointer to a sockaddr structure. The **ai_addr** member in each returned **addrinfoex2** structure points to a filled-in socket address structure. The length, in bytes, of each returned **addrinfoex2** structure is specified in the **ai_addrlen** member.

`ai_blob`

A pointer to data that is used to return provider-specific namespace information that is associated with the name beyond a list of addresses. The length, in bytes, of the buffer pointed to by **ai_blob** must be specified in the **ai_bloblen** member.

`ai_bloblen`

The length, in bytes, of the **ai_blob** member.

`ai_provider`

A pointer to the GUID of a specific namespace provider.

`ai_next`

A pointer to the next structure in a linked list. This parameter is set to **NULL** in the last **addrinfoex2** structure of a linked list.

`ai_version`

The version number of this structure. The value currently used for this version of the structure is 2.

`ai_fqdn`

The fully qualified domain name for the host.

Remarks

The **addrinfoex2** structure is supported on Windows 8 and Windows Server 2012

The **addrinfoex2** structure is used by the GetAddrInfoEx function to hold host address information when both the **AI_FQDN** and **AI_CANONNAME** bits are set in the **ai_flags** member of the optional addrinfoex structure provided in the *hints* parameter to the **GetAddrInfoEx** function. The **addrinfoex2** structure is an enhanced version of the **addrinfoex** structure that can return both the canonical name and the fully qualified domain name for the host. The extra structure members are for a version number of the structure and the fully qualified domain name for the host.

The **addrinfoex2** structure used with GetAddrInfoEx function is an enhanced version of the addrinfo and addrinfoW structures used with the getaddrinfo and GetAddrInfoW functions. The **GetAddrInfoEx** function allows specifying the namespace provider to resolve the query. For use with the IPv6 and IPv4 protocol, name resolution can be by the Domain Name System (DNS), a local *hosts* file, an email provider (the **NS_EMAIL** namespace), or by other naming mechanisms.

The blob data in tha **ai_blob** member is used to return additional provider-specific namespace information associated with a name. The format of data in the **ai_blob** member is specific to a particular namespace provider. Currently, blob data is used by the **NS_EMAIL** namespace provider to supply additional information.

When UNICODE or _UNICODE is defined, **addrinfoex2** is defined to **addrinfoex2W**, the Unicode version of this structure. The string parameters are defined to the **PWSTR** data type and the **addrinfoex2W** structure is used.

When UNICODE or _UNICODE is not defined, **addrinfoex2** is defined to **addrinfoex2A**, the ANSI version of this structure. The string parameters are of the **char *** data type and the **addrinfoex2A** structure is used.

Upon a successful call to GetAddrInfoEx, a linked list of **addrinfoex2** structures is returned in the *ppResult* parameter passed to the **GetAddrInfoEx** function. The list can be processed by following the pointer provided in the **ai_next** member of each returned **addrinfoex2** structure until a **NULL** pointer is encountered. In each returned **addrinfoex2** structure, the **ai_family**, **ai_socktype**, and **ai_protocol** members correspond to respective arguments in a socket or WSASocket function call. Also, the **ai_addr** member in each returned **addrinfoex2** structure points to a filled-in socket address structure, the length of which is specified in its **ai_addrlen** member.

Requirements

Minimum supported client	Windows 8 [desktop apps only]
Minimum supported server	Windows Server 2012 [desktop apps only]
Header	ws2def.h

See also

GetAddrInfoEx

addrinfo

addrinfoW

addrinfoex

ADDRINFOEX3 structure (ws2def.h)

10/22/2022 • 9 minutes to read • Edit Online

The **addrinfoex3** structure is used by the GetAddrInfoEx function to hold host address information when a specific network interface has been requested.

Syntax

```
typedef struct addrinfoex3 {
  int                ai_flags;
  int                ai_family;
  int                ai_socktype;
  int                ai_protocol;
  size_t             ai_addrlen;
  PWSTR              ai_canonname;
  struct sockaddr    *ai_addr;
  void               *ai_blob;
  size_t             ai_bloblen;
  LPGUID             ai_provider;
  struct addrinfoex3 *ai_next;
  int                ai_version;
  PWSTR              ai_fqdn;
  int                ai_interfaceindex;
} ADDRINFOEX3, *PADDRINFOEX3, *LPADDRINFOEX3;
```

Members

`ai_flags`

Flags that indicate options used in the GetAddrInfoEx function.

Supported values for the **ai_flags** member are defined in the *Winsock2.h* include file and can be a combination of the following options.

VALUE	MEANING
AI_PASSIVE 0x01	The socket address will be used in a call to the bind function.
AI_CANONNAME 0x02	The canonical name is returned in the first **ai_canonname** member.
AI_NUMERICHOST 0x04	The *nodename* parameter passed to the GetAddrInfoEx function must be a numeric string.
AI_ALL 0x0100	If this bit is set, a request is made for IPv6 addresses and IPv4 addresses with **AI_V4MAPPED**. This option is supported on Windows Vista and later.

AI_ADDRCONFIG 0x0400	The GetAddrInfoEx will resolve only if a global address is configured. The IPv6 and IPv4 loopback address is not considered a valid global address. This option is supported on Windows Vista and later.
AI_V4MAPPED 0x0800	If the GetAddrInfoEx request for an IPv6 addresses fails, a name service request is made for IPv4 addresses and these addresses are converted to IPv4-mapped IPv6 address format. This option is supported on Windows Vista and later.
AI_NON_AUTHORITATIVE 0x04000	The address information is from non-authoritative results. When this option is set in the *pHints* parameter of GetAddrInfoEx, the **NS_EMAIL** namespace provider returns both authoritative and non-authoritative results. If this option is not set, then only authoritative results are returned. This option is only supported on Windows Vista and later for the **NS_EMAIL** namespace.
AI_SECURE 0x08000	The address information is from a secure channel. If the **AI_SECURE** bit is set, the **NS_EMAIL** namespace provider will return results that were obtained with enhanced security to minimize possible spoofing. When this option is set in the *pHints* parameter of GetAddrInfoEx, the **NS_EMAIL** namespace provider returns only results that were obtained with enhanced security to minimize possible spoofing. This option is only supported on Windows Vista and later for the **NS_EMAIL** namespace.
AI_RETURN_PREFERRED_NAMES 0x010000	The address information is for a preferred names for publication with a specific namespace. When this option is set in the *pHints* parameter of GetAddrInfoEx, no name should be provided in the *pName* parameter and the **NS_EMAIL** namespace provider will return preferred names for publication. This option is only supported on Windows Vista and later for the **NS_EMAIL** namespace.
AI_FQDN 0x00020000	The fully qualified domain name is returned in the first **ai_fqdn** member. When this option is set in the *pHints* parameter of GetAddrInfoEx and a flat name (single label) is specified in the *pName* parameter, the fully qualified domain name that the name eventually resolved to will be returned. This option is supported on Windows 7, Windows Server 2008 R2, and later.

AI_FILESERVER 0x00040000	A hint to the namespace provider that the hostname being queried is being used in a file share scenario. The namespace provider may ignore this hint. This option is supported on Windows 7, Windows Server 2008 R2, and later.
AI_DISABLE_IDN_ENCODING 0x00080000	Disable the automatic International Domain Name encoding using Punycode in the name resolution functions called by the GetAddrInfoEx function. This option is supported on Windows 8, Windows Server 2012, and later.
AI_EXTENDED 0x80000000	Indicates that the current object is extended: that is, an addrinfoex2 or greater. This option is supported on Windows 8.1, Windows Server 2012 R2, and later.

`ai_family`

The address family.

The 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. Note that the values for the AF_ address family and PF_ protocol family constants are identical (for example, **AF_INET** and **PF_INET**), so either constant can be used.

The table below lists common values for the address family although many other values are possible.

VALUE	MEANING
AF_UNSPEC 0	The address family is unspecified.
AF_INET 2	The Internet Protocol version 4 (IPv4) address family.
AF_NETBIOS 17	The NetBIOS address family. This address family is only supported if a Windows Sockets provider for NetBIOS is installed.
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 a Bluetooth adapter is installed.

`ai_socktype`

The socket type. Possible values for the socket type are defined in the *Winsock2.h* include file.

The following table lists the possible values for the socket type supported for Windows Sockets 2:

VALUE	MEANING
SOCK_STREAM 1	Provides sequenced, reliable, two-way, connection-based byte streams with an OOB data transmission mechanism. Uses the Transmission Control Protocol (TCP) for the Internet address family (**AF_INET** or **AF_INET6**). If the **ai_family** member is **AF_IRDA**, then **SOCK_STREAM** is the only supported socket type.
SOCK_DGRAM 2	Supports datagrams, which are connectionless, unreliable buffers of a fixed (typically small) maximum length. Uses the User Datagram Protocol (UDP) for the Internet address family (**AF_INET** or **AF_INET6**).
SOCK_RAW 3	Provides a raw socket that allows an application to manipulate the next upper-layer protocol header. To manipulate the IPv4 header, the IP_HDRINCL socket option must be set on the socket. To manipulate the IPv6 header, the IPV6_HDRINCL socket option must be set on the socket.
SOCK_RDM 4	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 reliable multicast programming.
SOCK_SEQPACKET 5	Provides a pseudo-stream packet based on datagrams.

In Windows Sockets 2, new socket types were introduced. An application can dynamically discover the attributes of each available transport protocol through the WSAEnumProtocols function. So an application can determine the possible socket type and protocol options for an address family and use this information when specifying this parameter. Socket type definitions in the *Winsock2.h* and *Ws2def.h* header files will be periodically updated as new socket types, address families, and protocols are defined.

In Windows Sockets 1.1, the only possible socket types are **SOCK_DATAGRAM** and **SOCK_STREAM**.

`ai_protocol`

The protocol type. The possible options are specific to the address family and socket type specified. Possible values for the **ai_protocol** are defined in *Winsock2.h* and the *Wsrm.h* header files.

On the Windows SDK released for Windows Vista and later,, the organization of header files has changed and this member can be one of the values from 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 for **ai_protocol**, the caller does not wish to specify a protocol and the service provider

will choose the **ai_protocol** to use. For protocols other than IPv4 and IPv6, set **ai_protocol** to zero.

The following table lists common values for the **ai_protocol** member although many other values are possible.

VALUE	MEANING
IPPROTO_TCP 6	The Transmission Control Protocol (TCP). This is a possible value when the **ai_family** member is **AF_INET** or **AF_INET6** and the **ai_socktype** member is **SOCK_STREAM**.
IPPROTO_UDP 17	The User Datagram Protocol (UDP). This is a possible value when the **ai_family** member is **AF_INET** or **AF_INET6** and the *type* parameter is **SOCK_DGRAM**.
IPPROTO_RM 113	The PGM protocol for reliable multicast. This is a possible value when the **ai_family** member is **AF_INET** and the **ai_socktype** member is **SOCK_RDM**. On the Windows SDK released for Windows Vista and later, this value is also called **IPPROTO_PGM**.

If the **ai_family** member is **AF_IRDA**, then the **ai_protocol** must be 0.

`ai_addrlen`

The length, in bytes, of the buffer pointed to by the **ai_addr** member.

`ai_canonname`

The canonical name for the host.

`ai_addr`

A pointer to a sockaddr structure. The **ai_addr** member in each returned **addrinfoex3** structure points to a filled-in socket address structure. The length, in bytes, of each returned **addrinfoex3** structure is specified in the **ai_addrlen** member.

`ai_blob`

A pointer to data that is used to return provider-specific namespace information that is associated with the name beyond a list of addresses. The length, in bytes, of the buffer pointed to by **ai_blob** must be specified in the **ai_bloblen** member.

`ai_bloblen`

The length, in bytes, of the **ai_blob** member.

`ai_provider`

A pointer to the GUID of a specific namespace provider.

`ai_next`

A pointer to the next structure in a linked list. This parameter is set to **NULL** in the last **addrinfoex3** structure of a linked list.

`ai_version`

The version number of this structure. The value currently used for this version of the structure is 3.

`ai_fqdn`

The fully qualified domain name for the host.

`ai_interfaceindex`

The interface index, as defined by the IP_ADAPTER_ADDRESSES.*IfIndex* property returned in GetAdaptersAddresses.

Remarks

The **addrinfoex3** structure is supported on Windows 8.1 and Windows Server 2012 R2

The **addrinfoex3** structure is used by the GetAddrInfoEx function to hold host address information when the AI_**EXTENDED** | **AI_FQDN** | **AI_CANONNAME** bits are set in the **addrinfoex3**.**ai_flags** member passed in through the **GetAddrInfoEx**.*hints* parameter. You must also set the **addrinfoex3**.**ai_interfaceindex** member to the interface you wish to retrieve information for, and **ai_version** to 3.

The **addrinfoex3** structure is an enhanced version of the addrinfoex structure that can return the canonical name and the fully qualified domain name for the host. In turn, **addrinfoex** is an enhanced version of the addrinfo and addrinfoW structures used with the getaddrinfo and GetAddrInfoW functions. The GetAddrInfoEx function allows specifying the namespace provider to resolve the query. For use with the IPv6 and IPv4 protocol, name resolution can be by the Domain Name System (DNS), a local *hosts* file, an email provider (the **NS_EMAIL** namespace), or by other naming mechanisms.

The blob data in the **ai_blob** member is used to return additional provider-specific namespace information associated with a name. The format of data in the **ai_blob** member is specific to a particular namespace provider. Currently, blob data is used by the **NS_EMAIL** namespace provider to supply additional information.

When UNICODE or _UNICODE is defined, **addrinfoex3** is defined to **addrinfoex3W**, the Unicode version of this structure. The string parameters are defined to the **PWSTR** data type and the **addrinfoex3W** structure is used.

When UNICODE or _UNICODE is not defined, **addrinfoex3** is defined to **addrinfoex3A**, the ANSI version of this structure. The string parameters are of the **char *** data type and the **addrinfoex3A** structure is used.

Upon a successful call to GetAddrInfoEx, a linked list of **addrinfoex3** structures is returned in the *ppResult* parameter passed to the **GetAddrInfoEx** function. The list can be processed by following the pointer provided in the **ai_next** member of each returned **addrinfoex3** structure until a **NULL** pointer is encountered. In each returned **addrinfoex3** structure, the **ai_family**, **ai_socktype**, and **ai_protocol** members correspond to respective arguments in a socket or WSASocket function call. Also, the **ai_addr** member in each returned **addrinfoex3** structure points to a filled-in socket address structure, the length of which is specified in its **ai_addrlen** member.

Requirements

Minimum supported client	Windows 8 [desktop apps only]
Minimum supported server	Windows Server 2012 [desktop apps only]
Header	ws2def.h

See also

GetAddrInfoEx

addrinfo

addrinfoW

addrinfoex

ADDRINFOEX4 structure (ws2def.h)

10/22/2022 • 11 minutes to read • Edit Online

The **addrinfoex4** structure is used by the GetAddrInfoEx function to hold host address information when a specific network interface has been requested.

Syntax

```
typedef struct addrinfoex4 {
  int                ai_flags;
  int                ai_family;
  int                ai_socktype;
  int                ai_protocol;
  size_t             ai_addrlen;
  PWSTR              ai_canonname;
  struct sockaddr    *ai_addr;
  void               *ai_blob;
  size_t             ai_bloblen;
  GUID               *ai_provider;
  struct addrinfoex4 *ai_next;
  int                ai_version;
  PWSTR              ai_fqdn;
  int                ai_interfaceindex;
  HANDLE             ai_resolutionhandle;
} ADDRINFOEX4, *PADDRINFOEX4, *LPADDRINFOEX4;
```

Members

`ai_flags`

Flags that indicate options used in the GetAddrInfoEx function.

Supported values for the **ai_flags** member are defined in the *Winsock2.h* include file and can be a combination of the following options.

VALUE	MEANING
AI_PASSIVE 0x01	The socket address will be used in a call to the bind function.
AI_CANONNAME 0x02	The canonical name is returned in the first **ai_canonname** member.
AI_NUMERICHOST 0x04	The *nodename* parameter passed to the GetAddrInfoEx function must be a numeric string.
AI_ALL 0x0100	If this bit is set, a request is made for IPv6 addresses and IPv4 addresses with **AI_V4MAPPED**. This option is supported on Windows Vista and later.

AI_ADDRCONFIG 0x0400	The GetAddrInfoEx will resolve only if a global address is configured. The IPv6 and IPv4 loopback address is not considered a valid global address. This option is supported on Windows Vista and later.
AI_V4MAPPED 0x0800	If the GetAddrInfoEx request for an IPv6 addresses fails, a name service request is made for IPv4 addresses and these addresses are converted to IPv4-mapped IPv6 address format. This option is supported on Windows Vista and later.
AI_NON_AUTHORITATIVE 0x04000	The address information is from non-authoritative results. When this option is set in the *pHints* parameter of GetAddrInfoEx, the **NS_EMAIL** namespace provider returns both authoritative and non-authoritative results. If this option is not set, then only authoritative results are returned. This option is only supported on Windows Vista and later for the **NS_EMAIL** namespace.
AI_SECURE 0x08000	The address information is from a secure channel. If the **AI_SECURE** bit is set, the **NS_EMAIL** namespace provider will return results that were obtained with enhanced security to minimize possible spoofing. When this option is set in the *pHints* parameter of GetAddrInfoEx, the **NS_EMAIL** namespace provider returns only results that were obtained with enhanced security to minimize possible spoofing. This option is only supported on Windows Vista and later for the **NS_EMAIL** namespace.
AI_RETURN_PREFERRED_NAMES 0x010000	The address information is for a preferred names for publication with a specific namespace. When this option is set in the *pHints* parameter of GetAddrInfoEx, no name should be provided in the *pName* parameter and the **NS_EMAIL** namespace provider will return preferred names for publication. This option is only supported on Windows Vista and later for the **NS_EMAIL** namespace.
AI_FQDN 0x00020000	The fully qualified domain name is returned in the first **ai_fqdn** member. When this option is set in the *pHints* parameter of GetAddrInfoEx and a flat name (single label) is specified in the *pName* parameter, the fully qualified domain name that the name eventually resolved to will be returned. This option is supported on Windows 7, Windows Server 2008 R2, and later.

AI_FILESERVER 0x00040000	A hint to the namespace provider that the hostname being queried is being used in a file share scenario. The namespace provider may ignore this hint. This option is supported on Windows 7, Windows Server 2008 R2, and later.
AI_DISABLE_IDN_ENCODING 0x00080000	Disable the automatic International Domain Name encoding using Punycode in the name resolution functions called by the GetAddrInfoEx function. This option is supported on Windows 8, Windows Server 2012, and later.
AI_EXTENDED 0x80000000	Indicates that the current object is extended: that is, an addrinfoex2 or greater. This option is supported on Windows 8.1, Windows Server 2012 R2, and later.
AI_RESOLUTION_HANDLE 0x40000000	A resolution handle is returned in the **ai_resolutionhandle** member. This option is supported on Windows 10, Windows Server 2016, and later.

`ai_family`

The address family.

The 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. Note that the values for the AF_ address family and PF_ protocol family constants are identical (for example, **AF_INET** and **PF_INET**), so either constant can be used.

The table below lists common values for the address family although many other values are possible.

VALUE	MEANING
AF_UNSPEC 0	The address family is unspecified.
AF_INET 2	The Internet Protocol version 4 (IPv4) address family.
AF_NETBIOS 17	The NetBIOS address family. This address family is only supported if a Windows Sockets provider for NetBIOS is installed.

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 a Bluetooth adapter is installed.

`ai_socktype`

The socket type. Possible values for the socket type are defined in the *Winsock2.h* include file.

The following table lists the possible values for the socket type supported for Windows Sockets 2:

VALUE	MEANING
SOCK_STREAM 1	Provides sequenced, reliable, two-way, connection-based byte streams with an OOB data transmission mechanism. Uses the Transmission Control Protocol (TCP) for the Internet address family (**AF_INET** or **AF_INET6**). If the **ai_family** member is **AF_IRDA**, then **SOCK_STREAM** is the only supported socket type.
SOCK_DGRAM 2	Supports datagrams, which are connectionless, unreliable buffers of a fixed (typically small) maximum length. Uses the User Datagram Protocol (UDP) for the Internet address family (**AF_INET** or **AF_INET6**).
SOCK_RAW 3	Provides a raw socket that allows an application to manipulate the next upper-layer protocol header. To manipulate the IPv4 header, the IP_HDRINCL socket option must be set on the socket. To manipulate the IPv6 header, the IPV6_HDRINCL socket option must be set on the socket.
SOCK_RDM 4	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 reliable multicast programming.
SOCK_SEQPACKET 5	Provides a pseudo-stream packet based on datagrams.

In Windows Sockets 2, new socket types were introduced. An application can dynamically discover the attributes of each available transport protocol through the WSAEnumProtocols function. So an application can determine the possible socket type and protocol options for an address family and use this information when specifying this parameter. Socket type definitions in the *Winsock2.h* and *Ws2def.h* header files will be periodically updated as new socket types, address families, and protocols are defined.

In Windows Sockets 1.1, the only possible socket types are **SOCK_DATAGRAM** and **SOCK_STREAM**.

`ai_protocol`

The protocol type. The possible options are specific to the address family and socket type specified. Possible values for the **ai_protocol** are defined in *Winsock2.h* and the *Wsrm.h* header files.

On the Windows SDK released for Windows Vista and later,, the organization of header files has changed and this member can be one of the values from 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 for **ai_protocol**, the caller does not wish to specify a protocol and the service provider will choose the **ai_protocol** to use. For protocols other than IPv4 and IPv6, set **ai_protocol** to zero.

The following table lists common values for the **ai_protocol** member although many other values are possible.

VALUE	MEANING
IPPROTO_TCP 6	The Transmission Control Protocol (TCP). This is a possible value when the **ai_family** member is **AF_INET** or **AF_INET6** and the **ai_socktype** member is **SOCK_STREAM**.
IPPROTO_UDP 17	The User Datagram Protocol (UDP). This is a possible value when the **ai_family** member is **AF_INET** or **AF_INET6** and the *type* parameter is **SOCK_DGRAM**.
IPPROTO_RM 113	The PGM protocol for reliable multicast. This is a possible value when the **ai_family** member is **AF_INET** and the **ai_socktype** member is **SOCK_RDM**. On the Windows SDK released for Windows Vista and later, this value is also called **IPPROTO_PGM**.

If the **ai_family** member is **AF_IRDA**, then the **ai_protocol** must be 0.

`ai_addrlen`

The length, in bytes, of the buffer pointed to by the **ai_addr** member.

`ai_canonname`

The canonical name for the host.

`ai_addr`

A pointer to a sockaddr structure. The **ai_addr** member in each returned **addrinfoex4** structure points to a filled-in socket address structure. The length, in bytes, of each returned **addrinfoex4** structure is specified in the **ai_addrlen** member.

`ai_blob`

A pointer to data that is used to return provider-specific namespace information that is associated with the name beyond a list of addresses. The length, in bytes, of the buffer pointed to by **ai_blob** must be specified in the **ai_bloblen** member.

`ai_bloblen`

The length, in bytes, of the **ai_blob** member.

`ai_provider`

A pointer to the GUID of a specific namespace provider.

`ai_next`

A pointer to the next structure in a linked list. This parameter is set to **NULL** in the last **addrinfoex4** structure of a linked list.

`ai_version`

The version number of this structure. The value currently used for this version of the structure is 4.

`ai_fqdn`

The fully qualified domain name for the host.

`ai_interfaceindex`

The interface index, as defined by the IP_ADAPTER_ADDRESSES.*IfIndex* property returned in GetAdaptersAddresses.

`ai_resolutionhandle`

Handle pointing to the fully qualified domain name for the host.

Remarks

The **addrinfoex4** structure is supported on Windows 10 and Windows Server 2016

The **addrinfoex4** structure is used by the GetAddrInfoEx function to hold host address information when the **AI_EXTENDED | AI_FQDN | AI_CANONNAME | AI_RESOLUTION_HANDLE** bits are set in the **addrinfoex4.ai_flags** member passed in through the GetAddrInfoEx.*hints* parameter.

The **addrinfoex4** structure is an enhanced version of the addrinfoex structure that can return the canonical name, the fully qualified domain name for the host, and a handle to the fully qualified domain name. In turn, GetAddrInfoEx is an enhanced version of the addrinfo and addrinfoW structures used with the getaddrinfo and GetAddrInfoW functions. The **GetAddrInfoEx** function allows specifying the namespace provider to resolve the query. For use with the IPv6 and IPv4 protocol, name resolution can be by the Domain Name System (DNS), a local *hosts* file, an email provider (the **NS_EMAIL** namespace), or by other naming mechanisms.

The blob data in tha **ai_blob** member is used to return additional provider-specific namespace information associated with a name. The format of data in the **ai_blob** member is specific to a particular namespace provider. Currently, blob data is used by the **NS_EMAIL** namespace provider to supply additional information.

When UNICODE or _UNICODE is defined, **addrinfoex4** is defined to **addrinfoex4W**, the Unicode version of this structure. The string parameters are defined to the **PWSTR** data type and the **addrinfoex4W** structure is used.

When UNICODE or _UNICODE is not defined, **addrinfoex4** is defined to **addrinfoex4A**, the ANSI version of this structure. The string parameters are of the **char *** data type and the **addrinfoex4A** structure is used.

Upon a successful call to GetAddrInfoEx, a linked list of **addrinfoex4** structures is returned in the *ppResult* parameter passed to the **GetAddrInfoEx** function. The list can be processed by following the pointer provided in the **ai_next** member of each returned **addrinfoex4** structure until a **NULL** pointer is encountered. In each returned **addrinfoex4** structure, the **ai_family**, **ai_socktype**, and **ai_protocol** members correspond to respective arguments in a socket or WSASocket function call. Also, the **ai_addr** member in each returned **addrinfoex4** structure points to a filled-in socket address structure, the length of which is specified in its **ai_addrlen** member.

Examples

The following code describes making a call to GetAddrInfoEx with an **addrinfoex4** structure to retrieve the handle to a FQDN. the sample then call WSAIoctl with the ASSOCIATE_NAMERES_CONTEXT_INPUT structure.

```
//
// Connect to a server using its IPv4 addresses
//

VOID
ConnectServer(
    PCWSTR server)
{
    int iResult;
    PADDRINFOEX4 pResult = NULL;
    ADDRINFOEX3 hints = { 0 };
    PADDRINFOEX4 pCur = NULL;
    WSADATA wsaData;
    SOCKET connectSocket = INVALID_SOCKET;
    ULONG bytesReturned = 0;
    ASSOCIATE_NAMERES_CONTEXT_INPUT input = { 0 };
    SOCKADDR_IN clientService;
    wchar_t ipstringbuffer[46];
    String string;
    DWORD dwRetval;
    //
    //  Initialize Winsock
    //
    iResult = WSAStartup(
        MAKEWORD(2, 2),
        &wsaData);
    if (iResult != 0) {
        printf("WSAStartup failed: %d\n", iResult);
        goto Exit;
    }

    //
    // Create a SOCKET for connection
    //
    connectSocket = socket(
        AF_UNSPEC,
        SOCK_STREAM,
        IPPROTO_TCP);
    if (connectSocket == INVALID_SOCKET)
    {
        printf("socket failed: %d\n", WSAGetLastError());
        goto Exit;
    }

    //
    // Do name resolution
    //

    hints.ai_family = AF_INET;
    hints.ai_socktype = SOCK_STREAM;
    hints.ai_flags = AI_EXTENDED | AI_FQDN | AI_CANONNAME | AI_RESOLUTION_HANDLE;
    hints.ai_version = ADDRINFOEX_VERSION_4;

    dwRetval = GetAddrInfoExW(
        server,
        NULL,
        NS_DNS,
        NULL,
        (const ADDRINFOEXW*)&hints,
        (PADDRINFOEXW*)&pResult,
        NULL,
        NULL,
        NULL, NULL);
    if (dwRetval != 0) {
        printf("GetAddrInfoEx failed with error: %d\n", dwRetval);
        goto Exit;
    }
    input.TransportSettingId.Guid = ASSOCIATE_NAMERES_CONTEXT;
```

```
    input.Handle = pResult->ai_resolutionhandle;

    //
    // Associate socket with the handle
    //

    if (WSAIoctl(
            connectSocket,
            SIO_APPLY_TRANSPORT_SETTING,
            (VOID *)&input,
            sizeof(input),
            NULL,
            0,
            &bytesReturned,
            NULL,
            NULL) == SOCKET_ERROR)
    if (iResult != 0){
        printf("WSAIoctl failed: %d\n", WSAGetLastError());
        goto Exit;
    }

    //
    // Connect to server
    //

    pCur = pResult;
    while (pCur != NULL)
    {
        if (pCur->ai_addr->sa_family == AF_INET)
        {
            clientService = *(const sockaddr_in*)pCur->ai_addr;
            clientService.sin_port = htons(80);
            if (connect(
                connectSocket,
                (const SOCKADDR *)&clientService,
                sizeof(clientService)) == SOCKET_ERROR)
            {
                printf("connect failed: %d\n", WSAGetLastError());
                goto Exit;
            }
        }
        pCur = pCur->ai_next;
    }

Exit:

    if (connectSocket != INVALID_SOCKET)
    {
        closesocket(connectSocket);
    }
    if (pResult)
    {
        FreeAddrInfoExW((ADDRINFOEXW*)pResult);
    }
    WSACleanup();
    return;
}
```

Requirements

Minimum supported client	Windows 10 [desktop apps only]

Minimum supported server	Windows Server 2016 [desktop apps only]
Header	ws2def.h

See also

GetAddrInfoEx

addrinfo

addrinfoW

addrinfoex

addrinfoex3

ADDRINFOEX5 structure (ws2def.h)

10/22/2022 • 8 minutes to read • Edit Online

The **addrinfoex4** structure is used by the GetAddrInfoEx function to hold host address information when a specific network interface has been requested.

Syntax

```
typedef struct addrinfoex5 {
  int                ai_flags;
  int                ai_family;
  int                ai_socktype;
  int                ai_protocol;
  size_t             ai_addrlen;
  PWSTR              ai_canonname;
  struct sockaddr    *ai_addr;
  void               *ai_blob;
  size_t             ai_bloblen;
  GUID               *ai_provider;
  struct addrinfoex5 *ai_next;
  int                ai_version;
  PWSTR              ai_fqdn;
  int                ai_interfaceindex;
  HANDLE             ai_resolutionhandle;
  unsigned int       ai_ttl;
} ADDRINFOEX5, *PADDRINFOEX5, *LPADDRINFOEX5;
```

Members

`ai_flags`

Flags that indicate options used in the GetAddrInfoEx function.

Supported values for the **ai_flags** member are defined in the *Winsock2.h* include file and can be a combination of the following options.

VALUE	MEANING
AI_PASSIVE 0x01	The socket address will be used in a call to the bindfunction.
AI_CANONNAME 0x02	The canonical name is returned in the first **ai_canonname** member.
AI_NUMERICHOST 0x04	The *nodename* parameter passed to the GetAddrInfoEx function must be a numeric string.

AI_ALL 0x0100	If this bit is set, a request is made for IPv6 addresses and IPv4 addresses with **AI_V4MAPPED**. This option is supported on Windows Vista and later.
AI_ADDRCONFIG 0x0400	The GetAddrInfoEx will resolve only if a global address is configured. The IPv6 and IPv4 loopback address is not considered a valid global address. This option is supported on Windows Vista and later.
AI_V4MAPPED 0x0800	If the GetAddrInfoEx request for an IPv6 addresses fails, a name service request is made for IPv4 addresses and these addresses are converted to IPv4-mapped IPv6 address format. This option is supported on Windows Vista and later.
AI_NON_AUTHORITATIVE 0x04000	The address information is from non-authoritative results. When this option is set in the *pHints* parameter of GetAddrInfoEx, the **NS_EMAIL** namespace provider returns both authoritative and non-authoritative results. If this option is not set, then only authoritative results are returned. This option is only supported on Windows Vista and later for the **NS_EMAIL** namespace.
AI_SECURE 0x08000	The address information is from a secure channel. If the **AI_SECURE** bit is set, the **NS_EMAIL** namespace provider will return results that were obtained with enhanced security to minimize possible spoofing. When this option is set in the *pHints* parameter of GetAddrInfoEx, the **NS_EMAIL** namespace provider returns only results that were obtained with enhanced security to minimize possible spoofing. This option is only supported on Windows Vista and later for the **NS_EMAIL** namespace.
AI_RETURN_PREFERRED_NAMES 0x010000	The address information is for a preferred names for publication with a specific namespace. When this option is set in the *pHints* parameter of GetAddrInfoEx, no name should be provided in the *pName* parameter and the **NS_EMAIL** namespace provider will return preferred names for publication. This option is only supported on Windows Vista and later for the **NS_EMAIL** namespace.

AI_FQDN 0x00020000	The fully qualified domain name is returned in the first **ai_fqdn** member. When this option is set in the *pHints* parameter of GetAddrInfoEx and a flat name (single label) is specified in the *pName* parameter, the fully qualified domain name that the name eventually resolved to will be returned. This option is supported on Windows 7, Windows Server 2008 R2, and later.
AI_FILESERVER 0x00040000	A hint to the namespace provider that the hostname being queried is being used in a file share scenario. The namespace provider may ignore this hint. This option is supported on Windows 7, Windows Server 2008 R2, and later.
AI_DISABLE_IDN_ENCODING 0x00080000	Disable the automatic International Domain Name encoding using Punycode in the name resolution functions called by the GetAddrInfoEx function. This option is supported on Windows 8, Windows Server 2012, and later.
AI_EXTENDED 0x80000000	Indicates that the current object is extended: that is, an addrinfoex2 or greater. This option is supported on Windows 8.1, Windows Server 2012 R2, and later.
AI_RESOLUTION_HANDLE 0x40000000	A resolution handle is returned in the **ai_resolutionhandle** member. This option is supported on Windows 10, Windows Server 2016, and later.
AI_RETURN_TTL 0x0080	The number of seconds for which the DNS record is valid. If this flag is present, the **GetAddrInfoEx**.*ppResult* parameter will return a list of **addrinfoex5** structures, where the **ai_ttl** member will contain the individual TTL of the DNS record.

`ai_family`

The address family.

The 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. Note that the values for the AF_ address family and PF_ protocol family constants are identical (for example, **AF_INET** and **PF_INET**), so either constant can be used.

The table below lists common values for the address family although many other values are possible.

VALUE	MEANING

AF_UNSPEC 0	The address family is unspecified.
AF_INET 2	The Internet Protocol version 4 (IPv4) address family.
AF_NETBIOS 17	The NetBIOS address family. This address family is only supported if a Windows Sockets provider for NetBIOS is installed.
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 a Bluetooth adapter is installed.

`ai_socktype`

The socket type. Possible values for the socket type are defined in the *Winsock2.h* include file.

The following table lists the possible values for the socket type supported for Windows Sockets 2:

VALUE	MEANING
SOCK_STREAM 1	Provides sequenced, reliable, two-way, connection-based byte streams with an OOB data transmission mechanism. Uses the Transmission Control Protocol (TCP) for the Internet address family (**AF_INET** or **AF_INET6**). If the **ai_family** member is **AF_IRDA**, then **SOCK_STREAM** is the only supported socket type.
SOCK_DGRAM 2	Supports datagrams, which are connectionless, unreliable buffers of a fixed (typically small) maximum length. Uses the User Datagram Protocol (UDP) for the Internet address family (**AF_INET** or **AF_INET6**).
SOCK_RAW 3	Provides a raw socket that allows an application to manipulate the next upper-layer protocol header. To manipulate the IPv4 header, the IP_HDRINCL socket option must be set on the socket. To manipulate the IPv6 header, the IPV6_HDRINCL socket option must be set on the socket.
SOCK_RDM 4	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 reliable multicast programming.

SOCK_SEQPACKET 5	Provides a pseudo-stream packet based on datagrams.

In Windows Sockets 2, new socket types were introduced. An application can dynamically discover the attributes of each available transport protocol through the WSAEnumProtocols function. So an application can determine the possible socket type and protocol options for an address family and use this information when specifying this parameter. Socket type definitions in the *Winsock2.h* and *Ws2def.h* header files will be periodically updated as new socket types, address families, and protocols are defined.

In Windows Sockets 1.1, the only possible socket types are **SOCK_DATAGRAM** and **SOCK_STREAM**.

`ai_protocol`

The protocol type. The possible options are specific to the address family and socket type specified. Possible values for the **ai_protocol** are defined in *Winsock2.h* and the *Wsrm.h* header files.

On the Windows SDK released for Windows Vista and later,, the organization of header files has changed and this member can be one of the values from 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 for **ai_protocol**, the caller does not wish to specify a protocol and the service provider will choose the **ai_protocol** to use. For protocols other than IPv4 and IPv6, set **ai_protocol** to zero.

The following table lists common values for the **ai_protocol** member although many other values are possible.

VALUE	MEANING
IPPROTO_TCP 6	The Transmission Control Protocol (TCP). This is a possible value when the **ai_family** member is **AF_INET** or **AF_INET6** and the **ai_socktype** member is **SOCK_STREAM**.
IPPROTO_UDP 17	The User Datagram Protocol (UDP). This is a possible value when the **ai_family** member is **AF_INET** or **AF_INET6** and the *type* parameter is **SOCK_DGRAM**.
IPPROTO_RM 113	The PGM protocol for reliable multicast. This is a possible value when the **ai_family** member is **AF_INET** and the **ai_socktype** member is **SOCK_RDM**. On the Windows SDK released for Windows Vista and later, this value is also called **IPPROTO_PGM**.

If the **ai_family** member is **AF_IRDA**, then the **ai_protocol** must be 0.

`ai_addrlen`

The length, in bytes, of the buffer pointed to by the **ai_addr** member.

`ai_canonname`

The canonical name for the host.

`ai_addr`

A pointer to a sockaddr structure. The **ai_addr** member in each returned **addrinfoex4** structure points to a

filled-in socket address structure. The length, in bytes, of each returned **addrinfoex4** structure is specified in the **ai_addrlen** member.

`ai_blob`

A pointer to data that is used to return provider-specific namespace information that is associated with the name beyond a list of addresses. The length, in bytes, of the buffer pointed to by **ai_blob** must be specified in the **ai_bloblen** member.

`ai_bloblen`

The length, in bytes, of the **ai_blob** member.

`ai_provider`

A pointer to the GUID of a specific namespace provider.

`ai_next`

A pointer to the next structure in a linked list. This parameter is set to **NULL** in the last **addrinfoex4** structure of a linked list.

`ai_version`

The version number of this structure. The value currently used for this version of the structure is 4.

`ai_fqdn`

The fully qualified domain name for the host.

`ai_interfaceindex`

The interface index, as defined by the IP_ADAPTER_ADDRESSES.*IfIndex* property returned in GetAdaptersAddresses.

`ai_resolutionhandle`

Handle pointing to the fully qualified domain name for the host.

`ai_ttl`

Number of seconds for which this DNS record is valid.

Remarks

The **ADDRINFOEX5** structure is used by the GetAddrInfoExW function to hold host address information when the **AI_EXTENDED** or **AI_RETURN_TTL** bits are set in the *ai_flags* member passed in through the **GetAddrInfoExW** *hints* parameter.

The **ADDRINFOEX5** structure is an extension of the ADDRINFOEX4 structure. In addition to all fields from the ADDRINFOEX4 structure, it also holds in the *ai_ttl* member the individual DNS TTL for each returned IP address.

If the **AI_RETURN_TTL** bit is set in the *ai_flags* member passed in through the **GetAddrInfoEx** *hints* parameter, the **GetAddrInfoExW** *ppResult* parameter will return a list of **ADDRINFOEX5** structures. Each node in this list will contain in the *ai_ttl* member the individual DNS TTL for the IP address present in the **sockaddr** member.

Examples

See example code in the ADDRINFOEX4 structure topic.

Requirements

Minimum supported client	Windows 10 Build 20348
Minimum supported server	Windows 10 Build 20348
Header	ws2def.h

See also

GetAddrInfoEx

addrinfo

addrinfoW

addrinfoex

addrinfoex3

ADDRINFOEX6 structure (ws2def.h)

10/22/2022 • 9 minutes to read • Edit Online

The **addrinfoex4** structure is used by the GetAddrInfoEx function to hold host address information when a specific network interface has been requested.

Syntax

```
typedef struct addrinfoex6 {
  int                 ai_flags;
  int                 ai_family;
  int                 ai_socktype;
  int                 ai_protocol;
  size_t              ai_addrlen;
  PWSTR               ai_canonname;
  struct sockaddr     *ai_addr;
  void                *ai_blob;
  size_t              ai_bloblen;
  GUID                *ai_provider;
  struct addrinfoex5  *ai_next;
  int                 ai_version;
  PWSTR               ai_fqdn;
  int                 ai_interfaceindex;
  HANDLE              ai_resolutionhandle;
  unsigned int        ai_ttl;
  unsigned int        ai_numservers;
  ADDRINFO_DNS_SERVER *ai_servers;
  ULONG64             ai_responseflags;
} ADDRINFOEX6, *PADDRINFOEX6;
```

Members

`ai_flags`

Flags that indicate options used in the GetAddrInfoEx function.

Supported values for the **ai_flags** member are defined in the *Winsock2.h* include file and can be a combination of the following options.

VALUE	MEANING
AI_PASSIVE 0x01	The socket address will be used in a call to the bindfunction.
AI_CANONNAME 0x02	The canonical name is returned in the first **ai_canonname** member.
AI_NUMERICHOST 0x04	The *nodename* parameter passed to the GetAddrInfoEx function must be a numeric string.

AI_ALL 0x0100	If this bit is set, a request is made for IPv6 addresses and IPv4 addresses with **AI_V4MAPPED**. This option is supported on Windows Vista and later.
AI_ADDRCONFIG 0x0400	The GetAddrInfoEx will resolve only if a global address is configured. The IPv6 and IPv4 loopback address is not considered a valid global address. This option is supported on Windows Vista and later.
AI_V4MAPPED 0x0800	If the GetAddrInfoEx request for an IPv6 addresses fails, a name service request is made for IPv4 addresses and these addresses are converted to IPv4-mapped IPv6 address format. This option is supported on Windows Vista and later.
AI_NON_AUTHORITATIVE 0x04000	The address information is from non-authoritative results. When this option is set in the *pHints* parameter of GetAddrInfoEx, the **NS_EMAIL** namespace provider returns both authoritative and non-authoritative results. If this option is not set, then only authoritative results are returned. This option is only supported on Windows Vista and later for the **NS_EMAIL** namespace.
AI_SECURE 0x08000	The address information is from a secure channel. If the **AI_SECURE** bit is set, the **NS_EMAIL** namespace provider will return results that were obtained with enhanced security to minimize possible spoofing. When this option is set in the *pHints* parameter of GetAddrInfoEx, the **NS_EMAIL** namespace provider returns only results that were obtained with enhanced security to minimize possible spoofing. This option is only supported on Windows Vista and later for the **NS_EMAIL** namespace.
AI_RETURN_PREFERRED_NAMES 0x010000	The address information is for a preferred names for publication with a specific namespace. When this option is set in the *pHints* parameter of GetAddrInfoEx, no name should be provided in the *pName* parameter and the **NS_EMAIL** namespace provider will return preferred names for publication. This option is only supported on Windows Vista and later for the **NS_EMAIL** namespace.

AI_FQDN 0x00020000	The fully qualified domain name is returned in the first **ai_fqdn** member. When this option is set in the *pHints* parameter of GetAddrInfoEx and a flat name (single label) is specified in the *pName* parameter, the fully qualified domain name that the name eventually resolved to will be returned. This option is supported on Windows 7, Windows Server 2008 R2, and later.
AI_FILESERVER 0x00040000	A hint to the namespace provider that the hostname being queried is being used in a file share scenario. The namespace provider may ignore this hint. This option is supported on Windows 7, Windows Server 2008 R2, and later.
AI_DISABLE_IDN_ENCODING 0x00080000	Disable the automatic International Domain Name encoding using Punycode in the name resolution functions called by the GetAddrInfoEx function. This option is supported on Windows 8, Windows Server 2012, and later.
AI_EXTENDED 0x80000000	Indicates that the current object is extended: that is, an addrinfoex2 or greater. This option is supported on Windows 8.1, Windows Server 2012 R2, and later.
AI_RESOLUTION_HANDLE 0x40000000	A resolution handle is returned in the **ai_resolutionhandle** member. This option is supported on Windows 10, Windows Server 2016, and later.
AI_RETURN_TTL 0x0080	The number of seconds for which the DNS record is valid. If this flag is present, the **GetAddrInfoEx**.*ppResult* parameter will return a list of **addrinfoex5** structures, where the **ai_ttl** member will contain the individual TTL of the DNS record.
AI_EXCLUSIVE_CUSTOM_SERVERS 0x0080	By default, the servers specified in the *ai_servers* member are used as fallback to the system DNS servers. If this option is specified, then the DNS query will use the custom servers specified in *ai_server* exclusively.
AI_RETURN_RESPONSE_FLAGS 0x0080	Requests extra information about the DNS results. If this flag is present, the **GetAddrInfoEx** *ppResult* parameter will return a list of **addrinfoex6** structures, where the *ai_responseflags* member will contain information about the origin of the DNS results.

`ai_family`

The address family.

The 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. Note that the values for the AF_ address family and PF_ protocol family constants are identical (for example, **AF_INET** and **PF_INET**), so either constant can be used.

The table below lists common values for the address family although many other values are possible.

VALUE	MEANING
AF_UNSPEC 0	The address family is unspecified.
AF_INET 2	The Internet Protocol version 4 (IPv4) address family.
AF_NETBIOS 17	The NetBIOS address family. This address family is only supported if a Windows Sockets provider for NetBIOS is installed.
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 a Bluetooth adapter is installed.

`ai_socktype`

The socket type. Possible values for the socket type are defined in the *Winsock2.h* include file.

The following table lists the possible values for the socket type supported for Windows Sockets 2:

VALUE	MEANING
SOCK_STREAM 1	Provides sequenced, reliable, two-way, connection-based byte streams with an OOB data transmission mechanism. Uses the Transmission Control Protocol (TCP) for the Internet address family (**AF_INET** or **AF_INET6**). If the **ai_family** member is **AF_IRDA**, then **SOCK_STREAM** is the only supported socket type.
SOCK_DGRAM 2	Supports datagrams, which are connectionless, unreliable buffers of a fixed (typically small) maximum length. Uses the User Datagram Protocol (UDP) for the Internet address family (**AF_INET** or **AF_INET6**).

SOCK_RAW 3	Provides a raw socket that allows an application to manipulate the next upper-layer protocol header. To manipulate the IPv4 header, the IP_HDRINCL socket option must be set on the socket. To manipulate the IPv6 header, the IPV6_HDRINCL socket option must be set on the socket.
SOCK_RDM 4	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 reliable multicast programming.
SOCK_SEQPACKET 5	Provides a pseudo-stream packet based on datagrams.

In Windows Sockets 2, new socket types were introduced. An application can dynamically discover the attributes of each available transport protocol through the WSAEnumProtocols function. So an application can determine the possible socket type and protocol options for an address family and use this information when specifying this parameter. Socket type definitions in the *Winsock2.h* and *Ws2def.h* header files will be periodically updated as new socket types, address families, and protocols are defined.

In Windows Sockets 1.1, the only possible socket types are **SOCK_DATAGRAM** and **SOCK_STREAM**.

`ai_protocol`

The protocol type. The possible options are specific to the address family and socket type specified. Possible values for the **ai_protocol** are defined in *Winsock2.h* and the *Wsrm.h* header files.

On the Windows SDK released for Windows Vista and later,, the organization of header files has changed and this member can be one of the values from 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 for **ai_protocol**, the caller does not wish to specify a protocol and the service provider will choose the **ai_protocol** to use. For protocols other than IPv4 and IPv6, set **ai_protocol** to zero.

The following table lists common values for the **ai_protocol** member although many other values are possible.

VALUE	MEANING
IPPROTO_TCP 6	The Transmission Control Protocol (TCP). This is a possible value when the **ai_family** member is **AF_INET** or **AF_INET6** and the **ai_socktype** member is **SOCK_STREAM**.
IPPROTO_UDP 17	The User Datagram Protocol (UDP). This is a possible value when the **ai_family** member is **AF_INET** or **AF_INET6** and the *type* parameter is **SOCK_DGRAM**.
IPPROTO_RM 113	The PGM protocol for reliable multicast. This is a possible value when the **ai_family** member is **AF_INET** and the **ai_socktype** member is **SOCK_RDM**. On the Windows SDK released for Windows Vista and later, this value is also called **IPPROTO_PGM**.

If the **ai_family** member is **AF_IRDA**, then the **ai_protocol** must be 0.

`ai_addrlen`

The length, in bytes, of the buffer pointed to by the **ai_addr** member.

`ai_canonname`

The canonical name for the host.

`ai_addr`

A pointer to a sockaddr structure. The **ai_addr** member in each returned **addrinfoex4** structure points to a filled-in socket address structure. The length, in bytes, of each returned **addrinfoex4** structure is specified in the **ai_addrlen** member.

`ai_blob`

A pointer to data that is used to return provider-specific namespace information that is associated with the name beyond a list of addresses. The length, in bytes, of the buffer pointed to by **ai_blob** must be specified in the **ai_bloblen** member.

`ai_bloblen`

The length, in bytes, of the **ai_blob** member.

`ai_provider`

A pointer to the GUID of a specific namespace provider.

`ai_next`

A pointer to the next structure in a linked list. This parameter is set to **NULL** in the last **addrinfoex4** structure of a linked list.

`ai_version`

The version number of this structure. The value currently used for this version of the structure is 4.

`ai_fqdn`

The fully qualified domain name for the host.

`ai_interfaceindex`

The interface index, as defined by the IP_ADAPTER_ADDRESSES.*IfIndex* property returned in GetAdaptersAddresses.

`ai_resolutionhandle`

Handle pointing to the fully qualified domain name for the host.

`ai_ttl`

Number of seconds for which this DNS record is valid.

`ai_numservers`

Number of custom DNS servers present in the *ai_servers* member.

`ai_servers`

An array of N (where N == *ai_numservers*) ADDRINFO_DNS_SERVER objects. If *ai_numservers* is 0, then *ai_servers* must be **NULL**.

`ai_responseflags`

Returns a bitmap containing any of the following values.

CONSTANT	VALUE	MEANING
AI_DNS_RESPONSE_SECURE	0x1	The response came from secure protocols, such as *DNS-over-HTTPS*.
AI_DNS_RESPONSE_HOSTFILE	0x2	The response came from the *hosts* file.

Remarks

The **ADDRINFOEX6** structure is an extension of the ADDRINFOEX5 structure. In addition to all fields from the ADDRINFOEX5 structure, it holds the *ai_numservers* and *ai_servers* members, enabling the use of custom per-query DNS servers. It also holds the *ai_responseflags* member, which contains information about the origin of the DNS responses.

If the **AI_RETURN_RESPONSE_FLAGS** flag is present in the *ai_flags* member, then the **GetAddrInfoEx** *ppResult* parameter will return a list of **ADDRINFOEX6** structures, where the *ai_responseflags* member will contain information about the origin of the DNS results.

If the *ai_numservers* and *ai_servers* members point to an array of valid ADDRINFO_DNS_SERVER objects, then the DNS query will add these servers as fallback to the system-configured DNS servers. If the **AI_EXCLUSIVE_CUSTOM_SERVERS** option is present in the *ai_flags* member, then the DNS query will use the custom servers exclusively.

Examples

See example code in the ADDRINFOEX4 structure topic.

Requirements

Minimum supported client	Windows Build 22000
Minimum supported server	Windows Build 22000
Header	ws2def.h

See also

GetAddrInfoEx

addrinfo

addrinfoW

addrinfoex

addrinfoex3

ADDRINFOEXA structure (ws2def.h)

10/22/2022 • 12 minutes to read • Edit Online

The **addrinfoex** structure is used by the GetAddrInfoEx function to hold host address information.

Syntax

```
typedef struct addrinfoexA {
  int                ai_flags;
  int                ai_family;
  int                ai_socktype;
  int                ai_protocol;
  size_t             ai_addrlen;
  char               *ai_canonname;
  struct sockaddr    *ai_addr;
  void               *ai_blob;
  size_t             ai_bloblen;
  LPGUID             ai_provider;
  struct addrinfoexA *ai_next;
} ADDRINFOEXA, *PADDRINFOEXA, *LPADDRINFOEXA;
```

Members

`ai_flags`

Type: **int**

Flags that indicate options used in the GetAddrInfoEx function.

Supported values for the **ai_flags** member are defined in the *Winsock2.h* include file and can be a combination of the following options.

VALUE	MEANING
AI_PASSIVE 0x01	The socket address will be used in a call to the bind function.
AI_CANONNAME 0x02	The canonical name is returned in the first **ai_canonname** member. When both the **AI_CANONNAME** and **AI_FQDN** bits are set, an addrinfoex2 structure is returned not the **addrinfoex** structure.
AI_NUMERICHOST 0x04	The *nodename* parameter passed to the GetAddrInfoEx function must be a numeric string.
AI_ALL 0x0100	If this bit is set, a request is made for IPv6 addresses and IPv4 addresses with **AI_V4MAPPED**. This option is supported on Windows Vista and later.

AI_ADDRCONFIG 0x0400	The GetAddrInfoEx will resolve only if a global address is configured. The IPv6 and IPv4 loopback address is not considered a valid global address. This option is only supported on Windows Vista and later.
AI_V4MAPPED 0x0800	If the GetAddrInfoEx request for an IPv6 addresses fails, a name service request is made for IPv4 addresses and these addresses are converted to IPv4-mapped IPv6 address format. This option is supported on Windows Vista and later.
AI_NON_AUTHORITATIVE 0x04000	The address information is from non-authoritative results. When this option is set in the *pHints* parameter of GetAddrInfoEx, the **NS_EMAIL** namespace provider returns both authoritative and non-authoritative results. If this option is not set, then only authoritative results are returned. In the *ppResults* parameter returned by GetAddrInfoEx, this flag is set in the **ai_flags** member of the **addrinfoex** structure for non-authoritative results. This option is only supported on Windows Vista and later for the **NS_EMAIL** namespace.
AI_SECURE 0x08000	The address information is from a secure channel. If the **AI_SECURE** bit is set, the **NS_EMAIL** namespace provider will return results that were obtained with enhanced security to minimize possible spoofing. When this option is set in the *pHints* parameter of GetAddrInfoEx, the **NS_EMAIL** namespace provider returns only results that were obtained with enhanced security to minimize possible spoofing. In the *ppResults* parameter returned by GetAddrInfoEx, this flag is set in the **ai_flags** member of the **addrinfoex** structure for results returned with enhanced security to minimize possible spoofing. This option is only supported on Windows Vista and later for the **NS_EMAIL** namespace.
AI_RETURN_PREFERRED_NAMES 0x010000	The address information is for a preferred names for publication with a specific namespace. When this option is set in the *pHints* parameter of GetAddrInfoEx, no name should be provided in the *pName* parameter and the **NS_EMAIL** namespace provider will return preferred names for publication. In the *ppResults* parameter returned by GetAddrInfoEx, this flag is set in the **ai_flags** member of the **addrinfoex** structure for results returned for preferred names for publication. This option is only supported on Windows Vista and later for the **NS_EMAIL** namespace.

AI_FQDN 0x00020000	The fully qualified domain name is returned in the first **ai_canonicalname** member. When this option is set in the *pHints* parameter of GetAddrInfoEx and a flat name (single label) is specified in the *pName* parameter, the fully qualified domain name that the name eventually resolved to will be returned. When both the **AI_CANONNAME** and **AI_FQDN** bits are set, an addrinfoex2 structure is returned not the **addrinfoex** structure. This option is supported on Windows 7, Windows Server 2008 R2, and later.
AI_FILESERVER 0x00040000	A hint to the namespace provider that the hostname being queried is being used in a file share scenario. The namespace provider may ignore this hint. This option is supported on Windows 7, Windows Server 2008 R2, and later.
AI_DISABLE_IDN_ENCODING 0x00080000	Disable the automatic International Domain Name encoding using Punycode in the name resolution functions called by the GetAddrInfoEx function. This option is supported on Windows 8, Windows Server 2012, and later.

`ai_family`

Type: **int**

The address family. Possible values for the address family are defined in the *Winsock2.h* include file.

On the Windows SDK released for Windows Vista and later,, the organization of header files has changed and the 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. Note that the values for the AF_ address family and PF_ protocol family constants are identical (for example, **AF_UNSPEC** and **PF_UNSPEC**), so either constant can be used.

The table below lists common values for the address family although many other values are possible.

VALUE	MEANING
AF_UNSPEC 0	The address family is unspecified.
AF_INET 2	The Internet Protocol version 4 (IPv4) address family.

AF_NETBIOS 17	The NetBIOS address family. This address family is only supported if a Windows Sockets provider for NetBIOS is installed.
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 a Bluetooth adapter is installed on Windows Server 2003 or later.

`ai_socktype`

Type: **int**

The socket type. Possible values for the socket type are defined in the *Winsock2.h* include file.

The following table lists the possible values for the socket type supported for Windows Sockets 2:

VALUE	MEANING
SOCK_STREAM 1	Provides sequenced, reliable, two-way, connection-based byte streams with an OOB data transmission mechanism. Uses the Transmission Control Protocol (TCP) for the Internet address family (**AF_INET** or **AF_INET6**). If the **ai_family** member is **AF_IRDA**, then **SOCK_STREAM** is the only supported socket type.
SOCK_DGRAM 2	Supports datagrams, which are connectionless, unreliable buffers of a fixed (typically small) maximum length. Uses the User Datagram Protocol (UDP) for the Internet address family (**AF_INET** or **AF_INET6**).
SOCK_RAW 3	Provides a raw socket that allows an application to manipulate the next upper-layer protocol header. To manipulate the IPv4 header, the IP_HDRINCL socket option must be set on the socket. To manipulate the IPv6 header, the IPV6_HDRINCL socket option must be set on the socket.
SOCK_RDM 4	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 reliable multicast programming.
SOCK_SEQPACKET 5	Provides a pseudo-stream packet based on datagrams.

In Windows Sockets 2, new socket types were introduced. An application can dynamically discover the attributes of each available transport protocol through the WSAEnumProtocols function. So an application can determine the possible socket type and protocol options for an address family and use this information when specifying

this parameter. Socket type definitions in the *Winsock2.h* and *Ws2def.h* header files will be periodically updated as new socket types, address families, and protocols are defined.

In Windows Sockets 1.1, the only possible socket types are **SOCK_DATAGRAM** and **SOCK_STREAM**.

`ai_protocol`

Type: **int**

The protocol type. The possible options are specific to the address family and socket type specified. Possible values for the **ai_protocol** are defined in *Winsock2.h* and the *Wsrm.h* header files.

On the Windows SDK released for Windows Vista and later,, the organization of header files has changed and this member can be one of the values from 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 for **ai_protocol**, the caller does not wish to specify a protocol and the service provider will choose the **ai_protocol** to use. For protocols other than IPv4 and IPv6, set **ai_protocol** to zero.

The following table lists common values for the **ai_protocol** member although many other values are possible.

VALUE	MEANING
IPPROTO_TCP 6	The Transmission Control Protocol (TCP). This is a possible value when the **ai_family** member is **AF_INET** or **AF_INET6** and the **ai_socktype** member is **SOCK_STREAM**.
IPPROTO_UDP 17	The User Datagram Protocol (UDP). This is a possible value when the **ai_family** member is **AF_INET** or **AF_INET6** and the *type* parameter is **SOCK_DGRAM**.
IPPROTO_RM 113	The PGM protocol for reliable multicast. This is a possible value when the **ai_family** member is **AF_INET** and the **ai_socktype** member is **SOCK_RDM**. On the Windows SDK released for Windows Vista and later, this value is also called **IPPROTO_PGM**.

If the **ai_family** member is **AF_IRDA**, then the **ai_protocol** must be 0.

`ai_addrlen`

Type: **size_t**

The length, in bytes, of the buffer pointed to by the **ai_addr** member.

`ai_canonname`

Type: **PCTSTR**

The canonical name for the host.

`ai_addr`

Type: **struct sockaddr***

A pointer to a sockaddr structure. The **ai_addr** member in each returned **addrinfoex** structure points to a filled-in socket address structure. The length, in bytes, of each returned **addrinfoex** structure is specified in the **ai_addrlen** member.

`ai_blob`

Type: **void***

A pointer to data that is used to return provider-specific namespace information that is associated with the name beyond a list of addresses. The length, in bytes, of the buffer pointed to by **ai_blob** must be specified in the **ai_bloblen** member.

`ai_bloblen`

Type: **size_t**

The length, in bytes, of the **ai_blob** member.

`ai_provider`

Type: **LPGUID**

A pointer to the GUID of a specific namespace provider.

`ai_next`

Type: **struct addrinfoex***

A pointer to the next structure in a linked list. This parameter is set to **NULL** in the last **addrinfoex** structure of a linked list.

Remarks

The **addrinfoex** structure is used by the GetAddrInfoEx function to hold host address information. The **addrinfoex** structure is an enhanced version of the addrinfo and addrinfoW structures. The extra structure members are for blob data and the GUID for the namespace provider. The blob data is used to return additional provider-specific namespace information associated with a name. The format of data in the **ai_blob** member is specific to a particular namespace provider. Currently, blob data is used by the **NS_EMAIL** namespace provider to supply additional information.

The **addrinfoex** structure is an enhanced version of the addrinfo and addrinfoW structure used with GetAddrInfoEx function. The **GetAddrInfoEx** function allows specifying the namespace provider to resolve the query. For use with the IPv6 and IPv4 protocol, name resolution can be by the Domain Name System (DNS), a local *hosts* file, an email provider (the **NS_EMAIL** namespace), or by other naming mechanisms.

When UNICODE or _UNICODE is defined, **addrinfoex** is defined to **addrinfoexW**, the Unicode version of this structure. The string parameters are defined to the **PWSTR** data type and the **addrinfoexW** structure is used.

When UNICODE or _UNICODE is not defined, **addrinfoex** is defined to **addrinfoexA**, the ANSI version of this structure. The string parameters are of the **PCSTR** data type and the **addrinfoexA** structure is used.

Upon a successful call to GetAddrInfoEx, a linked list of **addrinfoex** structures is returned in the *ppResult* parameter passed to the **GetAddrInfoEx** function. The list can be processed by following the pointer provided in the **ai_next** member of each returned **addrinfoex** structure until a **NULL** pointer is encountered. In each returned **addrinfoex** structure, the **ai_family**, **ai_socktype**, and **ai_protocol** members correspond to respective arguments in a socket or WSASocket function call. Also, the **ai_addr** member in each returned **addrinfoex** structure points to a filled-in socket address structure, the length of which is specified in its **ai_addrlen** member.

Examples

The following example demonstrates the use of the **addrinfoex** structure.

```
#ifndef UNICODE
```

```
#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>

#pragma comment(lib, "Ws2_32.lib")

int __cdecl wmain(int argc, wchar_t ** argv)
{
//--------------------------------
// Declare and initialize variables.
    WSADATA wsaData;
    int iResult;

    ADDRINFOEX *result = NULL;
    ADDRINFOEX *ptr = NULL;
    ADDRINFOEX hints;

    DWORD dwRetval = 0;
    int i = 1;

    DWORD dwNamespace = NS_DNS;
    LPGUID lpNspid = NULL;

    struct sockaddr_in *sockaddr_ipv4;
    struct sockaddr_in6 *sockaddr_ipv6;
//    LPSOCKADDR sockaddr_ip;

    wchar_t ipstringbuffer[46];

    // Validate the parameters
    if (argc != 3) {
        wprintf(L"usage: %ws <hostname> <servicename>\n", argv[0]);
        wprintf(L"       provides protocol-independent translation\n");
        wprintf(L"       from a host name to an IP address\n");
        wprintf(L"%ws example usage\n", argv[0]);
        wprintf(L"   %ws www.contoso.com 0\n", argv[0]);
        return 1;
    }
    // Initialize Winsock
    iResult = WSAStartup(MAKEWORD(2, 2), &wsaData);
    if (iResult != 0) {
        wprintf(L"WSAStartup failed: %d\n", iResult);
        return 1;
    }
//--------------------------------
// Setup the hints address info structure
// which is passed to the GetAddrInfoW() function
    memset(&hints, 0, sizeof (hints));
    hints.ai_family = AF_UNSPEC;
    hints.ai_socktype = SOCK_STREAM;
    hints.ai_protocol = IPPROTO_TCP;

    wprintf(L"Calling GetAddrInfoEx with following parameters:\n");
    wprintf(L"\tName = %ws\n", argv[1]);
    wprintf(L"\tServiceName (or port) = %ws\n\n", argv[2]);

//--------------------------------
// Call GetAddrInfoEx(). If the call succeeds,
// the aiList variable will hold a linked list
// of ADDRINFOEX structures containing response
// information about the host
```

```
// information about the host
    dwRetval = GetAddrInfoEx(argv[1], argv[2],
                             dwNamespace, lpNspid, &hints, &result,
                             NULL, NULL, NULL, NULL);

    if (dwRetval != 0) {
        wprintf(L"GetAddrInfoEx failed with error: %d\n", dwRetval);
        WSACleanup();
        return 1;
    }
    wprintf(L"GetAddrInfoEx returned success\n");

    // Retrieve each address and print out the hex bytes
    for (ptr = result; ptr != NULL; ptr = ptr->ai_next) {

        wprintf(L"GetAddrInfoEx response %d\n", i++);
        wprintf(L"\tFlags: 0x%x\n", ptr->ai_flags);
        wprintf(L"\tFamily: ");
        switch (ptr->ai_family) {
        case AF_UNSPEC:
            wprintf(L"Unspecified\n");
            break;
        case AF_INET:
            wprintf(L"AF_INET (IPv4)\n");
            // the InetNtop function is available on Windows Vista and later
            sockaddr_ipv4 = (struct sockaddr_in *) ptr->ai_addr;
            wprintf(L"\tIPv4 address %ws\n",
                    InetNtop(AF_INET, &sockaddr_ipv4->sin_addr, ipstringbuffer,
                             46));

            // We could also use the WSAAddressToString function
            // sockaddr_ip = (LPSOCKADDR) ptr->ai_addr;
            // The buffer length is changed by each call to WSAAddresstoString
            // So we need to set it for each iteration through the loop for safety
            // ipbufferlength = 46;
            // iRetval = WSAAddressToString(sockaddr_ip, (DWORD) ptr->ai_addrlen, NULL,
            //    ipstringbuffer, &ipbufferlength );
            // if (iRetval)
            //    wprintf(L"WSAAddressToString failed with %u\n", WSAGetLastError() );
            // else
            //    wprintf(L"\tIPv4 address %ws\n", ipstringbuffer);
            break;
        case AF_INET6:
            wprintf(L"AF_INET6 (IPv6)\n");
            // the InetNtop function is available on Windows Vista and later
            sockaddr_ipv6 = (struct sockaddr_in6 *) ptr->ai_addr;
            wprintf(L"\tIPv6 address %ws\n",
                    InetNtop(AF_INET6, &sockaddr_ipv6->sin6_addr,
                             ipstringbuffer, 46));

            // We could also use WSAAddressToString which also returns the scope ID
            // sockaddr_ip = (LPSOCKADDR) ptr->ai_addr;
            // The buffer length is changed by each call to WSAAddresstoString
            // So we need to set it for each iteration through the loop for safety
            // ipbufferlength = 46;
            //iRetval = WSAAddressToString(sockaddr_ip, (DWORD) ptr->ai_addrlen, NULL,
            //    ipstringbuffer, &ipbufferlength );
            //if (iRetval)
            //    wprintf(L"WSAAddressToString failed with %u\n", WSAGetLastError() );
            //else
            //    wprintf(L"\tIPv6 address %ws\n", ipstringbuffer);
            break;
        default:
            wprintf(L"Other %ld\n", ptr->ai_family);
            break;
        }
        wprintf(L"\tSocket type: ");
        switch (ptr->ai_socktype) {
        case 0:
```

```
            wprintf(L"Unspecified\n");
            break;
        case SOCK_STREAM:
            wprintf(L"SOCK_STREAM (stream)\n");
            break;
        case SOCK_DGRAM:
            wprintf(L"SOCK_DGRAM (datagram) \n");
            break;
        case SOCK_RAW:
            wprintf(L"SOCK_RAW (raw) \n");
            break;
        case SOCK_RDM:
            wprintf(L"SOCK_RDM (reliable message datagram)\n");
            break;
        case SOCK_SEQPACKET:
            wprintf(L"SOCK_SEQPACKET (pseudo-stream packet)\n");
            break;
        default:
            wprintf(L"Other %ld\n", ptr->ai_socktype);
            break;
        }
        wprintf(L"\tProtocol: ");
        switch (ptr->ai_protocol) {
        case 0:
            wprintf(L"Unspecified\n");
            break;
        case IPPROTO_TCP:
            wprintf(L"IPPROTO_TCP (TCP)\n");
            break;
        case IPPROTO_UDP:
            wprintf(L"IPPROTO_UDP (UDP) \n");
            break;
        default:
            wprintf(L"Other %ld\n", ptr->ai_protocol);
            break;
        }
        wprintf(L"\tLength of this sockaddr: %d\n", ptr->ai_addrlen);
        wprintf(L"\tCanonical name: %s\n", ptr->ai_canonname);
    }

    FreeAddrInfoEx(result);
    WSACleanup();

    return 0;
}
```

Note Ensure that the development environment targets the newest version of *Ws2tcpip.h* which includes structure and function definitions for **ADDRINFOEX** and GetAddrInfoEx, respectively.

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]

Header	ws2def.h (include Windows Server 2012, Windows 7 Windows Server 2008 R2)

See also

GetAddrInfoEx

addrinfo

addrinfoW

ADDRINFOEXW structure (ws2def.h)

10/22/2022 • 12 minutes to read • Edit Online

The **addrinfoex** structure is used by the GetAddrInfoEx function to hold host address information.

Syntax

```
typedef struct addrinfoexW {
  int                 ai_flags;
  int                 ai_family;
  int                 ai_socktype;
  int                 ai_protocol;
  size_t              ai_addrlen;
  PWSTR               ai_canonname;
  struct sockaddr     *ai_addr;
  void                *ai_blob;
  size_t              ai_bloblen;
  LPGUID              ai_provider;
  struct addrinfoexW *ai_next;
} ADDRINFOEXW, *PADDRINFOEXW, *LPADDRINFOEXW;
```

Members

`ai_flags`

Type: **int**

Flags that indicate options used in the GetAddrInfoEx function.

Supported values for the **ai_flags** member are defined in the *Winsock2.h* include file and can be a combination of the following options.

VALUE	MEANING
AI_PASSIVE 0x01	The socket address will be used in a call to the bind function.
AI_CANONNAME 0x02	The canonical name is returned in the first **ai_canonname** member. When both the **AI_CANONNAME** and **AI_FQDN** bits are set, an addrinfoex2 structure is returned not the **addrinfoex** structure.
AI_NUMERICHOST 0x04	The *nodename* parameter passed to the GetAddrInfoEx function must be a numeric string.
AI_ALL 0x0100	If this bit is set, a request is made for IPv6 addresses and IPv4 addresses with **AI_V4MAPPED**. This option is supported on Windows Vista and later.

AI_ADDRCONFIG 0x0400	The GetAddrInfoEx will resolve only if a global address is configured. The IPv6 and IPv4 loopback address is not considered a valid global address. This option is only supported on Windows Vista and later.
AI_V4MAPPED 0x0800	If the GetAddrInfoEx request for an IPv6 addresses fails, a name service request is made for IPv4 addresses and these addresses are converted to IPv4-mapped IPv6 address format. This option is supported on Windows Vista and later.
AI_NON_AUTHORITATIVE 0x04000	The address information is from non-authoritative results. When this option is set in the *pHints* parameter of GetAddrInfoEx, the **NS_EMAIL** namespace provider returns both authoritative and non-authoritative results. If this option is not set, then only authoritative results are returned. In the *ppResults* parameter returned by GetAddrInfoEx, this flag is set in the **ai_flags** member of the **addrinfoex** structure for non-authoritative results. This option is only supported on Windows Vista and later for the **NS_EMAIL** namespace.
AI_SECURE 0x08000	The address information is from a secure channel. If the **AI_SECURE** bit is set, the **NS_EMAIL** namespace provider will return results that were obtained with enhanced security to minimize possible spoofing. When this option is set in the *pHints* parameter of GetAddrInfoEx, the **NS_EMAIL** namespace provider returns only results that were obtained with enhanced security to minimize possible spoofing. In the *ppResults* parameter returned by GetAddrInfoEx, this flag is set in the **ai_flags** member of the **addrinfoex** structure for results returned with enhanced security to minimize possible spoofing. This option is only supported on Windows Vista and later for the **NS_EMAIL** namespace.
AI_RETURN_PREFERRED_NAMES 0x010000	The address information is for a preferred names for publication with a specific namespace. When this option is set in the *pHints* parameter of GetAddrInfoEx, no name should be provided in the *pName* parameter and the **NS_EMAIL** namespace provider will return preferred names for publication. In the *ppResults* parameter returned by GetAddrInfoEx, this flag is set in the **ai_flags** member of the **addrinfoex** structure for results returned for preferred names for publication. This option is only supported on Windows Vista and later for the **NS_EMAIL** namespace.

AI_FQDN 0x00020000	The fully qualified domain name is returned in the first **ai_canonicalname** member. When this option is set in the *pHints* parameter of GetAddrInfoEx and a flat name (single label) is specified in the *pName* parameter, the fully qualified domain name that the name eventually resolved to will be returned. When both the **AI_CANONNAME** and **AI_FQDN** bits are set, an addrinfoex2 structure is returned not the **addrinfoex** structure. This option is supported on Windows 7, Windows Server 2008 R2, and later.
AI_FILESERVER 0x00040000	A hint to the namespace provider that the hostname being queried is being used in a file share scenario. The namespace provider may ignore this hint. This option is supported on Windows 7, Windows Server 2008 R2, and later.
AI_DISABLE_IDN_ENCODING 0x00080000	Disable the automatic International Domain Name encoding using Punycode in the name resolution functions called by the GetAddrInfoEx function. This option is supported on Windows 8, Windows Server 2012, and later.

`ai_family`

Type: **int**

The address family. Possible values for the address family are defined in the *Winsock2.h* include file.

On the Windows SDK released for Windows Vista and later,, the organization of header files has changed and the 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. Note that the values for the AF_ address family and PF_ protocol family constants are identical (for example, **AF_UNSPEC** and **PF_UNSPEC**), so either constant can be used.

The table below lists common values for the address family although many other values are possible.

VALUE	MEANING
AF_UNSPEC 0	The address family is unspecified.
AF_INET 2	The Internet Protocol version 4 (IPv4) address family.

AF_NETBIOS 17	The NetBIOS address family. This address family is only supported if a Windows Sockets provider for NetBIOS is installed.
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 a Bluetooth adapter is installed on Windows Server 2003 or later.

`ai_socktype`

Type: **int**

The socket type. Possible values for the socket type are defined in the *Winsock2.h* include file.

The following table lists the possible values for the socket type supported for Windows Sockets 2:

VALUE	MEANING
SOCK_STREAM 1	Provides sequenced, reliable, two-way, connection-based byte streams with an OOB data transmission mechanism. Uses the Transmission Control Protocol (TCP) for the Internet address family (**AF_INET** or **AF_INET6**). If the **ai_family** member is **AF_IRDA**, then **SOCK_STREAM** is the only supported socket type.
SOCK_DGRAM 2	Supports datagrams, which are connectionless, unreliable buffers of a fixed (typically small) maximum length. Uses the User Datagram Protocol (UDP) for the Internet address family (**AF_INET** or **AF_INET6**).
SOCK_RAW 3	Provides a raw socket that allows an application to manipulate the next upper-layer protocol header. To manipulate the IPv4 header, the IP_HDRINCL socket option must be set on the socket. To manipulate the IPv6 header, the IPV6_HDRINCL socket option must be set on the socket.
SOCK_RDM 4	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 reliable multicast programming.
SOCK_SEQPACKET 5	Provides a pseudo-stream packet based on datagrams.

In Windows Sockets 2, new socket types were introduced. An application can dynamically discover the attributes of each available transport protocol through the WSAEnumProtocols function. So an application can determine the possible socket type and protocol options for an address family and use this information when specifying

this parameter. Socket type definitions in the *Winsock2.h* and *Ws2def.h* header files will be periodically updated as new socket types, address families, and protocols are defined.

In Windows Sockets 1.1, the only possible socket types are **SOCK_DATAGRAM** and **SOCK_STREAM**.

`ai_protocol`

Type: **int**

The protocol type. The possible options are specific to the address family and socket type specified. Possible values for the **ai_protocol** are defined in *Winsock2.h* and the *Wsrm.h* header files.

On the Windows SDK released for Windows Vista and later,, the organization of header files has changed and this member can be one of the values from 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 for **ai_protocol**, the caller does not wish to specify a protocol and the service provider will choose the **ai_protocol** to use. For protocols other than IPv4 and IPv6, set **ai_protocol** to zero.

The following table lists common values for the **ai_protocol** member although many other values are possible.

VALUE	MEANING
IPPROTO_TCP 6	The Transmission Control Protocol (TCP). This is a possible value when the **ai_family** member is **AF_INET** or **AF_INET6** and the **ai_socktype** member is **SOCK_STREAM**.
IPPROTO_UDP 17	The User Datagram Protocol (UDP). This is a possible value when the **ai_family** member is **AF_INET** or **AF_INET6** and the *type* parameter is **SOCK_DGRAM**.
IPPROTO_RM 113	The PGM protocol for reliable multicast. This is a possible value when the **ai_family** member is **AF_INET** and the **ai_socktype** member is **SOCK_RDM**. On the Windows SDK released for Windows Vista and later, this value is also called **IPPROTO_PGM**.

If the **ai_family** member is **AF_IRDA**, then the **ai_protocol** must be 0.

`ai_addrlen`

Type: **size_t**

The length, in bytes, of the buffer pointed to by the **ai_addr** member.

`ai_canonname`

Type: **PCTSTR**

The canonical name for the host.

`ai_addr`

Type: **struct sockaddr***

A pointer to a sockaddr structure. The **ai_addr** member in each returned **addrinfoex** structure points to a filled-in socket address structure. The length, in bytes, of each returned **addrinfoex** structure is specified in the **ai_addrlen** member.

`ai_blob`

Type: **void***

A pointer to data that is used to return provider-specific namespace information that is associated with the name beyond a list of addresses. The length, in bytes, of the buffer pointed to by **ai_blob** must be specified in the **ai_bloblen** member.

`ai_bloblen`

Type: **size_t**

The length, in bytes, of the **ai_blob** member.

`ai_provider`

Type: **LPGUID**

A pointer to the GUID of a specific namespace provider.

`ai_next`

Type: **struct addrinfoex***

A pointer to the next structure in a linked list. This parameter is set to **NULL** in the last **addrinfoex** structure of a linked list.

Remarks

The **addrinfoex** structure is used by the GetAddrInfoEx function to hold host address information. The **addrinfoex** structure is an enhanced version of the addrinfo and addrinfoW structures. The extra structure members are for blob data and the GUID for the namespace provider. The blob data is used to return additional provider-specific namespace information associated with a name. The format of data in the **ai_blob** member is specific to a particular namespace provider. Currently, blob data is used by the **NS_EMAIL** namespace provider to supply additional information.

The **addrinfoex** structure is an enhanced version of the addrinfo and addrinfoW structure used with GetAddrInfoEx function. The **GetAddrInfoEx** function allows specifying the namespace provider to resolve the query. For use with the IPv6 and IPv4 protocol, name resolution can be by the Domain Name System (DNS), a local *hosts* file, an email provider (the **NS_EMAIL** namespace), or by other naming mechanisms.

When UNICODE or _UNICODE is defined, **addrinfoex** is defined to **addrinfoexW**, the Unicode version of this structure. The string parameters are defined to the **PWSTR** data type and the **addrinfoexW** structure is used.

When UNICODE or _UNICODE is not defined, **addrinfoex** is defined to **addrinfoexA**, the ANSI version of this structure. The string parameters are of the **PCSTR** data type and the **addrinfoexA** structure is used.

Upon a successful call to GetAddrInfoEx, a linked list of **addrinfoex** structures is returned in the *ppResult* parameter passed to the **GetAddrInfoEx** function. The list can be processed by following the pointer provided in the **ai_next** member of each returned **addrinfoex** structure until a **NULL** pointer is encountered. In each returned **addrinfoex** structure, the **ai_family**, **ai_socktype**, and **ai_protocol** members correspond to respective arguments in a socket or WSASocket function call. Also, the **ai_addr** member in each returned **addrinfoex** structure points to a filled-in socket address structure, the length of which is specified in its **ai_addrlen** member.

Examples

The following example demonstrates the use of the **addrinfoex** structure.

```
#ifndef UNICODE
```

```
#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>

#pragma comment(lib, "Ws2_32.lib")

int __cdecl wmain(int argc, wchar_t ** argv)
{
//--------------------------------
// Declare and initialize variables.
    WSADATA wsaData;
    int iResult;

    ADDRINFOEX *result = NULL;
    ADDRINFOEX *ptr = NULL;
    ADDRINFOEX hints;

    DWORD dwRetval = 0;
    int i = 1;

    DWORD dwNamespace = NS_DNS;
    LPGUID lpNspid = NULL;

    struct sockaddr_in *sockaddr_ipv4;
    struct sockaddr_in6 *sockaddr_ipv6;
//    LPSOCKADDR sockaddr_ip;

    wchar_t ipstringbuffer[46];

    // Validate the parameters
    if (argc != 3) {
        wprintf(L"usage: %ws <hostname> <servicename>\n", argv[0]);
        wprintf(L"       provides protocol-independent translation\n");
        wprintf(L"       from a host name to an IP address\n");
        wprintf(L"%ws example usage\n", argv[0]);
        wprintf(L"   %ws www.contoso.com 0\n", argv[0]);
        return 1;
    }
    // Initialize Winsock
    iResult = WSAStartup(MAKEWORD(2, 2), &wsaData);
    if (iResult != 0) {
        wprintf(L"WSAStartup failed: %d\n", iResult);
        return 1;
    }
//--------------------------------
// Setup the hints address info structure
// which is passed to the GetAddrInfoW() function
    memset(&hints, 0, sizeof (hints));
    hints.ai_family = AF_UNSPEC;
    hints.ai_socktype = SOCK_STREAM;
    hints.ai_protocol = IPPROTO_TCP;

    wprintf(L"Calling GetAddrInfoEx with following parameters:\n");
    wprintf(L"\tName = %ws\n", argv[1]);
    wprintf(L"\tServiceName (or port) = %ws\n\n", argv[2]);

//--------------------------------
// Call GetAddrInfoEx(). If the call succeeds,
// the aiList variable will hold a linked list
// of ADDRINFOEX structures containing response
// information about the host
```

```
// information about the host
    dwRetval = GetAddrInfoEx(argv[1], argv[2],
                             dwNamespace, lpNspid, &hints, &result,
                             NULL, NULL, NULL, NULL);

    if (dwRetval != 0) {
        wprintf(L"GetAddrInfoEx failed with error: %d\n", dwRetval);
        WSACleanup();
        return 1;
    }
    wprintf(L"GetAddrInfoEx returned success\n");

    // Retrieve each address and print out the hex bytes
    for (ptr = result; ptr != NULL; ptr = ptr->ai_next) {

        wprintf(L"GetAddrInfoEx response %d\n", i++);
        wprintf(L"\tFlags: 0x%x\n", ptr->ai_flags);
        wprintf(L"\tFamily: ");
        switch (ptr->ai_family) {
        case AF_UNSPEC:
            wprintf(L"Unspecified\n");
            break;
        case AF_INET:
            wprintf(L"AF_INET (IPv4)\n");
            // the InetNtop function is available on Windows Vista and later
            sockaddr_ipv4 = (struct sockaddr_in *) ptr->ai_addr;
            wprintf(L"\tIPv4 address %ws\n",
                    InetNtop(AF_INET, &sockaddr_ipv4->sin_addr, ipstringbuffer,
                             46));

            // We could also use the WSAAddressToString function
            // sockaddr_ip = (LPSOCKADDR) ptr->ai_addr;
            // The buffer length is changed by each call to WSAAddresstoString
            // So we need to set it for each iteration through the loop for safety
            // ipbufferlength = 46;
            // iRetval = WSAAddressToString(sockaddr_ip, (DWORD) ptr->ai_addrlen, NULL,
            //    ipstringbuffer, &ipbufferlength );
            // if (iRetval)
            //    wprintf(L"WSAAddressToString failed with %u\n", WSAGetLastError() );
            // else
            //    wprintf(L"\tIPv4 address %ws\n", ipstringbuffer);
            break;
        case AF_INET6:
            wprintf(L"AF_INET6 (IPv6)\n");
            // the InetNtop function is available on Windows Vista and later
            sockaddr_ipv6 = (struct sockaddr_in6 *) ptr->ai_addr;
            wprintf(L"\tIPv6 address %ws\n",
                    InetNtop(AF_INET6, &sockaddr_ipv6->sin6_addr,
                             ipstringbuffer, 46));

            // We could also use WSAAddressToString which also returns the scope ID
            // sockaddr_ip = (LPSOCKADDR) ptr->ai_addr;
            // The buffer length is changed by each call to WSAAddresstoString
            // So we need to set it for each iteration through the loop for safety
            // ipbufferlength = 46;
            //iRetval = WSAAddressToString(sockaddr_ip, (DWORD) ptr->ai_addrlen, NULL,
            //    ipstringbuffer, &ipbufferlength );
            //if (iRetval)
            //    wprintf(L"WSAAddressToString failed with %u\n", WSAGetLastError() );
            //else
            //    wprintf(L"\tIPv6 address %ws\n", ipstringbuffer);
            break;
        default:
            wprintf(L"Other %ld\n", ptr->ai_family);
            break;
        }
        wprintf(L"\tSocket type: ");
        switch (ptr->ai_socktype) {
        case 0:
```

```
            wprintf(L"Unspecified\n");
            break;
        case SOCK_STREAM:
            wprintf(L"SOCK_STREAM (stream)\n");
            break;
        case SOCK_DGRAM:
            wprintf(L"SOCK_DGRAM (datagram) \n");
            break;
        case SOCK_RAW:
            wprintf(L"SOCK_RAW (raw) \n");
            break;
        case SOCK_RDM:
            wprintf(L"SOCK_RDM (reliable message datagram)\n");
            break;
        case SOCK_SEQPACKET:
            wprintf(L"SOCK_SEQPACKET (pseudo-stream packet)\n");
            break;
        default:
            wprintf(L"Other %ld\n", ptr->ai_socktype);
            break;
        }
        wprintf(L"\tProtocol: ");
        switch (ptr->ai_protocol) {
        case 0:
            wprintf(L"Unspecified\n");
            break;
        case IPPROTO_TCP:
            wprintf(L"IPPROTO_TCP (TCP)\n");
            break;
        case IPPROTO_UDP:
            wprintf(L"IPPROTO_UDP (UDP) \n");
            break;
        default:
            wprintf(L"Other %ld\n", ptr->ai_protocol);
            break;
        }
        wprintf(L"\tLength of this sockaddr: %d\n", ptr->ai_addrlen);
        wprintf(L"\tCanonical name: %s\n", ptr->ai_canonname);
    }

    FreeAddrInfoEx(result);
    WSACleanup();

    return 0;
}
```

Note Ensure that the development environment targets the newest version of *Ws2tcpip.h* which includes structure and function definitions for **ADDRINFOEX** and GetAddrInfoEx, respectively.

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]

Header	ws2def.h (include Windows Server 2012, Windows 7 Windows Server 2008 R2)

See also

GetAddrInfoEx

addrinfo

addrinfoW

ADDRINFOW structure (ws2def.h)

10/22/2022 • 10 minutes to read • Edit Online

The **addrinfoW** structure is used by the GetAddrInfoW function to hold host address information.

Syntax

```
typedef struct addrinfoW {
  int              ai_flags;
  int              ai_family;
  int              ai_socktype;
  int              ai_protocol;
  size_t           ai_addrlen;
  PWSTR            ai_canonname;
  struct sockaddr  *ai_addr;
  struct addrinfoW *ai_next;
} ADDRINFOW, *PADDRINFOW;
```

Members

`ai_flags`

Type: **int**

Flags that indicate options used in the GetAddrInfoW function.

Supported values for the **ai_flags** member are defined in the *Winsock2.h* header file and can be a combination of the options listed in the following table.

VALUE	MEANING
AI_PASSIVE 0x01	The socket address will be used in a call to the bind function.
AI_CANONNAME 0x02	The canonical name is returned in the first **ai_canonname** member.
AI_NUMERICHOST 0x04	The *nodename* parameter passed to the GetAddrInfoW function must be a numeric string.
AI_ALL 0x0100	If this bit is set, a request is made for IPv6 addresses and IPv4 addresses with **AI_V4MAPPED**. This option is supported on Windows Vista and later.
AI_ADDRCONFIG 0x0400	The GetAddrInfoW will resolve only if a global address is configured. The IPv6 and IPv4 loopback address is not considered a valid global address. This option is only supported on Windows Vista and later.

AI_V4MAPPED 0x0800	If the GetAddrInfoW request for an IPv6 addresses fails, a name service request is made for IPv4 addresses and these addresses are converted to IPv4-mapped IPv6 address format. This option is supported on Windows Vista and later.
AI_NON_AUTHORITATIVE 0x04000	The address information can be from a non-authoritative namespace provider. This option is only supported on Windows Vista and later for the **NS_EMAIL** namespace.
AI_SECURE 0x08000	The address information is from a secure channel. This option is only supported on Windows Vista and later for the **NS_EMAIL** namespace.
AI_RETURN_PREFERRED_NAMES 0x010000	The address information is for a preferred name for a user. This option is only supported on Windows Vista and later for the **NS_EMAIL** namespace.
AI_FQDN 0x00020000	If a flat name (single label) is specified, GetAddrInfoW will return the fully qualified domain name that the name eventually resolved to. The fully qualified domain name is returned in the **ai_canonname** member. This is different than **AI_CANONNAME** bit flag that returns the canonical name registered in DNS which may be different than the fully qualified domain name that the flat name resolved to. Only one of the **AI_FQDN** and **AI_CANONNAME** bits can be set. The GetAddrInfoW function will fail if both flags are present with **EAI_BADFLAGS**. This option is supported on Windows 7, Windows Server 2008 R2, and later.
AI_FILESERVER 0x00040000	A hint to the namespace provider that the hostname being queried is being used in a file share scenario. The namespace provider may ignore this hint. This option is supported on Windows 7, Windows Server 2008 R2, and later.
AI_DISABLE_IDN_ENCODING 0x00080000	Disable the automatic International Domain Name encoding using Punycode in the name resolution functions called by the GetAddrInfoW function. This option is supported on Windows 8, Windows Server 2012, and later.

`ai_family`

Type: **int**

The address family. Possible values for the address family are defined in the *Winsock2.h* header file.

On the Windows SDK released for Windows Vista and later,, the organization of header files has changed and the 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. Note that the values for the AF_ address family and PF_ protocol family constants are identical (for example, **AF_UNSPEC** and **PF_UNSPEC**), so either constant can be used.

The following table lists common values for the address family although many other values are possible.

VALUE	MEANING
AF_UNSPEC 0	The address family is unspecified.
AF_INET 2	The Internet Protocol version 4 (IPv4) address family.
AF_NETBIOS 17	The NetBIOS address family. This address family is only supported if a Windows Sockets provider for NetBIOS is installed.
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 a Bluetooth adapter is installed on Windows Server 2003 or later.

`ai_socktype`

Type: **int**

The socket type. Possible values for the socket type are defined in the *Winsock2.h* include file.

The following table lists the possible values for the socket type supported for Windows Sockets 2.

VALUE	MEANING
SOCK_STREAM 1	Provides sequenced, reliable, two-way, connection-based byte streams with an OOB data transmission mechanism. Uses the Transmission Control Protocol (TCP) for the Internet address family (**AF_INET** or **AF_INET6**). If the **ai_family** member is **AF_IRDA**, then **SOCK_STREAM** is the only supported socket type.
SOCK_DGRAM 2	Supports datagrams, which are connectionless, unreliable buffers of a fixed (typically small) maximum length. Uses the User Datagram Protocol (UDP) for the Internet address family (**AF_INET** or **AF_INET6**).

SOCK_RAW 3	Provides a raw socket that allows an application to manipulate the next upper-layer protocol header. To manipulate the IPv4 header, the IP_HDRINCL socket option must be set on the socket. To manipulate the IPv6 header, the IPV6_HDRINCL socket option must be set on the socket.
SOCK_RDM 4	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 reliable multicast programming.
SOCK_SEQPACKET 5	Provides a pseudo-stream packet based on datagrams.

In Windows Sockets 2, new socket types were introduced. An application can dynamically discover the attributes of each available transport protocol through the WSAEnumProtocols function. So an application can determine the possible socket type and protocol options for an address family and use this information when specifying this parameter. Socket type definitions in the *Winsock2.h* and *Ws2def.h* header files will be periodically updated as new socket types, address families, and protocols are defined.

In Windows Sockets 1.1, the only possible socket types are **SOCK_DATAGRAM** and **SOCK_STREAM**.

`ai_protocol`

Type: **int**

The protocol type. The possible options are specific to the address family and socket type specified. Possible values for the **ai_protocol** are defined in *Winsock2.h* and the *Wsrm.h* header files.

On the Windows SDK released for Windows Vista and later,, the organization of header files has changed and this member can be one of the values from 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 for **ai_protocol**, the caller does not wish to specify a protocol and the service provider will choose the **ai_protocol** to use. For protocols other than IPv4 and IPv6, set **ai_protocol** to zero.

The following table lists common values for the **ai_protocol** member although many other values are possible.

VALUE	MEANING
IPPROTO_TCP 6	The Transmission Control Protocol (TCP). This is a possible value when the **ai_family** member is **AF_INET** or **AF_INET6** and the **ai_socktype** member is **SOCK_STREAM**.
IPPROTO_UDP 17	The User Datagram Protocol (UDP). This is a possible value when the **ai_family** member is **AF_INET** or **AF_INET6** and the *type* parameter is **SOCK_DGRAM**.
IPPROTO_RM 113	The PGM protocol for reliable multicast. This is a possible value when the **ai_family** member is **AF_INET** and the **ai_socktype** member is **SOCK_RDM**. On the Windows SDK released for Windows Vista and later, this value is also called **IPPROTO_PGM**.

If the **ai_family** member is **AF_IRDA**, then the **ai_protocol** must be 0.

`ai_addrlen`

Type: **size_t**

The length, in bytes, of the buffer pointed to by the **ai_addr** member.

`ai_canonname`

Type: **PWSTR**

The canonical name for the host.

`ai_addr`

Type: **struct sockaddr***

A pointer to a sockaddr structure. The **ai_addr** member in each returned ADDRINFOW structure points to a filled-in socket address structure. The length, in bytes, of each returned **ADDRINFOW** structure is specified in the **ai_addrlen** member.

`ai_next`

Type: **struct addrinfoW***

A pointer to the next structure in a linked list. This parameter is set to **NULL** in the last **addrinfoW** structure of a linked list.

Remarks

The **addrinfoW** structure is used by the Unicode GetAddrInfoW function to hold host address information.

The addrinfo structure is ANSI version of this structure used by the ANSI getaddrinfo function.

Macros in the *Ws2tcpip.h* header file define a **ADDRINFOT** structure and a mixed-case function name of **GetAddrInfo**. The **GetAddrInfo** function should be called with the *nodename* and *servname* parameters of a pointer of type **TCHAR** and the *hints* and *res* parameters of a pointer of type **ADDRINFOT**. When UNICODE or _UNICODE is defined, **ADDRINFOT** is defined to the **addrinfoW** structure and **GetAddrInfo** is defined to GetAddrInfoW, the Unicode version of this function. When UNICODE or _UNICODE is not defined, **ADDRINFOT** is defined to the addrinfo structure and **GetAddrInfo** is defined to getaddrinfo, the ANSI version of this function.

Upon a successful call to GetAddrInfoW, a linked list of ADDRINFOW structures is returned in the *ppResult* parameter passed to the **GetAddrInfoW** function. The list can be processed by following the pointer provided in the **ai_next** member of each returned **ADDRINFOW** structure until a **NULL** pointer is encountered. In each returned **ADDRINFOW** structure, the **ai_family**, **ai_socktype**, and **ai_protocol** members correspond to respective arguments in a socket or WSASocket function call. Also, the **ai_addr** member in each returned **ADDRINFOW** structure points to a filled-in socket address structure, the length of which is specified in its **ai_addrlen** member.

Examples

The following code example shows how to use the **addrinfoW** structure.

```
#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>

#pragma comment(lib, "Ws2_32.lib")

int __cdecl wmain(int argc, wchar_t ** argv)
{
//--------------------------------
// Declare and initialize variables.
    WSADATA wsaData;
    int iResult;

    ADDRINFOW *result = NULL;
    ADDRINFOW *ptr = NULL;
    ADDRINFOW hints;

    DWORD dwRetval = 0;
    int i = 1;

    struct sockaddr_in *sockaddr_ipv4;
    struct sockaddr_in6 *sockaddr_ipv6;
//    LPSOCKADDR sockaddr_ip;

    wchar_t ipstringbuffer[46];

    // Validate the parameters
    if (argc != 3) {
        wprintf(L"usage: %ws <hostname> <servicename>\n", argv[0]);
        wprintf(L"       provides protocol-independent translation\n");
        wprintf(L"       from a host name to an IP address\n");
        wprintf(L"%ws example usage\n", argv[0]);
        wprintf(L"   %ws www.contoso.com 0\n", argv[0]);
        return 1;
    }
    // Initialize Winsock
    iResult = WSAStartup(MAKEWORD(2, 2), &wsaData);
    if (iResult != 0) {
        wprintf(L"WSAStartup failed: %d\n", iResult);
        return 1;
    }
//--------------------------------
// Setup the hints address info structure
// which is passed to the GetAddrInfoW() function
    memset(&hints, 0, sizeof (hints));
    hints.ai_family = AF_UNSPEC;
    hints.ai_socktype = SOCK_STREAM;
    hints.ai_protocol = IPPROTO_TCP;

    wprintf(L"Calling GetAddrInfoW with following parameters:\n");
    wprintf(L"\tName = %ws\n", argv[1]);
    wprintf(L"\tServiceName (or port) = %ws\n\n", argv[2]);

//--------------------------------
// Call GetAddrInfoW(). If the call succeeds,
// the aiList variable will hold a linked list
// of addrinfo structures containing response
// information about the host
    dwRetval = GetAddrInfoW(argv[1], argv[2], &hints, &result);

    if (dwRetval != 0) {
        wprintf(L"GetAddrInfoW failed with error: %d\n", dwRetval);
        WSACleanup();
        return 1;
    }
    wprintf(L"GetAddrInfoW returned success\n");
```

```
    // Retrieve each address and print out the hex bytes
    for (ptr = result; ptr != NULL; ptr = ptr->ai_next) {

        wprintf(L"GetAddrInfoW response %d\n", i++);
        wprintf(L"\tFlags: 0x%x\n", ptr->ai_flags);
        wprintf(L"\tFamily: ");
        switch (ptr->ai_family) {
        case AF_UNSPEC:
            wprintf(L"Unspecified\n");
            break;
        case AF_INET:
            wprintf(L"AF_INET (IPv4)\n");
            // the InetNtop function is available on Windows Vista and later
            sockaddr_ipv4 = (struct sockaddr_in *) ptr->ai_addr;
            wprintf(L"\tIPv4 address %ws\n",
                    InetNtop(AF_INET, &sockaddr_ipv4->sin_addr, ipstringbuffer,
                             46));

            // We could also use the WSAAddressToString function
            // sockaddr_ip = (LPSOCKADDR) ptr->ai_addr;
            // The buffer length is changed by each call to WSAAddresstoString
            // So we need to set it for each iteration through the loop for safety
            // ipbufferlength = 46;
            // iRetval = WSAAddressToString(sockaddr_ip, (DWORD) ptr->ai_addrlen, NULL,
            //    ipstringbuffer, &ipbufferlength );
            // if (iRetval)
            //    wprintf(L"WSAAddressToString failed with %u\n", WSAGetLastError() );
            // else
            //    wprintf(L"\tIPv4 address %ws\n", ipstringbuffer);
            break;
        case AF_INET6:
            wprintf(L"AF_INET6 (IPv6)\n");
            // the InetNtop function is available on Windows Vista and later
            sockaddr_ipv6 = (struct sockaddr_in6 *) ptr->ai_addr;
            wprintf(L"\tIPv6 address %ws\n",
                    InetNtop(AF_INET6, &sockaddr_ipv6->sin6_addr,
                             ipstringbuffer, 46));

            // We could also use WSAAddressToString which also returns the scope ID
            // sockaddr_ip = (LPSOCKADDR) ptr->ai_addr;
            // The buffer length is changed by each call to WSAAddresstoString
            // So we need to set it for each iteration through the loop for safety
            // ipbufferlength = 46;
            //iRetval = WSAAddressToString(sockaddr_ip, (DWORD) ptr->ai_addrlen, NULL,
            //    ipstringbuffer, &ipbufferlength );
            //if (iRetval)
            //    wprintf(L"WSAAddressToString failed with %u\n", WSAGetLastError() );
            //else
            //    wprintf(L"\tIPv6 address %ws\n", ipstringbuffer);
            break;
        default:
            wprintf(L"Other %ld\n", ptr->ai_family);
            break;
        }
        wprintf(L"\tSocket type: ");
        switch (ptr->ai_socktype) {
        case 0:
            wprintf(L"Unspecified\n");
            break;
        case SOCK_STREAM:
            wprintf(L"SOCK_STREAM (stream)\n");
            break;
        case SOCK_DGRAM:
            wprintf(L"SOCK_DGRAM (datagram) \n");
            break;
        case SOCK_RAW:
            wprintf(L"SOCK_RAW (raw) \n");
            break;
        case SOCK_RDM:
```

```
        case SOCK_RDM:
            wprintf(L"SOCK_RDM (reliable message datagram)\n");
            break;
        case SOCK_SEQPACKET:
            wprintf(L"SOCK_SEQPACKET (pseudo-stream packet)\n");
            break;
        default:
            wprintf(L"Other %ld\n", ptr->ai_socktype);
            break;
        }
        wprintf(L"\tProtocol: ");
        switch (ptr->ai_protocol) {
        case 0:
            wprintf(L"Unspecified\n");
            break;
        case IPPROTO_TCP:
            wprintf(L"IPPROTO_TCP (TCP)\n");
            break;
        case IPPROTO_UDP:
            wprintf(L"IPPROTO_UDP (UDP) \n");
            break;
        default:
            wprintf(L"Other %ld\n", ptr->ai_protocol);
            break;
        }
        wprintf(L"\tLength of this sockaddr: %d\n", ptr->ai_addrlen);
        wprintf(L"\tCanonical name: %s\n", ptr->ai_canonname);
    }

    FreeAddrInfo(result);
    WSACleanup();

    return 0;
}
```

Requirements

Minimum supported client	Windows Vista, Windows XP with SP2 [desktop apps only]
Minimum supported server	Windows Server 2003 [desktop apps only]
Header	ws2def.h (include Windows Server 2012, Windows 7 Windows Server 2008 R2)

See also

GetAddrInfoEx

GetAddrInfoW

WSAEnumProtocols

addrinfo

addrinfoex

addrinfoex2

getaddrinfo

sockaddr

CSADDR_INFO structure (ws2def.h)

10/22/2022 • 3 minutes to read • Edit Online

The **CSADDR_INFO** structure contains Windows Sockets address information for a socket, network service, or namespace provider.

Syntax

```
typedef struct _CSADDR_INFO {
  SOCKET_ADDRESS LocalAddr;
  SOCKET_ADDRESS RemoteAddr;
  INT            iSocketType;
  INT            iProtocol;
} CSADDR_INFO, *PCSADDR_INFO, *LPCSADDR_INFO;
```

Members

`LocalAddr`

Type: **SOCKET_ADDRESS**

The Windows Sockets local address.

In a client application, pass this address to the **bind** function to obtain access to a network service.

In a network service, pass this address to the **bind** function so that the service is bound to the appropriate local address.

`RemoteAddr`

Type: **SOCKET_ADDRESS**

Windows Sockets remote address.

There are several uses for this remote address:

- You can use this remote address to connect to the service through the connect function. This is useful if an application performs send/receive operations that involve connection-oriented protocols.
- You can use this remote address with the sendto function when you are communicating over a connectionless (datagram) protocol. If you are using a connectionless protocol, such as UDP, **sendto** is typically the way you pass data to the remote system.

`iSocketType`

Type: **INT**

The type of Windows socket. Possible values for the socket type are defined in the *Winsock2.h* header file.

The following table lists the possible values supported for Windows Sockets 2:

VALUE	MEANING

SOCK_STREAM	A stream socket. This is a protocol that sends data as a stream of bytes, with no message boundaries. This socket type 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	A datagram socket. This socket type 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). Services use recvfrom function to obtain datagrams. The listen and accept functions do not work with datagrams.
SOCK_RDM	A reliable message datagram socket. This socket type preserves message boundaries in data. An example of this type is the Pragmatic General Multicast (PGM) multicast protocol implementation in Windows, often referred to as reliable multicast programming.
SOCK_SEQPACKET	A sequenced packet stream socket. This socket type provides a pseudo-stream packet based on datagrams.

`iProtocol`

Type: **INT**

The protocol used. The possible options for the *protocol* parameter are specific to the address family and socket type specified. Possible values are defined in the *Winsock2.h* and *Wsrm.h* header files.

On the Windows SDK released for Windows Vista and later, the organization of header files has changed and this parameter can be one of the values from 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.

The table below lists common values for the *protocol* although many other values are possible.

PROTOCOL	MEANING
IPPROTO_TCP 6	The Transmission Control Protocol (TCP). This is a possible value when the address family is **AF_INET** or **AF_INET6** and the **iSocketType** member is **SOCK_STREAM**.
IPPROTO_UDP 17	The User Datagram Protocol (UDP). This is a possible value when the address family is **AF_INET** or **AF_INET6** and the **iSocketType** member is **SOCK_DGRAM**.
IPPROTO_RM 113	The PGM protocol for reliable multicast. This is a possible value when the address family is **AF_INET** and the **iSocketType** member is **SOCK_RDM**. On the Windows SDK released for Windows Vista and later, this value is also called **IPPROTO_PGM**.

Remarks

The GetAddressByName function obtains Windows Sockets address information using **CSADDR_INFO** structures.

The getsockopt function called with the SO_BSP_STATE socket option retrieves a **CSADDR_INFO** structure for the specified socket.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	ws2def.h (include Nspapi.h)

See also

GetAddressByName

SOCKET_ADDRESS

SO_BSP_STATE

bind

connect

getsockopt

recv

send

sendto

SOCKET_ADDRESS structure (ws2def.h)

10/22/2022 • 2 minutes to read • Edit Online

The **SOCKET_ADDRESS** structure stores protocol-specific address information.

Syntax

```
typedef struct _SOCKET_ADDRESS {
  LPSOCKADDR lpSockaddr;
  INT        iSockaddrLength;
} SOCKET_ADDRESS, *PSOCKET_ADDRESS, *LPSOCKET_ADDRESS;
```

Members

`lpSockaddr`

A pointer to a socket address represented as a SOCKADDR structure.

`iSockaddrLength`

The length, in bytes, of the socket address.

Remarks

The SOCKADDR structure pointed to by the **lpSockaddr** member varies depending on the protocol or address family selected. For example, the **sockaddr_in6** structure is used for an IPv6 socket address while the **sockaddr_in4** structure is used for an IPv4 socket address. The address family is the first member of all of the **SOCKADDR** structures. The address family is used to determine which structure is used.

On the Microsoft Windows Software Development Kit (SDK) released for Windows Vista and later, the organization of header files has changed and the **SOCKET_ADDRESS** structure 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.

Requirements

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

SOCKADDR

SOCKET_ADDRESS_LIST

Using SIO_ADDRESS_LIST_SORT

WSAIoctl

LPWSPIoctl

SOCKET_PROCESSOR_AFFINITY structure (ws2def.h)

10/22/2022 • 2 minutes to read • Edit Online

The **SOCKET_PROCESSOR_AFFINITY** structure contains the association between a socket and an RSS processor core and NUMA node..

Syntax

```
typedef struct _SOCKET_PROCESSOR_AFFINITY {
  PROCESSOR_NUMBER Processor;
  USHORT           NumaNodeId;
  USHORT           Reserved;
} SOCKET_PROCESSOR_AFFINITY, *PSOCKET_PROCESSOR_AFFINITY;
```

Members

`Processor`

A structure to represent a system wide processor number. This PROCESSOR_NUMBER structure contains a group number and relative processor number within the group.

`NumaNodeId`

The NUMA node ID.

`Reserved`

A value reserved for future use.

Remarks

The **SOCKET_PROCESSOR_AFFINITY** structure 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.

Requirements

Minimum supported client	Windows 8 [desktop apps only]
Minimum supported server	Windows Server 2012 [desktop apps only]
Header	ws2def.h

See also

PROCESSOR_NUMBER

SIO_QUERY_RSS_PROCESSOR_INFO

WSABUF structure (ws2def.h)

10/22/2022 • 2 minutes to read • Edit Online

The **WSABUF** structure enables the creation or manipulation of a data buffer used by some Winsock functions.

Syntax

```
typedef struct _WSABUF {
  ULONG len;
  CHAR  *buf;
} WSABUF, *LPWSABUF;
```

Members

`len`

The length of the buffer, in bytes.

`buf`

A pointer to the buffer.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	ws2def.h (include Winsock2.h)

WSAMSG structure (ws2def.h)

10/22/2022 • 5 minutes to read • Edit Online

The **WSAMSG** structure is used with the LPFN_WSARECVMSG (WSARecvMsg) and WSASendMsg functions to store address and optional control information about connected and unconnected sockets as well as an array of buffers used to store message data.

Syntax

```
typedef struct _WSAMSG {
  LPSOCKADDR name;
  INT        namelen;
  LPWSABUF   lpBuffers;
#if ...
  ULONG      dwBufferCount;
#else
  DWORD      dwBufferCount;
#endif
  WSABUF     Control;
#if ...
  ULONG      dwFlags;
#else
  DWORD      dwFlags;
#endif
} WSAMSG, *PWSAMSG, *LPWSAMSG;
```

Members

`name`

Type: **LPSOCKADDR**

A pointer to a SOCKET_ADDRESS structure that stores information about the remote address. Used only with unconnected sockets.

`namelen`

Type: **INT**

The length, in bytes, of the SOCKET_ADDRESS structure pointed to in the **pAddr** member. Used only with unconnected sockets.

`lpBuffers`

Type: **LPWSABUF**

An array of WSABUF structures used to receive the message data. The capability of the **lpBuffers** member to contain multiple buffers enables the use of scatter/gather I/O.

`dwBufferCount`

Type: **DWORD**

The number of buffers pointed to in the **lpBuffers** member.

`Control`

Type: **WSABUF**

A structure of WSABUF type used to specify optional control data. See Remarks.

`dwFlags`

Type: **DWORD**

One or more control flags, specified as the logical **OR** of values. The possible values for **dwFlags** member on input are defined in the Winsock2.h header file. The possible values for **dwFlags** member on output are defined in the Ws2def.h header file which is automatically included by the Winsock2.h header file.

FLAGS ON INPUT	MEANING
MSG_PEEK	Peek at the incoming data. The data is copied into the buffer, but is not removed from the input queue. This flag is valid only for non-overlapped sockets.

FLAG RETURNED	MEANING
MSG_BCAST	The datagram was received as a link-layer broadcast or with a destination IP address that is a broadcast address.
MSG_MCAST	The datagram was received with a destination IP address that is a multicast address.
MSG_TRUNC	The datagram was truncated. More data was present than the process allocated room for.
MSG_CTRUNC	The control (ancillary) data was truncated. More control data was present than the process allocated room for.

Remarks

In the Microsoft Windows Software Development Kit (SDK), the version of this structure for use on Windows Vista is defined with the data type for the **dwBufferCount** and **dwFlags** members as a **ULONG**. When compiling an application if the target platform is Windows Vista and later (**NTDDI_VERSION >= NTDDI_LONGHORN, _WIN32_WINNT >= 0x0600**, or **WINVER >= 0x0600**), the data type for the **dwBufferCount** and **dwFlags** members is a **ULONG**.

Windows Server 2003 and Windows XP: When compiling an application, the data type for the **dwBufferCount** and **dwFlags** members is a **DWORD**.

On the Windows SDK released for Windows Vista and later, the **WSAMSG** structure 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

If the datagram or control data is truncated during the transmission, the function being used in association with the **WSAMSG** structure returns SOCKET_ERROR and a call to the WSAGetLastError function returns WSAEMSGSIZE. It is up to the application to determine what was truncated by checking for MSG_TRUNC and/or MSG_CTRUNC flags.

Use of the control member

The following table summarizes the various uses of control data available for use in the *Control* member for

IPv4 and IPv6.

PROTOCOL	CMSG_LEVEL	CMSG_TYPE	DESCRIPTION
IPv4	IPPROTO_IP	IP_ORIGINAL_ARRIVAL_IF	Receives the original IPv4 arrival interface where the packet was received for datagram sockets. This control data is used by firewalls when a Teredo, 6to4, or ISATAP tunnel is used for IPv4 NAT traversal. The cmsg_data[] member in the **WSAMSG** structure is a **ULONG** that contains the IF_INDEX defined in the Ifdef.h header file. For more information, see the IPPROTO_IP Socket Options for the IP_ORIGINAL_ARRIVAL_IF socket option. **Windows Server 2008, Windows Vista, Windows Server 2003 and Windows XP:** The IP_ORIGINAL_ARRIVAL_IF **cmsg_type** is not supported.
IPv4	IPPROTO_IP	IP_PKTINFO	Specifies/receives packet information for an IPv4 socket. For more information, see the IP_PKTINFO socket option.
IPv4	IPPROTO_IP	IP_ECN	Specifies/receives ECN codepoint in the Type of Service (TOS) IPv4 header field. For more info, see WSASetRecvIPEcn.
IPv6	IPPROTO_IPV6	IPV6_DSTOPTS	Specifies/receives destination options.
IPv6	IPPROTO_IPV6	IPV6_HOPLIMIT	Specifies/receives hop limit. For more information, see the IPPROTO_IPV6 Socket Options for the IPV6_HOPLIMIT socket option.
IPv6	IPPROTO_IPV6	IPV6_HOPOPTS	Specifies/receives hop-by-hop options.
IPv6	IPPROTO_IPV6	IPV6_NEXTHOP	Specifies next-hop address.

PROTOCOL	CMSG_LEVEL	CMSG_TYPE	DESCRIPTION
IPv6	IPPROTO_IPV6	IPV6_PKTINFO	Specifies/receives packet information for an IPv6 socket. For more information, see the IPV6_PKTINFO socket option.
IPv6	IPPROTO_IPV6	IPV6_RTHDR	Specifies/receives routing header.
IPv6	IPPROTO_IPV6	IPV6_ECN	Specifies/receives ECN codepoint in the Traffic Class IPv6 header field. For more information, see WSASetRecvIPEcn.

Control data is made up of one or more control data objects, each beginning with a **WSACMSGHDR** structure, defined as the following.

```
struct wsacmsghdr {
  UINT        cmsg_len;
  INT         cmsg_level;
  INT         cmsg_type;
  /* followed by UCHAR cmsg_data[] */
} WSACMSGHDR;
```

> **Note** The transport, not the application, fills out the header information in the **WSACMSGHDR** structure. The application simply sets the needed socket options and provides the adequate buffer size.

The members of the **WSACMSGHDR** structure are as follows:

TERM	DESCRIPTION
cmsg_len	The number of bytes of data starting from the beginning of the **WSACMSGHDR** to the end of data (excluding padding bytes that may follow data).
cmsg_level	The protocol that originated the control information.
cmsg_type	The protocol-specific type of control information.

The following macros are used to navigate the data objects:

```
#define LPCMSGHDR *WSA_CMSG_FIRSTHDR(LPWSAMSG msg);
```

Returns a pointer to the first control data object. Returns a **NULL** pointer if there is no control data in the **WSAMSG** structure, such as when the **Control** member is a **NULL** pointer.

```
#define LPCMSGHDR *WSA_CMSG_NXTHDR(LPWSAMSG msg, LPWSACMSGHDR cmsg);
```

Returns a pointer to the next control data object, or **NULL** if there are no more data objects. If the *pcmsg* parameter is **NULL**, a pointer to the first control data object is returned.

```
#define UCHAR *WSA_CMSG_DATA(LPWSACMSGHDR pcmsg);
```

Returns a pointer to the first byte of data (referred to as the **cmsg_data** member, though it is not defined in the structure).

```
#define UINT WSA_CMSG_SPACE(UINT length);
```

Returns the total size of a control data object, given the amount of data. Used to allocate the correct amount of buffer space. Includes alignment padding.

```
#define UINT WSA_CMSG_LEN(UINT length);
```

Returns the value in **cmsg_len** given the amount of data. Includes alignment padding.

Requirements

Minimum supported client	Windows XP [desktop apps only]
Minimum supported server	Windows Server 2003 [desktop apps only]
Header	ws2def.h (include Winsock2.h)

See also

IPV6_PKTINFO

IP_PKTINFO

SOCKET_ADDRESS

WSABUF

WSARecv

WSARecvFrom

LPFN_WSARECVMSG (WSARecvMsg)

WSASendMsg

ws2ipdef.h header

10/22/2022 • 2 minutes to read • Edit Online

This header is used by multiple technologies. For more information, see:

- IP Helper
- Network Drivers, Windows Vista and Later
- Windows Sockets 2

ws2ipdef.h contains the following programming interfaces:

Structures

GROUP_FILTER	Provides multicast filtering parameters for multicast IPv6 or IPv4 addresses.
GROUP_REQ	Provides multicast group information for IPv6 or IPv4 addresses.
GROUP_SOURCE_REQ	Provides multicast group information for IPv6 or IPv4 addresses that includes the source IP address.
ICMP_ERROR_INFO	Used to store received ICMP error information.
IN_PKTINFO	The in_pktinfo structure is used to store received packet address information, and is used by Windows to return information about received packets and also allows specifying the local IPv4 address to use for sending packets.
IN6_PKTINFO	The in6_pktinfo structure is used to store received IPv6 packet address information, and is used by Windows to return information about received packets and also allows specifying the local IPv6 address to use for sending packets.
INTERFACE_INFO	The INTERFACE_INFO structure is used in conjunction with the SIO_GET_INTERFACE_LIST ioctl command to obtain information about an interface IP address.
INTERFACE_INFO_EX	The INTERFACE_INFO_EX structure is used in conjunction with the SIO_GET_INTERFACE_LIST IOCTL command to obtain information about an interface IP address.

IP_MREQ	The ip_mreq structure provides multicast group information for IPv4 addresses.
IP_MREQ_SOURCE	The ip_mreq_source structure provides multicast group information for IPv4 addresses.
IP_MSFILTER	The ip_msfilter structure provides multicast filtering parameters for IPv4 addresses.
IPV6_MREQ	The ipv6_mreq structure provides multicast group information for IPv6 addresses.
sockaddr_gen	Provides generic socket address information, and is used with the INTERFACE_INFO structure.
SOCKADDR_IN6_LH	The SOCKADDR_IN6_LH (ws2ipdef.h) structure specifies a transport address and port for the AF_INET6 address family.
sockaddr_in6_old	The sockaddr_in6_old (ws2ipdef.h) structure varies depending on the protocol selected.
SOCKADDR_IN6_PAIR	Contains pointers to a pair of IP addresses that represent a source and destination address pair.
SOCKADDR_IN6_W2KSP1	The SOCKADDR_IN6_W2KSP1 (ws2ipdef.h) structure specifies a transport address and port for the AF_INET6 address family.
SOCKADDR_INET	Contains an IPv4, an IPv6 address, or an address family.

Enumerations

MULTICAST_MODE_TYPE	Specifies the filter mode for multicast group addresses.

GROUP_FILTER structure (ws2ipdef.h)

10/22/2022 • 2 minutes to read • Edit Online

The **GROUP_FILTER** structure provides multicast filtering parameters for multicast IPv6 or IPv4 addresses.

Syntax

```
typedef struct group_filter {
  ULONG               gf_interface;
  SOCKADDR_STORAGE    gf_group;
  MULTICAST_MODE_TYPE gf_fmode;
  ULONG               gf_numsrc;
  SOCKADDR_STORAGE    gf_slist[1];
} GROUP_FILTER, *PGROUP_FILTER;
```

Members

`gf_interface`

The interface index of the local interface for the multicast group to filter.

`gf_group`

The multicast address group that should be filtered. This may be either an IPv6 or IPv4 multicast address.

`gf_fmode`

The multicast filter mode.

This member can be one of the values from the MULTICAST_MODE_TYPE enumeration type defined in the *Ws2ipdef.h* header file. This member determines if the list of IP addresses in the **gf_numsrc** member should be included or excluded.

VALUE	MEANING
MCAST_INCLUDE	The filter contains a list of IP addresses to include.
MCAST_EXCLUDE	The filter contains a list of IP addresses to exclude.

`gf_numsrc`

The number of multicast filter source address entries in the **gf_slist** member.

`gf_slist[1]`

An array of SOCKADDR_STORAGE structures specifying the multicast source addresses to include or exclude. These IP addresses may be either IPv6 or IPv4 addresses, but they must be the same address family (IPv6 or IPv4) as the address specified in the **gf_group** member..

Remarks

The **GROUP_FILTER** structure is used with either IPv6 or IPv4 multicast addresses. The **GROUP_FILTER** structure is passed as an argument for the **SIOCGMSFILTER** and **SIOCSMSFILTER** IOCTLs.

The **GROUP_FILTER** structure and related structures used for multicast programming are based on IETF recommendations in sections 5 and 8.2 of RFC 3768. For more information, see http://www.ietf.org/rfc/rfc3678.txt.

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. These IP agnostic options use the GROUP_REQ and the GROUP_SOURCE_REQ structures and are the preferred socket options for multicast programming on Windows Vista and later.

The GetAdaptersAddresses function can be used to obtain interface index information required for the *gf_interface* member.

The **GROUP_FILTER** structure and the Ioctls that use this structure are only valid on datagram and raw sockets (the socket type must be **SOCK_DGRAM** or **SOCK_RAW**).

The **GROUP_FILTER** structure is defined in the *Ws2ipdef.h* header file which is automatically included in the *Ws2tcpip.h* header file. The *Ws2ipdef.h* header files should never be used directly.

Requirements

Minimum supported client	None supported
Minimum supported server	Windows Server 2008 [desktop apps only]
Header	ws2ipdef.h (include Ws2tcpip.h)

See also

GROUP_REQ

GROUP_SOURCE_REQ

GetAdaptersAddresses

MULTICAST_MODE_TYPE

Multicast Programming

SOCKADDR_STORAGE

Socket Options

ip_mreq

ip_msfilter

ipv6_mreq

GROUP_REQ structure (ws2ipdef.h)

10/22/2022 • 2 minutes to read • Edit Online

The **GROUP_REQ** structure provides multicast group information for IPv6 or IPv4 addresses.

Syntax

```
typedef struct group_req {
  ULONG            gr_interface;
  SOCKADDR_STORAGE gr_group;
} GROUP_REQ, *PGROUP_REQ;
```

Members

`gr_interface`

The interface index of the local interface on which the multicast group should be joined or dropped.

`gr_group`

The address of the multicast group. This may be either an IPv6 or IPv4 multicast address.

Remarks

The **GROUP_REQ** structure is used with either IPv6 or IPv4 multicast addresses. The **GROUP_REQ** structure is used with the MCAST_JOIN_GROUP and MCAST_LEAVE_GROUP socket options.

The **GROUP_REQ** structure and related structures used for multicast programming are based on IETF recommendations in sections 5 and 8.2 of RFC 3768. For more information, see http://www.ietf.org/rfc/rfc3678.txt.

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. These IP agnostic options use the **GROUP_REQ** and the GROUP_SOURCE_REQ structures and are the preferred socket options for multicast programming on Windows Vista and later.

The GetAdaptersAddresses function can be used to obtain interface index information required for the *gr_interface* member.

The **GROUP_REQ** structure and the socket options that use this structure are only valid on datagram and raw sockets (the socket type must be **SOCK_DGRAM** or **SOCK_RAW**).

The **GROUP_REQ** structure is defined in the *Ws2ipdef.h* header file which is automatically included in the *Ws2tcpip.h* header file. The *Ws2ipdef.h* header files should never be used directly.

Requirements

Minimum supported client	None supported

Minimum supported server	Windows Server 2008 [desktop apps only]
Header	ws2ipdef.h (include Ws2tcpip.h)

See also

GROUP_FILTER

GROUP_SOURCE_REQ

GetAdaptersAddresses

Multicast Programming

SOCKADDR_STORAGE

Socket Options

ip_mreq

ipv6_mreq

GROUP_SOURCE_REQ structure (ws2ipdef.h)

10/22/2022 • 2 minutes to read • Edit Online

The **GROUP_SOURCE_REQ** structure provides multicast group information for IPv6 or IPv4 addresses that includes the source IP address.

Syntax

```
typedef struct group_source_req {
  ULONG            gsr_interface;
  SOCKADDR_STORAGE gsr_group;
  SOCKADDR_STORAGE gsr_source;
} GROUP_SOURCE_REQ, *PGROUP_SOURCE_REQ;
```

Members

`gsr_interface`

The interface index of the local interface on which the multicast group should be joined, dropped, blocked, or unblocked.

`gsr_group`

The address of the multicast group. This may be either an IPv6 or IPv4 multicast address.

`gsr_source`

The source address that should be used. This may be either an IPv6 or IPv4 multicast address, but it must be the same address family (IPv6 or IPv4) as the address specified in the **gsr_group** member.

Remarks

The **GROUP_SOURCE_REQ** structure is used with either IPv6 or IPv4 multicast addresses. The **GROUP_SOURCE_REQ** structure is used with the MCAST_BLOCK_SOURCE, MCAST_JOIN_SOURCE_GROUP, MCAST_LEAVE_SOURCE_GROUP, and MCAST_UNBLOCK_SOURCE socket options.

The **GROUP_SOURCE_REQ** structure and related structures used for multicast programming are based on IETF recommendations in sections 5 and 8.2 of RFC 3768. For more information, see http://www.ietf.org/rfc/rfc3678.txt.

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. These IP agnostic options use the GROUP_REQ and the **GROUP_SOURCE_REQ** structures and are the preferred socket options for multicast programming on Windows Vista and later.

The GetAdaptersAddresses function can be used to obtain interface index information required for the *gsr_interface* member.

The **GROUP_SOURCE_REQ** structure and the socket options that use this structure are only valid on datagram and raw sockets (the socket type must be **SOCK_DGRAM** or **SOCK_RAW**).

The **GROUP_SOURCE_REQ** structure is defined in the *Ws2ipdef.h* header file which is automatically included in the *Ws2tcpip.h* header file. The *Ws2ipdef.h* header files should never be used directly.

Requirements

Minimum supported client	None supported
Minimum supported server	Windows Server 2008 [desktop apps only]
Header	ws2ipdef.h (include Ws2tcpip.h)

See also

GROUP_FILTER

GROUP_REQ

GetAdaptersAddresses

Multicast Programming

SOCKADDR_STORAGE

Socket Options

ip_mreq

ipv6_mreq

ICMP_ERROR_INFO structure (ws2ipdef.h)

10/22/2022 • 2 minutes to read • Edit Online

Used to store received ICMP error information.

Syntax

```
typedef struct icmp_error_info {
  SOCKADDR_INET srcaddress;
  IPPROTO       protocol;
  UINT8         type;
  UINT8         code;
} ICMP_ERROR_INFO, *PICMP_ERROR_INFO;
```

Members

`srcaddress`

Type: **SOCKADDR_INET**

The source IP address of the ICMP error.

`protocol`

Type: **IPPROTO**

The protocol of the ICMP error (either **IPPROTO_ICMP** or **IPPROTO_ICMPV6**).

`type`

Type: **UINT8**

The ICMP error type.

`code`

Type: **UINT8**

The ICMP error code.

Requirements

Minimum supported client	Windows 10, version 2004 (10.0; Build 19041)
Minimum supported server	Windows Server, version 2004 (10.0; Build 19041)
Header	ws2ipdef.h (include ws2tcpip.h)

IN_PKTINFO structure (ws2ipdef.h)

10/22/2022 • 2 minutes to read • Edit Online

The **in_pktinfo** structure is used to store received packet address information, and is used by Windows to return information about received packets and also allows specifying the local IPv4 address to use for sending packets.

Syntax

```
typedef struct in_pktinfo {
  IN_ADDR ipi_addr;
  ULONG   ipi_ifindex;
} IN_PKTINFO, *PIN_PKTINFO;
```

Members

`ipi_addr`

The destination IPv4 address from the IP header of the received packet when used with the LPFN_WSARECVMSG (WSARecvMsg) function. The local source IPv4 address to set in the IP header when used with the WSASendMsg function.

`ipi_ifindex`

The interface on which the packet was received when used with the LPFN_WSARECVMSG (WSARecvMsg) function. The interface on which the packet should be sent when used with the WSASendMsg function.

Remarks

If the IP_PKTINFO socket option is set on a socket of type **SOCK_DGRAM** or **SOCK_RAW**, one of the control data objects returned by the LPFN_WSARECVMSG (WSARecvMsg) function will contain an **in_pktinfo** structure used to store received packet address information.

On an IPv4 socket of type **SOCK_DGRAM** or **SOCK_RAW**, an application can specific the local IP address to use for sending with the WSASendMsg function. One of the control data objects passed in the WSAMSG structure to the **WSASendMsg** function may contain an **in_pktinfo** structure used to specify the local IPv4 address to use for sending.

On the Microsoft Windows Software Development Kit (SDK) released for Windows Vista and later, the organization of header files has changed and the **in_pktinfo** structure is defined in the *Ws2ipdef.h* header file which is automatically included in the *Ws2tcpip.h* header file. The *Ws2ipdef.h* header files should never be used directly.

Requirements

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 for IPv6 Winsock Applications

IPPROTO_IP Socket Options

IPV6_PKTINFO

IP_PKTINFO

WSAMSG

LPFN_WSARECVMSG (WSARecvMsg)

WSASendMsg

in6_pktinfo

IN6_PKTINFO structure (ws2ipdef.h)

10/22/2022 • 2 minutes to read • Edit Online

The **in6_pktinfo** structure is used to store received IPv6 packet address information, and is used by Windows to return information about received packets and also allows specifying the local IPv6 address to use for sending packets.

Syntax

```
typedef struct in6_pktinfo {
  IN6_ADDR ipi6_addr;
  ULONG    ipi6_ifindex;
} IN6_PKTINFO, *PIN6_PKTINFO;
```

Members

`ipi6_addr`

The destination IPv6 address from the IP header of the received packet when used with the LPFN_WSARECVMSG (WSARecvMsg) function. The local source IPv6 address to set in the IP header when used with the WSASendMsg function.

`ipi6_ifindex`

The interface on which the packet was received when used with the LPFN_WSARECVMSG (WSARecvMsg) function. The interface on which the packet should be sent when used with the WSASendMsg function.

Remarks

If the IPV6_PKTINFO socket option is set on a socket of type **SOCK_DGRAM** or **SOCK_RAW**, one of the control data objects returned by the LPFN_WSARECVMSG (WSARecvMsg) function will contain an **in6_pktinfo** structure used to store received packet address information.

On an IPv6 socket of type **SOCK_DGRAM** or **SOCK_RAW**, an application can specific the local IP source address to use for sending with the WSASendMsg function. One of the control data objects passed in the WSAMSG structure to the **WSASendMsg** function may contain an **in6_pktinfo** structure used to specify the local IPv6 address to use for sending.

On the Microsoft Windows Software Development Kit (SDK) released for Windows Vista and later, the organization of header files has changed and the **in6_pktinfo** structure is defined in the *Ws2ipdef.h* header file which is automatically included in the *Ws2tcpip.h* header file. The *Ws2ipdef.h* header files should never be used directly.

Requirements

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 for IPv6 Winsock Applications

IPPROTO_IPV6 Socket Options

IPV6_PKTINFO

IP_PKTINFO

WSAMSG

LPFN_WSARECVMSG (WSARecvMsg)

WSASendMsg

in_pktinfo

INTERFACE_INFO structure (ws2ipdef.h)

10/22/2022 • 2 minutes to read • Edit Online

The **INTERFACE_INFO** structure is used in conjunction with the **SIO_GET_INTERFACE_LIST** ioctl command to obtain information about an interface IP address.

Syntax

```
typedef struct _INTERFACE_INFO {
  ULONG        iiFlags;
  sockaddr_gen iiAddress;
  sockaddr_gen iiBroadcastAddress;
  sockaddr_gen iiNetmask;
} INTERFACE_INFO, *LPINTERFACE_INFO;
```

Members

`iiFlags`

A bitmask describing the status of the interface. The following flags are possible.

FLAG	MEANING
IFF_UP	The interface is running.
IFF_BROADCAST	The broadcast feature is supported.
IFF_LOOPBACK	The loopback interface is running.
IFF_POINTTOPOINT	The interface is using point-to-point link.
IFF_MULTICAST	The multicast feature is supported.

`iiAddress`

Address of an interface.

`iiBroadcastAddress`

Broadcast address of the interface or the address of the other side for point-to-point links.

`iiNetmask`

Netmask used by the interface.

Remarks

On the Microsoft Windows Software Development Kit (SDK) released for Windows Vista and later, the organization of header files has changed and the **INTERFACE_INFO** structure is defined in the *Ws2ipdef.h* header file which is automatically included in the *Ws2tcpip.h* header file. The *Ws2ipdef.h* header files should never be used directly.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	ws2ipdef.h (include Ws2tcpip.h)

See also

Winsock IOCTLs

INTERFACE_INFO_EX structure (ws2ipdef.h)

10/22/2022 • 2 minutes to read • Edit Online

The **INTERFACE_INFO_EX** structure is used in conjunction with the **SIO_GET_INTERFACE_LIST IOCTL** command to obtain information about an interface IP address. Unlike the INTERFACE_INFO structure, **INTERFACE_INFO_EX** is address-size independent, enabling it to work with IPv6.

Syntax

```
typedef struct _INTERFACE_INFO_EX {
  ULONG          iiFlags;
  SOCKET_ADDRESS iiAddress;
  SOCKET_ADDRESS iiBroadcastAddress;
  SOCKET_ADDRESS iiNetmask;
} INTERFACE_INFO_EX, *LPINTERFACE_INFO_EX;
```

Members

`iiFlags`

Bitmask describing the status of the interface. The following flags are possible.

FLAG	MEANING
IFF_UP	The interface is running.
IFF_BROADCAST	The broadcast feature is supported.
IFF_LOOPBACK	The loopback interface is running.
IFF_POINTTOPOINT	The interface is using point-to-point link.
IFF_MULTICAST	The multicast feature is supported.

`iiAddress`

Address of an interface.

`iiBroadcastAddress`

Broadcast address of the interface or the address of the other side for point-to-point links.

`iiNetmask`

Netmask used by the interface.

Remarks

On the Microsoft Windows Software Development Kit (SDK) released for Windows Vista and later, the organization of header files has changed and the **INTERFACE_INFO_EX** structure is defined in the *Ws2ipdef.h* header file which is automatically included in the *Ws2tcpip.h* header file. The *Ws2ipdef.h* header files should never be used directly.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	ws2ipdef.h (include Ws2tcpip.h)

See also

Winsock IOCTLs

IP_MREQ structure (ws2ipdef.h)

10/22/2022 • 3 minutes to read • Edit Online

The **ip_mreq** structure provides multicast group information for IPv4 addresses.

Syntax

```
typedef struct ip_mreq {
  IN_ADDR imr_multiaddr;
  IN_ADDR imr_interface;
} IP_MREQ, *PIP_MREQ;
```

Members

`imr_multiaddr`

The address of the IPv4 multicast group.

`imr_interface`

The local IPv4 address of the interface or the interface index on which the multicast group should be joined or dropped. This value is in network byte order. If this member specifies an IPv4 address of 0.0.0.0, the default IPv4 multicast interface is used.

To use an interface index of 1 would be the same as an IP address of 0.0.0.1.

Remarks

The **ip_mreq** structure is used with IPv4 addresses. The **ip_mreq** structure is used with the IP_ADD_MEMBERSHIP and **IP_DROP_MEMBERSHIP** socket options.

The **ip_mreq** structure and related structures used for IPv4 multicast programming are based on IETF recommendations in sections 4 and 8.1 of RFC 3768. For more information, see http://www.ietf.org/rfc/rfc3678.txt.

For more configurable multicast capabilities with IPv4, use the ip_mreq_source structure. See Multicast Programming for more information.

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. These IP agnostic options use the GROUP_REQ and the GROUP_SOURCE_REQ structures and are the preferred socket options for multicast programming on Windows Vista and later.

The **ip_mreq** structure is the IPv4 equivalent of the IPv6-based ipv6_mreq structure.

The **imr_interface** member can be an interface index. Any IP address in the 0.x.x.x block (first octet of 0) except for the IP address of 0.0.0.0 is treated as an interface index. An interface index is a 24-bit number. The 0.0.0.0/8 IPv4 address block is not used (this range is reserved). The GetAdaptersAddresses function can be used to obtain interface index information to use for the **imr_interface** member.

It is recommended that a local IPv4 address or interface index always be specified in the **imr_interface** member of the **ip_mreq** structure, rather than use the default interface. This is particularly important on computers with multiple network interfaces and multiple public IPv4 addresses.

The default interface used for IPv4 multicast is determined by the networking stack in Windows. An application can determine the default interface used for IPv4 multicast using the GetIpForwardTable function to retrieve the IPv4 routing table. The network interface with the lowest value for the routing metric for a destination IP address of 224.0.0.0 is the default interface for IPv4 multicast. The routing table can also be displayed from the command prompt with the following command:

```
route print
```

The IP_MULTICAST_IF socket option can be used to set the default interface to send IPv4 multicast packets. This socket option does not change the default interface used to receive IPv4 multicast packets.

A typical IPv4 multicast application would use the IP_ADD_MEMBERSHIP socket option with the **ip_mreq** structure to join a multicast group and listen for multicast packets on a specific interface. The **IP_MULTICAST_IF** socket option would be used to set the interface to send IPv4 multicast packets to the multicast group. The most common scenario would be a multicast application that listens and sends on the same interface for a multicast group. Multiple sockets might be used by a multicast application with one socket for listening and one or more sockets for sending.

On the Microsoft Windows Software Development Kit (SDK) released for Windows Vista and later, the organization of header files has changed and the **ip_mreq** structure is defined in the *Ws2ipdef.h* header file which is automatically included in the *Ws2tcpip.h* header file. The *Ws2ipdef.h* header files should never be used directly.

> **Note** The **IP_MREQ** and **PIP_MREQ** derived structures are only defined on the Windows SDK released with Windows Vista and later. The **ip_mreq** structure should be used on earlier versions of the Windows SDK.

Requirements

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

GROUP_REQ

GROUP_SOURCE_REQ

GetAdaptersAddresses

GetIpForwardTable

IPPROTO_IP Socket Options

Multicast Programming

Socket Options

ip_mreq_source

ip_msfilter

ipv6_mreq

IP_MREQ_SOURCE structure (ws2ipdef.h)

10/22/2022 • 3 minutes to read • Edit Online

The **ip_mreq_source** structure provides multicast group information for IPv4 addresses.

Syntax

```
typedef struct ip_mreq_source {
  IN_ADDR imr_multiaddr;
  IN_ADDR imr_sourceaddr;
  IN_ADDR imr_interface;
} IP_MREQ_SOURCE, *PIP_MREQ_SOURCE;
```

Members

`imr_multiaddr`

The address of the IPv4 multicast group.

`imr_sourceaddr`

The address of the IPv4 multicast source.

`imr_interface`

The local IPv4 address of the interface or the interface index on which the multicast group should be joined, dropped, blocked, or unblocked. This value is in network byte order. If this member specifies an IPv4 address of 0.0.0.0, the default IPv4 multicast interface is used.

To use an interface index of 1 would be the same as an IP address of 0.0.0.1.

Remarks

The **ip_mreq_source** structure is used with IPv4 addresses. The **ip_mreq_source** structure is used with the IP_ADD_SOURCE_MEMBERSHIP, **IP_BLOCK_SOURCE**, **IP_DROP_SOURCE_MEMBERSHIP**, and **IP_UNBLOCK_SOURCE** socket options.

The **ip_mreq_source** structure and related structures used for IPv4 multicast programming are based on IETF recommendations in sections 4 and 8.1 of RFC 3768. For more information, see http://www.ietf.org/rfc/rfc3678.txt.

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. These IP agnostic options use the GROUP_REQ and the GROUP_SOURCE_REQ structures and are the preferred socket options for multicast programming on Windows Vista and later. See Multicast Programming for more information.

For less configurable multicast capabilities with IPv4, use the ip_mreq structure. For IPv6, use the ipv6_mreq structure.

The **imr_interface** member can be an interface index. Any IP address in the 0.x.x.x block (first octet of 0) except for the IP address of 0.0.0.0 is treated as an interface index. An interface index is a 24-bit number. The 0.0.0.0/8 IPv4 address block is not used (this range is reserved). The GetAdaptersAddresses function can be used to

obtain interface index information to use for the **imr_interface** member.

It is recommended that a local IPv4 address or interface index always be specified in the **imr_interface** member of the **ip_mreq_source** structure, rather than use the default interface. This is particularly important on computers with multiple network interfaces and multiple public IPv4 addresses.

The default interface used for IPv4 multicast is determined by the networking stack in Windows. An application can determine the default interface used for IPv4 multicast using the GetIpForwardTable function to retrieve the IPv4 routing table. The network interface with the lowest value for the routing metric for a destination IP address of 224.0.0.0 is the default interface for IPv4 multicast. The routing table can also be displayed from the command prompt with the following command:

route print

The IP_MULTICAST_IF socket option can be used to set the default interface to send IPv4 multicast packets. This socket option does not change the default interface used to receive IPv4 multicast packets.

A typical IPv4 multicast application would use the IP_ADD_SOURCE_MEMBERSHIP socket option with the **ip_mreq_source** structure to join a multicast group and listen for multicast packets on a specific interface. The **IP_MULTICAST_IF** socket option would be used to set the interface to send IPv4 multicast packets to the multicast group. The most common scenario would be a multicast application that listens and sends on the same interface for a multicast group. Multiple sockets might be used by a multicast application with one socket for listening and one or more sockets for sending.

On the Microsoft Windows Software Development Kit (SDK) released for Windows Vista and later, the organization of header files has changed and the **ip_mreq_source** structure is defined in the *Ws2ipdef.h* header file which is automatically included in the *Ws2tcpip.h* header file. The *Ws2ipdef.h* header files should never be used directly.

> **Note** The **IP_MREQ_SOURCE** and **PIP_MREQ_SOURCE** derived structures are only defined on the Windows SDK released with Windows Vista and later. The **ip_mreq_source** structure should be used on earlier versions of the Windows SDK.

Requirements

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

GROUP_REQ

GROUP_SOURCE_REQ

GetAdaptersAddresses

GetIpForwardTable

IPPROTO_IP Socket Options

Multicast Programming

Socket Options

ip_mreq

ip_msfilter

ipv6_mreq

IP_MSFILTER structure (ws2ipdef.h)

10/22/2022 • 3 minutes to read • Edit Online

The **ip_msfilter** structure provides multicast filtering parameters for IPv4 addresses.

Syntax

```
typedef struct ip_msfilter {
  IN_ADDR             imsf_multiaddr;
  IN_ADDR             imsf_interface;
  MULTICAST_MODE_TYPE imsf_fmode;
  ULONG               imsf_numsrc;
  IN_ADDR             imsf_slist[1];
} IP_MSFILTER, *PIP_MSFILTER;
```

Members

`imsf_multiaddr`

The IPv4 address of the multicast group.

`imsf_interface`

The local IPv4 address of the interface or the interface index on which the multicast group should be filtered. This value is in network byte order. If this member specifies an IPv4 address of 0.0.0.0, the default IPv4 multicast interface is used.

To use an interface index of 1 would be the same as an IP address of 0.0.0.1.

`imsf_fmode`

The multicast filter mode to be used. This parameter can be either MCAST_INCLUDE (value of 0) to include particular multicast sources, or MCAST_EXCLUDE (value of 1) to exclude traffic from specified sources.

On Windows Server 2003 and Windows XP, these values are defined in the *Ws2tcpip.h* header file.

On Windows Vista and later, these values are defined as enumeration values in the MULTICAST_MODE_TYPE enumeration defined in the *Ws2ipdef.h* header file.

`imsf_numsrc`

The number of sources in the **imsf_slist** member.

`imsf_slist[1]`

An array of in_addr structures that specify the IPv4 multicast source addresses to include or exclude.

Remarks

The **ip_msfilter** structure is used with IPv4 addresses. The **ip_msfilter** structure is passed as an argument for the **SIO_GET_MULTICAST_FILTER** and **SIO_SET_MULTICAST_FILTER** IOCTLs.

The **ip_msfilter** structure and related structures used for IPv4 multicast programming are based on IETF recommendations in sections 4 and 8.1 of RFC 3768. For more information, see http://www.ietf.org/rfc/rfc3678.txt.

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. These IP agnostic options use the GROUP_REQ and the GROUP_SOURCE_REQ structures and the **SIOCSMSFILTER** and **SIOCGMSFILTER** IOCTLs. These are the preferred socket options and IOCTLs for multicast programming on Windows Vista and later.

The **imsf_interface** member can be an interface index. Any IPv4 address in the 0.x.x.x block (first octet of 0) except for the IPv4 address of 0.0.0.0 is treated as an interface index. An interface index is a 24-bit number. The 0.0.0.0/8 IPv4 address block is not used (this range is reserved). The GetAdaptersAddresses function can be used to obtain interface index information to use for the **imsf_interface** member.

It is recommended that a local IPv4 address or interface index always be specified in the **imsf_interface** member of the **ip_msfilter** structure, rather than use the default interface. This is particularly important on computers with multiple network interfaces and multiple public IPv4 addresses.

The default interface used for IPv4 multicast is determined by the networking stack in Windows. An application can determine the default interface used for IPv4 multicast using the GetIpForwardTable function to retrieve the IPv4 routing table. The network interface with the lowest value for the routing metric for a destination IP address of 224.0.0.0 is the default interface for IPv4 multicast. The routing table can also be displayed from the command prompt with the following command:

route print

The IP_MULTICAST_IF socket option can be used to set the default interface to send IPv4 multicast packets. This socket option does not change the default interface used to receive IPv4 multicast packets.

A typical IPv4 multicast application would use the IP_ADD_MEMBERSHIP socket option with the ip_mreq structure or the **IP_ADD_SOURCE_MEMBERSHIP** socket option with the ip_mreq_source structure to join a multicast group and listen for multicast packets on a specific interface. The **IP_MULTICAST_IF** socket option would be used to set the interface to send IPv4 multicast packets to the multicast group. The most common scenario would be a multicast application that listens and sends on the same interface for a multicast group. Multiple sockets might be used by a multicast application with one socket for listening and one or more sockets for sending.

On the Microsoft Windows Software Development Kit (SDK) released for Windows Vista and later, the organization of header files has changed and the **ip_msfilter** structure is defined in the *Ws2ipdef.h* header file which is automatically included in the *Ws2tcpip.h* header file. The *Ws2ipdef.h* header files should never be used directly.

> **Note** The **IP_MSFILTER** and **PIP_MSFILTER** derived structures are only defined on the Windows SDK released with Windows Vista and later. The **ip_msfilter** structure should be used on earlier versions of the Windows SDK.

Requirements

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

Final-State-Based Multicast Programming

GROUP_FILTER

GROUP_REQ

GROUP_SOURCE_REQ

GetAdaptersAddresses

GetIpForwardTable

IPPROTO_IP Socket Options

MULTICAST_MODE_TYPE

Multicast Programming

Socket Options

ip_mreq

ip_mreq_source

IPV6_MREQ structure (ws2ipdef.h)

10/22/2022 • 3 minutes to read • Edit Online

The **ipv6_mreq** structure provides multicast group information for IPv6 addresses.

Syntax

```
typedef struct ipv6_mreq {
  IN6_ADDR ipv6mr_multiaddr;
  ULONG    ipv6mr_interface;
} IPV6_MREQ, *PIPV6_MREQ;
```

Members

`ipv6mr_multiaddr`

The address of the IPv6 multicast group.

`ipv6mr_interface`

The interface index of the local interface on which the multicast group should be joined or dropped. If this member specifies an interface index of 0, the default multicast interface is used.

Remarks

The **ipv6_mreq** structure is used with IPv6 addresses. The **ipv6_mreq** structure is used with the IPV6_ADD_MEMBERSHIP, **IPV6_DROP_MEMBERSHIP**, **IPV6_JOIN_GROUP**, and **IPV6_LEAVE_GROUP** socket options. The **IPV6_JOIN_GROUP** and **IPV6_ADD_MEMBERSHIP** socket options are defined to be the same. The **IPV6_LEAVE_GROUP** and **IPV6_DROP_MEMBERSHIP** socket options are defined to be the same.

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. These IP agnostic options use the GROUP_REQ and the GROUP_SOURCE_REQ structures and are the preferred socket options for multicast programming on Windows Vista and later.

The **ipv6_mreq** structure is the IPv6 equivalent of the IPv4-based ip_mreq structure.

The GetAdaptersAddresses function can be used to obtain interface index information required for the *ipv6mr_interface* member.

The **ipv6_mreq** structure and the **IPPROTO_IPV6** level socket options that use this structure are only valid on datagram and raw sockets (the socket type must be **SOCK_DGRAM** or **SOCK_RAW**).

It is recommended that a local IPv6 interface index always be specified in the **ipv6mr_interface** member of the **ipv6_mreq** structure, rather than use the default interface. This is particularly important on computers with multiple network interfaces and multiple public IPv6 addresses.

The default interface used for IPv6 multicast is determined by the networking stack in Windows. On Windows Vista and later, an application can determine the default interface used for IPv6 multicast using the GetIpForwardTable2 function to retrieve the IPv6 routing table. The network interface with the lowest value for the routing metric for a destination IPv6 multicast address (the FF00::/8 IPv6 address block) is the default interface for IPv6 multicast. The routing table can also be displayed from the command prompt with the

following command:

route print

The IPV6_MULTICAST_IF socket option can be used to set the default interface to send IPv6 multicast packets. This socket option does not change the default interface used to receive IPv6 multicast packets.

A typical IPv6 multicast application would use the IPV6_ADD_MEMBERSHIP or **IPV6_JOIN_GROUP** socket option with the **ipv6_mreq** structure to join a multicast group and listen for multicast packets on a specific interface. The **IPV6_MULTICAST_IF** socket option would be used to set the interface to send IPv6 multicast packets to the multicast group. The most common scenario would be a multicast application that listens and sends on the same interface for a multicast group. Multiple sockets might be used by a multicast application with one socket for listening and one or more sockets for sending.

On the Microsoft Windows Software Development Kit (SDK) released for Windows Vista and later, the organization of header files has changed and the **ipv6_mreq** structure is defined in the *Ws2ipdef.h* header file which is automatically included in the *Ws2tcpip.h* header file. The *Ws2ipdef.h* header files should never be used directly.

> **Note** The **PIP6_MREQ** derived structure is only defined on the Windows SDK released with Windows Vista and later. The GROUP_REQ and the GROUP_SOURCE_REQ structures and are the preferred socket options for multicast programming on Windows Vista and later.

Requirements

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

GROUP_REQ

GROUP_SOURCE_REQ

GetAdaptersAddresses

GetIpForwardTable2

IPPROTO_IPV6 Socket Options

Multicast Programming

Socket Options

ip_mreq

MULTICAST_MODE_TYPE enumeration (ws2ipdef.h)

10/22/2022 • 2 minutes to read • Edit Online

The **MULTICAST_MODE_TYPE** enumeration specifies the filter mode for multicast group addresses.

Syntax

```
typedef enum  {
  MCAST_INCLUDE = 0,
  MCAST_EXCLUDE
} MULTICAST_MODE_TYPE;
```

Constants

`MCAST_INCLUDE` Value: *0* The filter contains a list of IP addresses to include.
`MCAST_EXCLUDE` The filter contains a list of IP addresses to exclude.

Remarks

This enumeration is supported on Windows Vista and later.

The **MULTICAST_MODE_TYPE** enumeration is used in the **gf_fmode** member of the GROUP_SOURCE_REQ structure to determine if a list of IP addresses should included or excluded. The values from this enumeration can also be used in the **imsf_fmode** member of the ip_msfilter structure.

The **MULTICAST_MODE_TYPE** enumeration is defined in the *Ws2ipdef.h* header file which is automatically included in the *Ws2tcpip.h* header file. The *Ws2ipdef.h* header files should never be used directly.

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Header	ws2ipdef.h

See also

GROUP_REQ

GROUP_SOURCE_REQ

Multicast Programming

Socket Options

ip_mreq

ip_msfilter

ipv6_mreq

sockaddr_gen union (ws2ipdef.h)

10/22/2022 • 2 minutes to read • Edit Online

The **sockaddr_gen** union provides generic socket address information, and is used with the INTERFACE_INFO structure.

Syntax

```
typedef union sockaddr_gen {
  struct sockaddr         Address;
  struct sockaddr_in      AddressIn;
  struct sockaddr_in6_old AddressIn6;
} sockaddr_gen;
```

Members

`Address`

IP address information expressed in a sockaddr structure.

`AddressIn`

IP address information expressed in a sockaddr_in structure.

`AddressIn6`

IP address information expressed in a sockaddr_in6_old structure.

Remarks

On the Microsoft Windows Software Development Kit (SDK) released for Windows Vista and later, the organization of header files has changed and the **sockaddr_gen** union is defined in the *Ws2ipdef.h* header file which is automatically included in the *Ws2tcpip.h* header file. The *Ws2ipdef.h* header files should never be used directly.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	ws2ipdef.h (include Ws2tcpip.h)

See also

sockaddr

sockaddr_in

sockaddr_in6_old

sockaddr_in6_old structure (ws2ipdef.h)

10/22/2022 • 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.

Syntax

```
struct sockaddr_in6_old {
  SHORT    sin6_family;
  USHORT   sin6_port;
  ULONG    sin6_flowinfo;
  IN6_ADDR sin6_addr;
};
```

Members

`sin6_family`

`sin6_port`

`sin6_flowinfo`

`sin6_addr`

Requirements

Header	ws2ipdef.h (include Ws2ipdef.h)

See also

SOCKADDR_STORAGE

ws2spi.h header

10/22/2022 • 8 minutes to read • Edit Online

This header is used by multiple technologies. For more information, see:

- Network Drivers Prior to Windows Vista
- Windows Sockets 2

ws2spi.h contains the following programming interfaces:

Functions

NSPStartup Retrieves the dynamic information about a provider, such as the list of the DLL entry points.
WPUCloseEvent The WPUCloseEvent function closes an open event object handle.
WPUCloseSocketHandle The WPUCloseSocketHandle function closes an existing socket handle.
WPUCloseThread The WPUCloseThread function closes a thread opened with a call to WPUOpenCurrentThread.
WPUCompleteOverlappedRequest The WPUCompleteOverlappedRequest function performs overlapped I/O completion notification for overlapped I/O operations.
WPUCreateEvent The WPUCreateEvent function creates a new event object.
WPUCreateSocketHandle The WPUCreateSocketHandle function creates a new socket handle.
WPUFDIsSet The WPUFDIsSet function checks the membership of the specified socket handle.
WPUGetProviderPath The WPUGetProviderPath function retrieves the DLL path for the specified provider.

WPUModifyIFSHandle

The WPUModifyIFSHandle function receives a (possibly) modified IFS handle from Ws2_32.dll.

WPUOpenCurrentThread

The WPUOpenCurrentThread function opens a handle to the current thread that can be used with overlapped functions in a layered service provider.

WPUPostMessage

The WPUPostMessage function performs the standard Windows PostMessage function in a way that maintains backward compatibility with older versions of WSOCK32.dll.

WPUQueryBlockingCallback

The WPUQueryBlockingCallback function returns a pointer to a callback function the service provider should invoke periodically while servicing blocking operations.

WPUQuerySocketHandleContext

The WPUQuerySocketHandleContext function queries the context value associated with the specified socket handle.

WPUQueueApc

The WPUQueueApc function queues a user mode asynchronous procedure call (APC) to the specified thread in order to facilitate invocation of overlapped I/O completion routines.

WPUResetEvent

The WPUResetEvent function resets the state of the specified event object to nonsignaled. This call is safe for use within interrupt context.

WPUSetEvent

The WPUSetEvent function sets the state of the specified event object to signaled. This call is safe for use within interrupt context.

WSAAdvertiseProvider

Makes a specific namespace version-2 provider available for all eligible clients.

WSAProviderCompleteAsyncCall

Notifies a client when an asynchronous call to a namespace version-2 provider is completed.

WSAUnadvertiseProvider

Makes a specific namespace version-2 provider no longer available for clients.

WSCDeinstallProvider

Removes the specified transport provider from the system configuration database.

WSCDeinstallProvider32

Removes the specified 32-bit transport provider from the system configuration database.

WSCEnableNSProvider

Changes the state of a given namespace provider.

WSCEnableNSProvider32

Enables or disables a specified 32-bit namespace provider.

WSCEnumNameSpaceProviders32

Returns information on available 32-bit namespace providers.Note This call is a strictly 32-bit version of WSAEnumNameSpaceProviders for use on 64-bit platforms. It is provided to allow 64-bit processes to access the 32-bit catalogs. .

WSCEnumNameSpaceProvidersEx32

Retrieves information on available 32-bit namespace providers.

WSCEnumProtocols

The WSCEnumProtocols function retrieves information about available transport protocols.

WSCEnumProtocols32

Retrieves information about available transport protocols.Note This call is a strictly 32-bit version of WSCEnumProtocols for use on 64-bit platforms. It is provided to allow 64-bit processes to access the 32-bit catalogs. .

WSCGetApplicationCategory

Retrieves the layered service provider (LSP) categories associated with an application.

WSCGetProviderInfo

Retrieves the data associated with an information class for a layered service provider (LSP).

WSCGetProviderInfo32

Retrieves the data associated with an information class for a 32-bit layered service provider (LSP).Note This call is a strictly 32-bit version of WSCGetProviderInfo for use on 64-bit platforms.

WSCGetProviderPath

The WSCGetProviderPath function retrieves the DLL path for the specified provider.

WSCGetProviderPath32

Retrieves the DLL path for the specified 32-bit provider.Note This call is a strictly 32-bit version of WSCGetProviderPath for use on 64-bit platforms. It is provided to allow 64-bit processes to access the 32-bit catalogs. .

WSCInstallNameSpace

Installs a namespace provider. (WSCInstallNameSpace)

WSCInstallNameSpace32

Installs a specified 32-bit namespace provider. (WSCInstallNamespace32)

WSCInstallNameSpaceEx

Installs a namespace provider. (WSCInstallNameSpaceEx)

WSCInstallNameSpaceEx32

Installs a specified 32-bit namespace provider. (WSCInstallNameSpaceEx32)

WSCInstallProvider

Installs the specified transport provider into the system configuration database.

WSCInstallProvider64_32

Installs the specified transport service provider into the 32-bit and 64-bit system configuration databases on a 64-bit computer.

WSCInstallProviderAndChains

Installs the specified 32-bit transport provider as well as its specific protocol chains into the Winsock 2 system configuration database on a 32-bit computer.

WSCInstallProviderAndChains64_32

Installs the specified transport provider and its specific protocol chains into both the 32-bit and 64-bit Winsock 2 system configuration databases on a 64-bit computer.

WSCInstallQOSTemplate

Installs the specified QoS template in the system configuration database.

WSCRemoveQOSTemplate

Removes the specified QoS template from the system configuration database.

WSCSetApplicationCategory

Sets the permitted layered service provider (LSP) categories associated with an application.

WSCSetProviderInfo

Sets the data value for the specified information class for a layered service provider (LSP).

WSCSetProviderInfo32

Sets the data value for specified information class for a layered service provider (LSP).

WSCUnInstallNameSpace

Uninstalls the indicated name-space provider.

WSCUnInstallNameSpace32

Uninstalls a specific 32-bit namespace provider.

WSCUpdateProvider

Modifies the specified transport provider in the system configuration database.

WSCUpdateProvider32

Modifies the specified 32-bit transport provider in the system configuration database.Note This call is a strictly 32-bit version of WSCUpdateProvider for use on 64-bit platforms. It is provided to allow 64-bit processes to access the 32-bit catalogs. .

WSPStartup

The WSPStartup function initiates use of a Windows Sockets service provider interface (SPI) by a client.

Callback functions

LPNSPCLEANUP

Terminates the use of a particular Windows Sockets namespace service provider.

LPNSPGETSERVICECLASSINFO

Retrieves all the pertinent class information (schema) pertaining to the namespace provider.

LPNSPINSTALLSERVICECLASS

The NSPInstallServiceClass function registers service class schema within the namespace providers.

LPNSPIOCTL

Sends an IOCTL to a namespace service provider.

LPNSPLOOKUPSERVICEBEGIN

Initiates a client query that is constrained by the information contained within a WSAQUERYSET structure.

LPNSPLOOKUPSERVICEEND

Called to free the handle after previous calls to NSPLookupServiceBegin and NSPLookupServiceNext.

LPNSPLOOKUPSERVICENEXT

Called after obtaining a handle from a previous call to NSPLookupServiceBegin in order to retrieve the requested service information.

LPNSPREMOVESERVICECLASS

Permanently removes a specified service class from the namespace.

LPNSPSETSERVICE

Registers or deregisters a service instance within a namespace.

LPNSPV2CLEANUP

Notifies a namespace service provider version-2 (NSPv2) provider that a client session has terminated.

LPNSPV2CLIENTSESSIONRUNDOWN

Notifies a namespace service provider version-2 (NSPv2) provider that the client session is terminating.

LPNSPV2LOOKUPSERVICEBEGIN

Initiates a client query of a namespace version-2 service provider that is constrained by the information contained within a WSAQUERYSET2 structure.

LPNSPV2LOOKUPSERVICEEND

Called to free the handle after previous calls to NSPv2LookupServiceBegin and NSPv2LookupServiceNextEx.

LPNSPV2LOOKUPSERVICENEXTEX

Called after obtaining a handle from a previous call to NSPv2LookupServiceBegin in order to retrieve the requested information from a namespace version-2 service provider.

LPNSPV2SETSERVICEEX

Registers or deregisters a name or service instance within a namespace of a namespace service provider version-2 (NSPv2) provider.

LPNSPV2STARTUP

Notifies a namespace service provider version-2 (NSPv2) provider that a new client process is to begin using the provider.

LPWSPACCEPT

The LPWSPAccept function conditionally accepts a connection based on the return value of a condition function.

LPWSPADDRESSTOSTRING

The LPWSPAddressToString function converts all components of a sockaddr structure into a human-readable numeric string representation of the address. This is used mainly for display purposes.

LPWSPASYNCSELECT

The LPWSPAsyncSelect function requests Windows message-based event notification of network events for a socket.

LPWSPBIND

The LPWSPBind function associates a local address (that is, name) with a socket.

LPWSPCANCELBLOCKINGCALL

The LPWSPCancelBlockingCall function cancels a blocking call that is currently in progress.

LPWSPCLEANUP

The LPWSPCleanup function terminates use of the Windows Sockets service provider.

LPWSPCLOSESOCKET

The LPWSPCloseSocket function closes a socket.

LPWSPCONNECT

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

LPWSPDUPLICATESOCKET

The LPWSPDuplicateSocket function returns a WSAPROTOCOL_INFO structure that can be used to create a new socket descriptor for a shared socket.

LPWSPENUMNETWORKEVENTS

The LPWSPEnumNetworkEvents function reports occurrences of network events for the indicated socket.

LPWSPEVENTSELECT

The **LPWSPEventSelect** function specifies an event object to be associated with the supplied set of network events.

LPWSPGETOVERLAPPEDRESULT

The LPWSPGetOverlappedResult function returns the results of an overlapped operation on the specified socket.

LPWSPGETPEERNAME

The LPWSPGetPeerName function gets the address of the peer to which a socket is connected.

LPWSPGETQOSBYNAME

The LPWSPGetQOSByName function initializes a QOS structure based on a named template, or retrieves an enumeration of the available template names.

LPWSPGETSOCKNAME

The LPWSPGetSockName function gets the local name for a socket.

LPWSPGETSOCKOPT

The LPWSPGetSockOpt function retrieves a socket option.

LPWSPIOCTL

The LPWSPIoctl function controls the mode of a socket.

LPWSPJOINLEAF

The LPWSPJoinLeaf function joins a leaf node into a multipoint session, exchanges connect data, and specifies needed quality of service based on the supplied flow specifications.

LPWSPLISTEN The LPWSPListen function establishes a socket to listen for incoming connections.
LPWSPRECV The LPWSPRecv function receives data on a socket.
LPWSPRECVDISCONNECT The LPWSPRecvDisconnect function terminates reception on a socket and retrieves the disconnect data, if the socket is connection oriented.
LPWSPRECVFROM The LPWSPRecvFrom function receives a datagram and stores the source address.
LPWSPSELECT The LPWSPSelect function determines the status of one or more sockets.
LPWSPSEND The LPWSPSend function sends data on a connected socket.
LPWSPSENDDISCONNECT The LPWSPSendDisconnect function initiates termination of the connection for the socket and sends disconnect data.
LPWSPSENDTO The WSPSendTo function sends data to a specific destination using overlapped I/O.
LPWSPSETSOCKOPT The LPWSPSetSockOpt function sets a socket option.
LPWSPSHUTDOWN The LPWSPShutdown function disables sends and/or receives on a socket.
LPWSPSOCKET The LPWSPSocket function creates a socket.
LPWSPSTRINGTOADDRESS The WSPStringToAddress function converts a human-readable numeric string to a socket address structure (sockaddr) suitable to passing to Windows Sockets routines that take such a structure.

Structures

NSP_ROUTINE Contains information regarding the functions implemented by a namespace service provider version 1 (NSPv1) provider.
NSPV2_ROUTINE Contains information on the functions implemented by a namespace service provider version-2 (NSPv2) provider.
WSATHREADID The WSATHREADID structure enables a provider to identify a thread on which asynchronous procedure calls (APCs) can be queued using the WPUQueueApc function.
WSC_PROVIDER_AUDIT_INFO Contains audit information for a layered service provider (LSP) entry in Windows Sockets 2.
WSPDATA The WSPDATA structure contains service provider information.
WSPPROC_TABLE Contains a table of pointers to service provider functions.
WSPUPCALLTABLE Contains a table of pointers to service provider upcall functions.

Enumerations

WSC_PROVIDER_INFO_TYPE Enumeration type is used to specify the information class of a layered service protocol (LSP) in Windows Sockets 2.

LPNSPCLEANUP callback function (ws2spi.h)

10/22/2022 • 2 minutes to read • Edit Online

The **NSPCleanup** function terminates the use of a particular Windows Sockets namespace service provider.

Syntax

```
LPNSPCLEANUP Lpnspcleanup;

INT Lpnspcleanup(
  [in] LPGUID lpProviderId
)
{...}
```

Parameters

`[in] lpProviderId`

A pointer to the GUID of the namespace provider to be terminated.

Return value

If no error occurs, then **NSPCleanup** returns a value of **NO_ERROR** (zero). Otherwise, **SOCKET_ERROR** (–1) is returned, and the provider must set the appropriate error code using WSASetLastError.

ERROR CODE	MEANING
WSA_NOT_ENOUGH_MEMORY	There is not enough memory available to perform this operation.
WSAEINVAL	The *lpProviderId* parameter doesn't specify a valid provider.
WSAEOPNOTSUPP	The operation is not supported. This error is returned if the namespace provider does not implement this function.

Remarks

The **NSPCleanup** function is called when an application is finished using a Windows Sockets namespace service provider. The **NSPCleanup** function deregisters a particular namespace provider and allows the transport service provider to free any of the namespace provider's allocated resources.

The NSPStartup function must be called successfully before using any namespace providers. It is permissible to make more than one **NSPStartup** call. However, for each **NSPStartup** call, a corresponding **NSPCleanup** call must also be issued. Only the final **NSPCleanup** for the service provider does the actual cleanup; the preceding calls decrement an internal reference count in the service provider.

This function should not return until the namespace service provider DLL can be unloaded from memory.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	ws2spi.h

See also

- NSPStartup
- WSASetLastError

LPNSPGETSERVICECLASSINFO callback function (ws2spi.h)

10/22/2022 • 2 minutes to read • Edit Online

The **NSPGetServiceClassInfo** function retrieves all the pertinent class information (schema) pertaining to the namespace provider. This call retrieves any namespace-specific information that is common to all instances of the service, including connection information for SAP, or port information for SAP or TCP.

Syntax

```
LPNSPGETSERVICECLASSINFO Lpnspgetserviceclassinfo;

INT Lpnspgetserviceclassinfo(
  [in]      LPGUID lpProviderId,
  [in, out] LPDWORD lpdwBufSize,
  [in, out] LPWSASERVICECLASSINFOW lpServiceClassInfo
)
{...}
```

Parameters

`[in] lpProviderId`

A pointer to the GUID of the specific namespace provider from which the service class schema is to be retrieved.

`[in, out] lpdwBufSize`

On input, the size, in bytes, of the buffer pointed to by *lpServiceClassInfo* parameter.

On output, if the function fails and the error is WSAEFAULT, this parameter specifies the minimum size, in bytes, of the buffer pointed to the *lpServiceClassInfo* parameter needed to retrieve the record.

`[in, out] lpServiceClassInfo`

Returns a pointer to WSASERVICECLASSINFOW structure that contains the service class to namespace-specific mapping information. The *lpServiceClassId* parameter must be filled to indicate which **WSASERVICECLASSINFOW** record should be returned.

Return value

If no error occurs, the **NSPGetServiceClassInfo** function returns **NO_ERROR** (zero). Otherwise, **SOCKET_ERROR** (–1) is returned and the namespace provider must set the appropriate error code using WSASetLastError.

ERROR CODE	MEANING
WSA_NOT_ENOUGH_MEMORY	There is not enough memory available to perform this operation.
WSAEACCES	The calling routine does not have sufficient privileges to access the information.

WSAEFAULT	The buffer pointed to by the *lpServiceClass* parameter was too small to contain a WSASERVICECLASSINFOW structure. The application needs to pass in a larger buffer.
WSAEINVAL	The specified service class identifier or namespace provider identifier is not valid. This error is returned if the *lpProviderId* , *lpServiceClassId*, *lpdwBufSize*, or *lpServiceClassInfo* parameters are **NULL**.
WSAEOPNOTSUPP	The operation is not supported. This error is returned if the namespace provider does not implement this function.
WSANO_DATA	The requested name is valid, but no data of the requested type was found.
WSATYPE_NOT_FOUND	The specified class was not found.

Remarks

The W2_32.dll uses this function to implement the WSAGetServiceClassNameByClassId function, as well as to retrieve the namespace-specific information passed to the NSPLookupServiceBegin and NSPSetService functions.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	ws2spi.h

See also

NSPInstallServiceClass

NSPLookupServiceBegin

NSPRemoveServiceClass

NSPSetService

WSAGetServiceClassInfo

WSAGetServiceClassNameByClassId

WSASERVICECLASSINFOW

WSASetLastError

LPNSPINSTALLSERVICECLASS callback function (ws2spi.h)

10/22/2022 • 2 minutes to read • Edit Online

The **NSPInstallServiceClass** function registers service class schema within the namespace providers.

The schema includes the class name, class identifier, and any namespace-specific type information that is common to all instances of the service, such as SAP identifier or object identifier. A dynamic namespace provider is expected to store any class information associated with that namespace.

Syntax

```
LPNSPINSTALLSERVICECLASS Lpnspinstallserviceclass;

INT Lpnspinstallserviceclass(
  [in] LPGUID lpProviderId,
  [in] LPWSASERVICECLASSINFOW lpServiceClassInfo
)
{...}
```

Parameters

`[in] lpProviderId`

A pointer to the GUID of the specific namespace provider that this service class schema is registered in.

`[in] lpServiceClassInfo`

A pointer to the service class schema information.

Return value

The function should return **NO_ERROR** (zero) if the routine succeeds. It should return **SOCKET_ERROR** (–1) if the routine fails and it must set the appropriate error code using WSASetLastError.

ERROR CODE	MEANING
WSA_INVALID_PARAMETER	The namespace provider cannot supply the requested class information.
WSA_NOT_ENOUGH_MEMORY	There is not enough memory available to perform this operation.
WSAEACCES	The calling routine does not have sufficient privileges to perform this operation.
WSAEALREADY	The service class information has already been registered for this service class identifier. To modify service class information, first call NSPRemoveServiceClass, then reinstall with updated class information data.

WSAEINVAL	The service class identifier was invalid or improperly structured. This error is returned if the *lpServiceClassInfo* parameter is **NULL**.
WSAEOPNOTSUPP	The operation is not supported. This error is returned if the namespace provider does not implement this function.
WSANO_DATA	The requested name is valid, but no data of the requested type was found.

Remarks

Namespace providers are encouraged, but not required, to store information that is specific to the namespace they support.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	ws2spi.h

See also

NSPGetServiceClassInfo

NSPRemoveServiceClass

WSASERVICECLASSINFOW

WSASetLastError

LPNSPIOCTL callback function (ws2spi.h)

10/22/2022 • 7 minutes to read • Edit Online

The **NSPIoctl** function sends an IOCTL to a namespace service provider.

Syntax

```
LPNSPIOCTL Lpnspioctl;

INT Lpnspioctl(
  [in]      HANDLE hLookup,
  [in]      DWORD dwControlCode,
  [in]      LPVOID lpvInBuffer,
  [in, out] DWORD cbInBuffer,
  [out]     LPVOID lpvOutBuffer,
  [in]      DWORD cbOutBuffer,
  [out]     LPDWORD lpcbBytesReturned,
  [in]      LPWSACOMPLETION lpCompletion,
  [in]      LPWSATHREADID lpThreadId
)
{...}
```

Parameters

`[in] hLookup`

The lookup handle returned from a previous call to the NSPLookupServiceBegin function.

`[in] dwControlCode`

The control code of the operation to perform.

The values that may be used for the *dwControlCode* parameter are determined by the namespace provider.

The following value is supported by several Microsoft namespace providers including the Network Location Awareness (NS_NLA) namespace provider. This IOCTL is defined in the Winsock2.h header file.

VALUE	MEANING
SIO_NSP_NOTIFY_CHANGE	This operation checks if the results returned with previous calls using the *hLookup* parameter are still valid. These previous calls include the initial call to the NSPLookupServiceBegin function to retrieve the *hLookup* parameter. These previous calls may also include calls to the NSPLookupServiceNext function using the *hLookup* parameter.

`[in] lpvInBuffer`

A pointer to the input buffer.

`[in, out] cbInBuffer`

The size, in bytes, of the input buffer.

`[out] lpvOutBuffer`

A pointer to the output buffer.

`[in] cbOutBuffer`

The size, in bytes, of the output buffer.

`[out] lpcbBytesReturned`

A pointer to the number of bytes returned.

`[in] lpCompletion`

A pointer to a WSACOMPLETION structure, used for asynchronous processing. Set *lpCompletion* to **NULL** to force blocking (synchronous) execution.

`[in] 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) until after the **WPUQueueApc** function returns.

Return value

If no error occurs and the operation has completed immediately, the **NSPIoctl** function should return **NO_ERROR** (zero). Note that in this case the completion routine, if specified, will have already been queued.

The **NSPIoctl** function should return **SOCKET_ERROR** (that is, 1) if the routine fails and it must set the appropriate error code using WSASetLastError.

The error code WSA_IO_PENDING indicates that an overlapped operation has been successfully initiated and that completion will be indicated at a later time. Any other error code indicates that no overlapped operation was initiated and no completion indication will occur.

ERROR CODE	DESCRIPTION
WSA_INVALID_HANDLE	The *hLookup* parameter was not a valid query handle returned by NSPLookupServiceBegin.
WSA_IO_PENDING	An overlapped operation was successfully initiated and completion will be indicated at a later time.
WSAEFAULT	The *lpvInBuffer*, *cbInBuffer*, *lpvOutBuffer*, *cbOutBuffer*, or *lpCompletion* argument is not totally contained in a valid part of the user address space. Alternatively, the *cbInBuffer* or *cbOutBuffer* argument is too small, and the argument is modified to reflect the required allocation size.
WSAEINVAL	A supplied parameter is not acceptable, or the operation inappropriately returns results from multiple namespaces when it does not make sense for the specified operation.
WSAENETDOWN	The network subsystem has failed.
WSAEOPNOTSUPP	The operation is not supported. This error is returned if the namespace provider does not implement this function. This error can also be returned if the specified *dwControlCode* is an unrecognized command.

WSAEWOULDBLOCK	The resource is temporarily unavailable. The socket is not using overlapped I/O (asynchronous processing), yet the *lpCompletion* parameter is non-**NULL**. This error is used as a special notification for the SIO_NSP_NOTIFY_CHANGE IOCTL when the *lpCompletion* parameter is **NULL** (a poll) to indicate that a query set remains valid.

Remarks

The **NSPIoctl** function is used to send an I/O control code to a namespace provider in order to set or retrieve operating parameters associated with a query handle. The *hLookup* parameter is a handle to the namespace provider query previously returned by the NSPLookupServiceBegin function (not a socket handle).

Any IOCTL sent to a namespace provider may block indefinitely, depending upon the implementation of the namespace. If an application cannot tolerate blocking in a **NSPIoctl** function call, overlapped I/O should be used and the *lpCompletion* parameter should point to a WSACOMPLETION structure. To make a **NSPIoctl** function call nonblocking and return immediately, set the **Type** member of the **WSACOMPLETION** structure to **NSP_NOTIFY_IMMEDIATELY**.

If *lpCompletion* is **NULL**, the **NSPIoctl** function executes as a blocking call. The namespace provider should return immediately and should not block. But each namespace provider is responsible for enforcing this behavior.

The following IOCTL code is supported by several Microsoft namespace providers:

SIO_NSP_NOTIFY_CHANGE

This operation checks if the results returned with previous calls using the *hLookup* parameter are still valid. These previous calls include the initial call to the NSPLookupServiceBegin function to retrieve the *hLookup* parameter. These previous calls may also include calls to the NSPLookupServiceNext function using the *hLookup* parameter.

The Microsoft namespace providers that support this IOCTL include the following:

- NS_NLA - The Network Location Awareness (NLA) namespace provider.
- NS_PNRPNAME - The Peer Name Resolution Protocol (PNRP) namespace provider.
- NS_PNRPCLOUD - The Peer Name Resolution Protocol (PNRP) cloud namespace provider.

Other non-Microsoft namespace providers may be installed that also support this IOCTL.

When the *lpCompletion* parameter is **NULL**, this IOCTL implements a special behavior. If the *lpCompletion* parameter is **NULL** for this IOCTL, this operation is a poll and returns immediately. If the query set remains valid, WSAEWOULDBLOCK is returned as notification that the query set remains valid. If the query set has changed and is invalid, **NO_ERROR** is returned indicating success in polling for invalidation of the query set.

If the *lpCompletion* parameter is not **NULL** and points to an WSACOMPLETION structure, then the **NSPIoctl** function returns WSA_IO_PENDING if the overlapped operation was successfully initiated and completion will be indicated at a later time. The method specified in the **WSACOMPLETION** structure is used to notify the application if the query set is still valid.

Not all name resolution protocols are able to support this feature, and therefore, this function call may fail with WSAEOPNOTSUPP. A query containing data from multiple providers cannot call this IOCTL, and will return WSAEINVAL.

The *lpvInBuffer*, *cbInBuffer*, *lpvOutBuffer*, and *cbOutBuffer* parameters are currently ignored by Microsoft namespace providers.

For single-threaded applications, a typical method to use the **NSPIoctl** function is as follows. Call the **NSPIoctl** function with the *dwControlCode* parameter set to **SIO_NSP_NOTIFY_CHANGE** with no completion routine (the *lpCompletion* parameter set to **NULL**) after every NSPLookupServiceNext function call to make sure the query data is still valid. If the data becomes invalid, call the NSPLookupServiceEnd function to close the query handle. Call the NSPLookupServiceBegin function to retrieve a new query handle and begin the query again.

For multithreaded applications, a typical method to use the **NSPIoctl** function is as follows. Call the **NSPIoctl** function with the *dwControlCode* parameter set to **SIO_NSP_NOTIFY_CHANGE** with a completion routine after the initial call to the NSPLookupServiceBegin function. The application would use the mechanism for notification specified in the completion routine to be notified when data is no longer valid. One common mechanism is to use an event specified in the completion routine. If the data becomes invalid, call the NSPLookupServiceEnd function to close the query handle. Call the **NSPLookupServiceBegin** and the **NSPIoctl** functions to retrieve a new query handle and begin the query again.

Some protocols may simply cache the information locally and invalidate it after some time, in which case notification may be issued to indicate the local cache has been invalidated.

For name resolution protocols where changes are infrequent, it is possible for a namespace provider to indicate a global change event that may not be applicable to the query on which change notification was requested and issued.

Immediate poll operations are usually much less resource intensive since they do not require a notification object. In most cases, this is implemented as a simple Boolean variable check. Asynchronous notification, however, may necessitate the creation of dedicated worker threads and/or inter-process communication channels, depending on the implementation of the namespace provider service, and will incur processing overhead related to the notification object involved with signaling the change event.

To cancel an asynchronous notification request, end the original query with a NSPLookupServiceEnd function call on the affected query handle. Canceling the asynchronous notification for LUP_NOTIFY_HWND will not post any message, however, an overlapped operation will be completed and notification will be delivered with the error WSA_OPERATION_ABORTED.

> **Note** All I/O initiated by a given thread is canceled when that thread exits. For overlapped sockets, pending asynchronous operations can fail if the thread is closed before the operations complete. See ExitThread for more information.

Requirements

Minimum supported client	Windows XP [desktop apps only]
Minimum supported server	Windows Server 2003 [desktop apps only]
Target Platform	Windows
Header	ws2spi.h

See also

NSPLookupServiceBegin

NSPLookupServiceEnd

NSPLookupServiceNext

NSPStartup

NSP_ROUTINE

WPUQueueApc

WSACOMPLETION

WSATHREADID

LPNSPLOOKUPSERVICEBEGIN callback function (ws2spi.h)

10/22/2022 • 5 minutes to read • Edit Online

The **NSPLookupServiceBegin** function initiates a client query of a name service provider that is constrained by the information contained within a WSAQUERYSET structure.

NSPLookupServiceBegin only returns a handle, which should be used by subsequent calls to NSPLookupServiceNext to get the actual results. Because this operation cannot be canceled, it should be implemented to execute quickly. While it is acceptable to initiate a network query, this function should not require a response to return successfully.

Syntax

```
LPNSPLOOKUPSERVICEBEGIN Lpnsplookupservicebegin;

INT Lpnsplookupservicebegin(
  [in]  LPGUID lpProviderId,
  [in]  LPWSAQUERYSETW lpqsRestrictions,
  [in]  LPWSASERVICECLASSINFOW lpServiceClassInfo,
  [in]  DWORD dwControlFlags,
  [out] LPHANDLE lphLookup
)
{...}
```

Parameters

`[in] lpProviderId`

A pointer to the name service provider identifier to query.

`[in] lpqsRestrictions`

A pointer to the search criteria. See Remarks.

`[in] lpServiceClassInfo`

A pointer to the WSASERVICECLASSINFO structure that contains schema information for the service.

`[in] dwControlFlags`

A value that controls the depth of the search.

VALUE	MEANING
LUP_DEEP 0x0001	Queries down the hierarchy of a provider as opposed to just the first level.
LUP_CONTAINERS 0x0002	Returns containers only.

LUP_NOCONTAINERS 0x0004	Returns no containers.
LUP_NEAREST 0x0008	If possible, returns results in the order of distance. The measure of distance is provider specific.
LUP_RETURN_NAME 0x0010	Retrieves the name as **lpszServiceInstanceName**.
LUP_RETURN_TYPE 0x0020	Retrieves the type as **lpServiceClassId**.
LUP_RETURN_VERSION 0x0040	Retrieves the version as **lpVersion**.
LUP_RETURN_COMMENT 0x0080	Retrieves the comment as **lpszComment**.
LUP_RETURN_ADDR 0x0100	Retrieves the addresses as **lpcsaBuffer**.
LUP_RETURN_BLOB 0x0200	Retrieves the private data as **lpBlob**.
LUP_RETURN_ALIASES 0x0400	Any available alias information is to be returned in successive calls to NSPLookupServiceNext, and each alias returned will have the **RESULT_IS_ALIAS** flag set.
LUP_RETURN_QUERY_STRING 0x0800	Retrieves the query string as **lpszQueryString**.
LUP_RETURN_ALL 0x0ff0	Retrieves information including the name, type, version, comment, address, blob, aliases, and query string.
LUP_FLUSHCACHE 0x1000	If the provider has cached information, ignore the cache and query the namespace itself.
LUP_FLUSHPREVIOUS 0x2000	Used as a value for the *dwControlFlags* parameter in NSPLookupServiceNext. Setting this flag instructs the provider to discard the last result set, which was too large for the supplied buffer, and move on to the next result set.

LUP_NON_AUTHORITATIVE 0x4000	Indicates that the namespace provider should included non-authoritative results for names.
LUP_RES_RESERVICE 0x8000	Indicates whether prime response is in the remote or local part of CSADDR_INFO structure. The other part must be usable in either case. This option applies only to service instance requests.
LUP_SECURE 0x8000	Indicates that the namespace provider should use a secure query. This option only applies to name query requests.
LUP_RETURN_PREFERRED_NAMES 0x10000	Indicates that the namespace provider should return only preferred names.
LUP_ADDRCONFIG 0x100000	Indicates that the namespace provider should return the address configuration.
LUP_DUAL_ADDR 0x200000	Indicates that the namespace provider should return the dual addresses. This option only applies to dual-mode sockets (IPv6 and IPv4 mapped addresses).

`[out] lphLookup`

A pointer to the handle to be used in subsequent calls to NSPLookupServiceNext in order to retrieve the results set.

Return value

The function should return **NO_ERROR** (zero) if the routine succeeds. It should return **SOCKET_ERROR** (–1) if the routine fails and it must set the appropriate error code using WSASetLastError.

ERROR CODE	MEANING
WSA_NOT_ENOUGH_MEMORY	There is not enough memory available to perform this operation.
WSAEINVAL	One or more parameters were invalid, or missing, for this provider.
WSAEOPNOTSUPP	The operation is not supported. This error is returned if the namespace provider does not implement this function.
WSANO_DATA	The name was found in the database, but it does not have the correct associated data that is resolved for.
WSASERVICE_NOT_FOUND	Service is unknown. The service cannot be found in the specified namespace.

Remarks

If **LUP_CONTAINERS** is specified in a call, avoid all other restriction values. If any are supplied, the name service provider must decide if it can support this restriction over the containers. If not, it should return an error.

Some name service providers may have other means of finding containers. For example, containers can all be of some well-known type, or of a set of well-known types, and therefore a query restriction could be created for finding them. No matter what other means the name service provider has for locating containers, **LUP_CONTAINERS** and **LUP_NOCONTAINERS** take precedence. Therefore, if a query restriction is given that includes containers, specifying **LUP_NOCONTAINERS** will prevent the container items from being returned. Similarly, no matter what the query restriction, if **LUP_CONTAINERS** is given, only containers should be returned. If a namespace does not support containers and **LUP_CONTAINERS** is specified, it should return WSANO_DATA.

The preferred method of obtaining the containers within another container, is the call:

```
dwStatus = NSPLookupServiceBegin(
    lpqsRestrictions,
    LUP_CONTAINERS,
    lphLookup);
```

followed by the requisite number of NSPLookupServiceNext calls. This will return all containers contained immediately within the starting context; that is, it is not a deep query. With this, one can map the address space structure by walking the hierarchy, perhaps enumerating the content of selected containers. Subsequent uses of **NSPLookupServiceBegin** use the containers returned from a previous call.

Forming Queries

As mentioned, a WSAQUERYSET structure is used as an input parameter to **NSPLookupServiceBegin** to qualify the query. The following table lists **WSAQUERYSET** member names and describes how the **WSAQUERYSET** is used to construct a query. When a member is marked as (Optional) a null pointer can be supplied, indicating that the parameter will not be used as a search criteria. For more information, see Query-Related Data Structures.

WSAQUERYSET MEMBER NAME	QUERY INTERPRETATION
dwSize	Will be set to sizeof(WSAQUERYSET). This is a versioning mechanism.
dwOutputFlags	Ignored for queries.
lpszServiceInstanceName	Optional. Referenced string contains service name. The semantics for wildcarding within the string are not defined, but can be supported by certain namespace providers.
lpServiceClassId	Required. GUID corresponding to the service class.
lpVersion	Optional. References desired version number and provides version comparison semantics (that is, version must match exactly, or version must be not less than the value supplied).
lpszComment	Ignored for queries.
dwNameSpace	Identifier of a single namespace in which to constrain the search, or **NS_ALL** to include all namespaces.

lpNSProviderId	Optional. References the GUID of a specific namespace provider and limits the query to this provider only.
lpszContext	Optional. Specifies the starting point of the query in a hierarchical namespace.
dwNumberOfProtocols	Size, in bytes, of the number of entries in the protocol constraint array, can be zero.
lpafpProtocols	Optional. A references to an array of AFPROTOCOLS structures. Only services that use these protocols will be returned. It is permissible for the value **AF_UNSPEC** to appear as a protocol family value, signifying a wildcard. Namespace providers may supply information about any service that uses the corresponding protocol, regardless of address family.
lpszQueryString	Optional. Some namespaces (such as whois++) support rich SQL-like queries contained in a simple text string. This parameter is used to specify that string.
dwNumberOfCsAddrs	Ignored for queries.
lpcsaBuffer	Ignored for queries.
lpBlob	Optional. A pointer to a provider-specific entity.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	ws2spi.h

See also

AFPROTOCOLS

NSPLookupServiceEnd

NSPLookupServiceNext

NSP_ROUTINE

WSAQUERYSET

WSASERVICECLASSINFO

WSASetLastError

LPNSPLOOKUPSERVICEEND callback function (ws2spi.h)

10/22/2022 • 2 minutes to read • Edit Online

The **NSPLookupServiceEnd** function is called to free the handle after previous calls to NSPLookupServiceBegin and NSPLookupServiceNext.

It is possible to receive an **NSPLookupServiceEnd** call on another thread while processing an NSPLookupServiceNext. This indicates that the client has canceled the request and the provider should close the handle and return from the **NSPLookupServiceNext** call as well, setting the last error to **WSA_E_CANCELLED**.

Syntax

```
LPNSPLOOKUPSERVICEEND Lpnsplookupserviceend;

INT Lpnsplookupserviceend(
  [in] HANDLE hLookup
)
{...}
```

Parameters

`[in] hLookup`

The handle obtained previously by a call to NSPLookupServiceBegin.

Return value

The function should return **NO_ERROR** (zero) if the routine succeeds. It should return **SOCKET_ERROR** (–1) if the routine fails and it must set the appropriate error code using WSASetLastError.

ERROR CODE	MEANING
WSA_INVALID_HANDLE	The handle is not valid.
WSA_NOT_ENOUGH_MEMORY	There is not enough memory available to perform this operation.
WSAEOPNOTSUPP	The operation is not supported. This error is returned if the namespace provider does not implement this function.

Remarks

In Windows Sockets 2, conflicting error codes are defined for **WSAECANCELLED** and WSA_E_CANCELLED. The error code **WSAECANCELLED** will be removed in a future version and only WSA_E_CANCELLED will remain. Namespace Providers should use the WSA_E_CANCELLED error code to maintain compatibility with the widest

possible range of applications.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	ws2spi.h

See also

NSPLookupServiceBegin

NSPLookupServiceNext

NSP_ROUTINE

WSASetLastError

LPNSPLOOKUPSERVICENEXT callback function (ws2spi.h)

10/22/2022 • 4 minutes to read • Edit Online

The **NSPLookupServiceNext** function is called after obtaining a handle from a previous call to NSPLookupServiceBegin in order to retrieve the requested service information.

The provider will pass a WSAQUERYSET structure in the *lpqsResults* buffer. The client should call this function until it returns **WSA_E_NOMORE**, indicating that all the **WSAQUERYSET** have been returned.

Syntax

```
LPNSPLOOKUPSERVICENEXT Lpnsplookupservicenext;

INT Lpnsplookupservicenext(
  [in]      HANDLE hLookup,
  [in]      DWORD dwControlFlags,
  [in, out] LPDWORD lpdwBufferLength,
  [out]     LPWSAQUERYSETW lpqsResults
)
{...}
```

Parameters

`[in] hLookup`

A handle returned from the previous call to WSALookupServiceBegin.

`[in] dwControlFlags`

The flags used to control the next operation. Currently, only **LUP_FLUSHPREVIOUS** is defined as a means to handle a result set that is too large. If an application cannot supply a large enough buffer, setting **LUP_FLUSHPREVIOUS** instructs the provider to discard the last result set, which was too large, and move to the next set for this call.

`[in, out] lpdwBufferLength`

The size, in bytes, on input, that is contained in the buffer pointed to by *lpqsResults*. On output, if the function fails and the error is WSAEFAULT, then it contains the minimum size, in bytes to pass for the *lpqsResults* to retrieve the record.

`[out] lpqsResults`

A pointer to a memory block that will contain, on return, one result set in a WSAQUERYSET structure.

Return value

The function should return **NO_ERROR** (zero) if the routine succeeds. It should return **SOCKET_ERROR** (–1) if the routine fails and it must set the appropriate error code using WSASetLastError.

ERROR CODE	MEANING

WSA_E_CANCELLED	A call to NSPLookupServiceEnd was made while this call was still processing. The call has been canceled. The data in the *lpqsResults* buffer is undefined. In Windows Sockets 2, conflicting error codes are defined for **WSAECANCELLED** (10103) and **WSA_E_CANCELLED** (10111).The error code **WSAECANCELLED** will be removed in a future version and only WSA_E_CANCELLED will remain. Namespace providers should use the **WSA_E_CANCELLED** error code to maintain compatibility with the widest possible range of applications.
WSA_E_NO_MORE	There is no more data available. 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 use the **WSA_E_NO_MORE** error code to maintain compatibility with the widest possible range of applications.
WSA_INVALID_HANDLE	The specified lookup handle is invalid.
WSA_NOT_ENOUGH_MEMORY	There is not enough memory available to perform this operation.
WSAEFAULT	The *lpqsResults* buffer was too small to contain a WSAQUERYSET set.
WSAEINVAL	One or more parameters are invalid, or missing, for this provider.
WSAEOPNOTSUPP	The operation is not supported. This error is returned if the namespace provider does not implement this function.
WSANO_DATA	The name was found in the database, but no data, matching the given restrictions, was located.
WSASERVICE_NOT_FOUND	The service is unknown. The service cannot be found in the specified namespace.

Remarks

The *dwControlFlags* specified in this function and the ones specified at the time of NSPLookupServiceBegin are handled as "restrictions" for the purpose of combination. The restrictions are combined between the ones at **NSPLookupServiceBegin** time and the ones at **NSPLookupServiceNext** time. Therefore, the flags at **NSPLookupServiceNext** can never increase the amount of data returned beyond what was requested at **NSPLookupServiceBegin**, although it is not an error to specify more or less flags. The flags specified at a given **NSPLookupServiceNext** apply only to that call.

The *dwControlFlags* **LUP_FLUSHPREVIOUS** and **LUP_RES_SERVICE** are exceptions to the combined restrictions rule (because they are behavior flags instead of "restriction" flags). If either flag is used in **NSPLookupServiceNext**, they have their defined effect regardless of the setting of the same flags at NSPLookupServiceBegin.

For example, if **LUP_RETURN_VERSION** is specified at NSPLookupServiceBegin, the service provider retrieves records including the version. If **LUP_RETURN_VERSION** is not specified at **NSPLookupServiceNext**, the returned information does not include the version, even though it was available. No error is generated.

Also for example, if **LUP_RETURN_BLOB** is not specified at NSPLookupServiceBegin, but is specified at **NSPLookupServiceNext**, the returned information does not include the private data. No error is generated.

Query Results

The following table lists WSAQUERYSET and describes how query results are represented in the WSAQUERYSET structure. For more information, see Query-Related Data Structures.

WSAQUERYSET MEMBER NAME	RESULT INTERPRETATION
dwSize	Will be set to sizeof(WSAQUERYSET). This is used as a versioning mechanism.
dwOutputFlags	**RESULT_IS_ALIAS** flag indicates this is an alias result.
lpszServiceInstanceName	References the string that contains the service name.
lpServiceClassId	GUID corresponding to the service class.
lpVersion	References version number of the particular service instance.
lpszComment	Optional. Comment string supplied by service instance.
dwNameSpace	Namespace in which the service instance was found.
lpNSProviderId	Identifies the specific namespace provider that supplied this query result.
lpszContext	Specifies the context point in a hierarchical namespace at which the service is located.
dwNumberOfProtocols	Undefined for results.
lpafpProtocols	Undefined for results, all needed protocol information is in the CSADDR_INFO structures.
lpszQueryString	When *dwControlFlags* includes **LUP_RETURN_QUERY_STRING**, this member returns the unparsed remainder of the **lpszServiceInstanceName** specified in the original query. For example, in a namespace that identifies services by hierarchical names that specify a host name and a file path within that host, the address returned might be the host address and the unparsed remainder might be the file path. If the **lpszServiceInstanceName** is fully parsed and **LUP_RETURN_QUERY_STRING** is used, this member is null or points to a zero-length string.

dwNumberOfCsAddrs	Indicates the number of elements in the array of CSADDR_INFO structures.
lpcsaBuffer	A pointer to an array of CSADDR_INFO structures, with one complete transport address contained within each element.
lpBlob	Optional. A pointer to a provider-specific entity.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	ws2spi.h

See also

CSADDR_INFO

NSPLookupServiceBegin

NSPLookupServiceEnd

NSP_ROUTINE

WSAQUERYSET

WSASetLastError

LPNSPREMOVESERVICECLASS callback function (ws2spi.h)

10/22/2022 • 2 minutes to read • Edit Online

The **NSPRemoveServiceClass** function permanently removes a specified service class from the namespace.

Syntax

```
LPNSPREMOVESERVICECLASS Lpnspremoveserviceclass;

INT Lpnspremoveserviceclass(
  [in] LPGUID lpProviderId,
  [in] LPGUID lpServiceClassId
)
{...}
```

Parameters

`[in] lpProviderId`

A pointer to the GUID of the specific namespace provider that this service class schema is to be removed from.

`[in] lpServiceClassId`

A pointer to the GUID for the service class to remove.

Return value

The function should return **NO_ERROR** (zero) if the routine succeeds. It should return **SOCKET_ERROR** (–1) if the routine fails and it must set the appropriate error code using WSASetLastError.

ERROR CODE	MEANING
WSA_NOT_ENOUGH_MEMORY	There is not enough memory available to perform this operation.
WSA_INVALID_PARAMETER	The specified GUID was not valid.
WSAEACCES	The calling routine does not have sufficient privileges to remove the Service.
WSAEINVAL	The specified service class identifier GUID was not valid.
WSAEOPNOTSUPP	The operation is not supported. This error is returned if the namespace provider does not implement this function.

WSATYPE_NOT_FOUND	The specified class was not found in any of the namespaces.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	ws2spi.h

See also

WSASetLastError

LPNSPSETSERVICE callback function (ws2spi.h)

10/22/2022 • 4 minutes to read • Edit Online

The **NSPSetService** function registers or deregisters a service instance within a namespace.

Syntax

```
LPNSPSETSERVICE Lpnspsetservice;

INT Lpnspsetservice(
  [in] LPGUID lpProviderId,
  [in] LPWSASERVICECLASSINFOW lpServiceClassInfo,
  [in] LPWSAQUERYSETW lpqsRegInfo,
  [in] WSAESETSERVICEOP essOperation,
  [in] DWORD dwControlFlags
)
{...}
```

Parameters

`[in] lpProviderId`

A pointer to the GUID of the specific namespace provider in which the service is registered.

`[in] lpServiceClassInfo`

The service class schema information.

`[in] lpqsRegInfo`

The property information to be updated upon registration.

`[in] essOperation`

The type of operation requested.

This parameter can be one of the values from the **WSAESETSERVICEOP** enumeration type defined in the *Winsock2.h* header file.

VALUE	MEANING
RNRSERVICE_REGISTER 0	Register the service. For the Service Advertising Protocol (SAP) namespace used within a NetWare environment, this means sending a periodic broadcast. This is an NOP for the Domain Name System (DNS) namespace. For persistent data stores this means updating the address information.
RNRSERVICE_DEREGISTER 1	Deregister the service. For the SAP namespace, this means stop sending the periodic broadcast. This is a NOP for the DNS namespace. For persistent data stores this means deleting address information.

RNRSERVICE_DELETE 2	Delete the service from dynamic name and persistent spaces. For services represented by multiple CSADDR_INFO structures (using the SERVICE_MULTIPLE flag), only the supplied address will be deleted, and this must match exactly the corresponding **CSADDR_INFO** structure supplied when the service was registered.

`[in] dwControlFlags`

A set of flags that controls the service operation requested.

The possible values for this parameter are defined in the *Winsock2.h* header file.

VALUE	MEANING
SERVICE_MULTIPLE 0x00000001	Control the scope of the operation. When this value is set, the action is only performed on the given address set. A register operation does not invalidate existing addresses and a deregister operation only invalidates the given set of addresses. When this value is absent, service addresses are managed as a group. A register or deregister invalidates all existing addresses before adding the given address set.

Return value

The function should return **NO_ERROR** (zero) if the routine succeeds. It should return **SOCKET_ERROR** (–1) if the routine fails and it must set the appropriate error code using WSASetLastError.

ERROR CODE	MEANING
WSAEACCES	The calling routine does not have sufficient privileges to install the service.
WSA_NOT_ENOUGH_MEMORY	There is not enough memory available to perform this operation.
WSAEINVAL	One or more parameters were invalid, or missing, for this provider.
WSAEOPNOTSUPP	The operation is not supported. This error is returned if the namespace provider does not implement this function.
WSASERVICE_NOT_FOUND	Service is unknown. The service cannot be found in the specified namespace.

Remarks

The following table lists the available values for *essOperation* and *dwControlFlags*.

OPERATION	FLAGS	SERVICE ALREADY EXISTS	SERVICE DOES NOT EXIST

RNRSERVICE_REGISTER	None	Overwrites the object. Uses only addresses specified. Object is REGISTERED.	Creates a new object. Uses only addresses specified. Object is REGISTERED.
RNRSERVICE_REGISTER	**SERVICE_MULTIPLE**	Updates object. Adds new addresses to existing set. Object is REGISTERED.	Creates a new object. Uses all addresses specified. Object is REGISTERED.
RNRSERVICE_DEREGISTER	None	Removes all addresses, but does not remove object from namespace. Object is DEREGISTERED.	WSASERVICE_NOT_FOUND
RNRSERVICE_DEREGISTER	**SERVICE_MULTIPLE**	Updates object. Removes only addresses that are specified. Only mark object as DEREGISTERED if no addresses are present. Does not remove from the namespace.	WSASERVICE_NOT_FOUND
RNRSERVICE_DELETE	None	Removes object from the namespace.	WSASERVICE_NOT_FOUND
RNRSERVICE_DELETE	**SERVICE_MULTIPLE**	Removes only addresses that are specified. Only removes object from the namespace if no addresses remain.	WSASERVICE_NOT_FOUND

When the *dwControlFlags* parameter is set to **SERVICE_MULTIPLE**, this enables an application to manage its addresses independently. This is useful when the application must manage its protocols individually or when the service resides on more than one computer. For example, when a service uses more than one protocol, one listening socket may abort, but the other sockets remain operational. In this example, the service could deregister the aborted address without affecting the other addresses.

When using **SERVICE_MULTIPLE**, an application must not let old addresses remain in the object. This can happen if the application aborts without issuing a **RNRSERVICE_DEREGISTER** request. When a service registers, it should store its addresses. On its next call, the service should explicitly deregister these old addresses before registering new addresses.

Service Properties

The following table lists WSAQUERYSET member names and describes how service property data is represented. Members labeled as (Optional) can be supplied with a null pointer.

WSAQUERYSET MEMBER NAME	SERVICE PROPERTY DESCRIPTION
dwSize	Set to the sizeof(WSAQUERYSET). This is a versioning mechanism.
lpszServiceInstanceName	The referenced string contains the service instance name.
lpServiceClassId	The GUID that corresponds to this service class.
lpVersion	Optional. Supplies the service instance version number.

lpszComment	Optional. An optional comment string.
dwNameSpace	Ignored for this operation.
lpNSProviderId	Ignored for this operation. The provider identifier is contained in the *lpProviderId* parameter.
lpszContext	Optional. The starting point of the query in a hierarchical namespace.
dwNumberOfProtocols	Ignored for this operation.
lpafpProtocols	Ignored for this operation.
pszQueryString	Ignored for this operation.
dwNumberOfCsAddrs	The number of elements in the array of CSADDR_INFO structures referenced by *lpcsaBuffer*.
lpcsaBuffer	A pointer to an array of CSADDR_INFO structures that contain the address or addresses that the service is listening on.
dwOutputFlags	Ignored for this operation.
lpBlob	Optional. Pointer to a provider-specific entity.

> **Note** It is acceptable for the **iProtocol** member of the CSADDR_INFO structure to contain the manifest constant **IPROTOCOL_ANY**, indicating a wildcard value. The namespace provider should substitute an acceptable value for the given address family and socket type.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	ws2spi.h

See also

CSADDR_INFO

WSAQUERYSET

WSASetLastError

LPNSPV2CLEANUP callback function (ws2spi.h)

10/22/2022 • 2 minutes to read • Edit Online

The **NSPv2Cleanup** function notifies a namespace service provider version-2 (NSPv2) provider that a client session has terminated.

Syntax

```
LPNSPV2CLEANUP Lpnspv2cleanup;

INT Lpnspv2cleanup(
  [in] LPGUID lpProviderId,
  [in] LPVOID pvClientSessionArg
)
{...}
```

Parameters

`[in] lpProviderId`

A pointer to the GUID of the namespace provider to be notified.

`[in] pvClientSessionArg`

A pointer to the client session.

Return value

The function should return **NO_ERROR** (zero) if the routine succeeds. It should return **SOCKET_ERROR** (that is, 1) if the routine fails and it must set the appropriate error code using WSASetLastError.

ERROR CODE	MEANING
WSA_NOT_ENOUGH_MEMORY	There is not enough memory available to perform this operation.
WSAEACCES	The calling routine does not have sufficient privileges to initialize the service.
WSAEINVAL	One or more parameters were invalid, or missing, for this provider.
WSAEOPNOTSUPP	The operation is not supported. This error is returned if the namespace provider does not implement this function.
WSASERVICE_NOT_FOUND	Service is unknown. The service cannot be found in the specified namespace.

Remarks

The **NSPv2Cleanup** function is used as part of the namespace service provider version-2 (NSPv2) architecture available on Windows Vista and later.

On Windows Vista and Windows Server 2008, the **NSPv2Cleanup** function can only be used for operations on NS_EMAIL namespace providers.

The NSPv2Startup function is called each time a new client process begins using namespace provider. Providers may use the client session argument pointed to by the *ppvClientSessionArg* parameter to store information about this session. If a value was specified for the client session argument in the call to the **NSPv2Startup** function, then this same client session argument can be passed in the pvClientSessionArg parameter to the **NSPv2Cleanup** function to notify namespace provider that the client session has terminated.

The **NSPv2Cleanup** function is called when an application is finished using a Windows Sockets namespace service provider. The **NSPv2Cleanup** allows the namespace provider to free any of namespace provider resources that were allocated for the client session.

The NSPv2Startup function must be called successfully before calling the **NSPv2Cleanup** function. It is permissible to make more than one **NSPv2Startup** call. However, for each **NSPv2Startup** call, a corresponding **NSPv2Cleanup** call must also be issued. Only the final **NSPv2Cleanup** for the service provider does the actual cleanup; the preceding calls decrement an internal reference count in the service provider.

The NSPv2Startup, NSPv2ClientSessionRundown, and **NSPv2Cleanup** functions are optional, dependent on the requirements of the NSPv2 provider.

If the **NSPv2Cleanup** function isn't implemented, then calls to that function should be intercepted by a stub function that returns WSAEOPNOTSUPP. The NSPv2 function pointer to the unimplemented **NSPv2Cleanup** function in the NSPV2_ROUTINE structure should point be to the stub function.

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows
Header	ws2spi.h

See also

NSPV2_ROUTINE

NSPv2ClientSessionRundown

NSPv2LookupServiceBegin

NSPv2LookupServiceEnd

NSPv2LookupServiceNextEx

NSPv2SetServiceEx

NSPv2Startup

WSAQUERYSET2

LPNSPV2CLIENTSESSIONRUNDOWN callback function (ws2spi.h)

10/22/2022 • 2 minutes to read • Edit Online

The **NSPv2ClientSessionRundown** function notifies a namespace service provider version-2 (NSPv2) provider that the client session is terminating.

Syntax

```
LPNSPV2CLIENTSESSIONRUNDOWN Lpnspv2clientsessionrundown;

void Lpnspv2clientsessionrundown(
  [in] LPGUID lpProviderId,
  [in] LPVOID pvClientSessionArg
)
{...}
```

Parameters

`[in] lpProviderId`

A pointer to the GUID of the specific namespace provider to notify.

`[in] pvClientSessionArg`

A pointer to the client session that is terminating.

Return value

The function should return **NO_ERROR** (zero) if the routine succeeds. It should return **SOCKET_ERROR** (that is, 1) if the routine fails and it must set the appropriate error code using WSASetLastError.

ERROR CODE	MEANING
WSA_NOT_ENOUGH_MEMORY	There is not enough memory available to perform this operation.
WSAEACCES	The calling routine does not have sufficient privileges to install the service.
WSAEINVAL	One or more parameters were invalid, or missing, for this provider.
WSAEOPNOTSUPP	The operation is not supported. This error is returned if the namespace provider does not implement this function. This error can also be returned if the specified *dwControlCode* is an unrecognized command.
WSASERVICE_NOT_FOUND	Service is unknown. The service cannot be found in the specified namespace.

Remarks

The **NSPv2ClientSessionRundown** function is used as part of the namespace service provider version-2 (NSPv2) architecture available on Windows Vista and later.

On Windows Vista and Windows Server 2008, the **NSPv2ClientSessionRundown** function can only be used for operations on NS_EMAIL namespace providers.

The NSPv2Startup function is called each time a new client process begins using namespace provider. Providers may use the client session argument pointed to by the *ppvClientSessionArg* parameter to store information about this session. If a value was specified for the client session argument in the call to the **NSPv2Startup** function, then this same client session argument is passed in the *pvClientSessionArg* parameter to the **NSPv2ClientSessionRundown** function.

The NSPv2Startup, **NSPv2ClientSessionRundown**, and NSPv2Cleanup functions are optional, dependent on the requirements of the NSPv2 provider.

If the **NSPv2ClientSessionRundown** function isn't implemented, then calls to that function should be intercepted by a stub function that returns WSAEOPNOTSUPP. The NSPv2 function pointer to the unimplemented **NSPv2ClientSessionRundown** function in the NSPV2_ROUTINE structure should point be to the stub function.

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows
Header	ws2spi.h

See also

NSPV2_ROUTINE

NSPv2Cleanup

NSPv2LookupServiceBegin

NSPv2LookupServiceEnd

NSPv2LookupServiceNextEx

NSPv2SetServiceEx

NSPv2Startup

WSAQUERYSET2

WSASetLastError

LPNSPV2LOOKUPSERVICEBEGIN callback function (ws2spi.h)

10/22/2022 • 6 minutes to read • Edit Online

The NSPv2LookupServiceBegin function initiates a client query of a namespace version-2 service provider that is constrained by the information contained within a WSAQUERYSET2 structure.

Syntax

```
LPNSPV2LOOKUPSERVICEBEGIN Lpnspv2lookupservicebegin;

INT Lpnspv2lookupservicebegin(
  [in]  LPGUID lpProviderId,
  [in]  LPWSAQUERYSET2W lpqsRestrictions,
  [in]  DWORD dwControlFlags,
  [out] LPVOID lpvClientSessionArg,
  [out] LPHANDLE lphLookup
)
{...}
```

Parameters

`[in] lpProviderId`

A pointer to the identifier for the namespace service provider to query.

`[in] lpqsRestrictions`

A pointer to the search criteria. See Remarks.

`[in] dwControlFlags`

A set of flags that affect the search. This parameter can be a combination of the following values defined in the *Winsock2.h* header file.

VALUE	MEANING
LUP_DEEP 0x0001	Queries down the hierarchy of a provider as opposed to just the first level.
LUP_CONTAINERS 0x0002	Returns containers only.
LUP_NOCONTAINERS 0x0004	Returns no containers.

LUP_NEAREST 0x0008	If possible, returns results in the order of distance. The measure of distance is provider specific.
LUP_RETURN_NAME 0x0010	Retrieves the name as **lpszServiceInstanceName**.
LUP_RETURN_TYPE 0x0020	Retrieves the type as **lpServiceClassId**.
LUP_RETURN_VERSION 0x0040	Retrieves the version as **lpVersion**.
LUP_RETURN_COMMENT 0x0080	Retrieves the comment as **lpszComment**.
LUP_RETURN_ADDR 0x0100	Retrieves the addresses as **lpcsaBuffer**.
LUP_RETURN_BLOB 0x0200	Retrieves the private data as **lpBlob**.
LUP_RETURN_ALIASES 0x0400	Any available alias information is to be returned in successive calls to NSPv2LookupServiceNextEx, and each alias returned will have the **RESULT_IS_ALIAS** flag set.
LUP_RETURN_QUERY_STRING 0x0800	Retrieves the query string as **lpszQueryString**.
LUP_RETURN_ALL 0x0ff0	Retrieves information including the name, type, version, comment, address, blob, aliases, and query string.
LUP_FLUSHCACHE 0x1000	If the provider has cached information, ignore the cache and query the namespace itself.
LUP_FLUSHPREVIOUS 0x2000	Used as a value for the *dwControlFlags* parameter in NSPv2LookupServiceNextEx. Setting this flag instructs the provider to discard the last result set, which was too large for the supplied buffer, and move on to the next result set.
LUP_NON_AUTHORITATIVE 0x4000	Indicates that the namespace provider should included non-authoritative results for names.

LUP_RES_RESERVICE 0x8000	Indicates whether prime response is in the remote or local part of CSADDR_INFO structure. The other part must be usable in either case. This option applies only to service instance requests.
LUP_SECURE 0x8000	Indicates that the namespace provider should use a secure query. This option only applies to name query requests.
LUP_RETURN_PREFERRED_NAMES 0x10000	Indicates that the namespace provider should return only preferred names.
LUP_ADDRCONFIG 0x100000	Indicates that the namespace provider should return the address configuration.
LUP_DUAL_ADDR 0x200000	Indicates that the namespace provider should return the dual addresses. This option only applies to dual-mode sockets (IPv6 and IPv4 mapped addresses).
LUP_DISABLE_IDN_ENCODING 0x800000	Indicates that the namespace provider should disable automatic International Domain Names encoding. This value is supported on Windows 8 and Windows Server 2012

`[out] lpvClientSessionArg`

A pointer to the client session.

`[out] lphLookup`

A pointer to the handle to be used in subsequent calls to NSPv2LookupServiceNextEx in order to retrieve the results set.

Return value

The function should return **NO_ERROR** (zero) if the routine succeeds. It should return **SOCKET_ERROR** (that is, 1) if the routine fails and it must set the appropriate error code using WSASetLastError.

ERROR CODE	MEANING
WSAEINVAL	One or more parameters were invalid, or missing, for this provider.
WSANO_DATA	The name was found in the database, but it does not have the correct associated data that is resolved for.
WSASERVICE_NOT_FOUND	The service is unknown. The service cannot be found in the specified namespace.
WSA_NOT_ENOUGH_MEMORY	There is not enough memory available to perform this operation.

Remarks

The NSPv2LookupServiceBegin function is used as part of the namespace service provider version-2 (NSPv2) architecture available on Windows Vista and later.

On Windows Vista and Windows Server 2008, the NSPv2LookupServiceBegin function can only be used for operations on NS_EMAIL namespace providers.

The NSPv2LookupServiceBegin function only returns a handle, which should be used by subsequent calls to NSPv2LookupServiceNextEx to get the actual results. Because this operation cannot be canceled, it should be implemented to execute quickly. While it is acceptable to initiate a network query, this function should not require a response to return successfully.

The NSPv2Startup function is called each time a new client process begins using namespace provider. Providers may use the client session argument pointed to by the *ppvClientSessionArg* parameter to store information about this session. If a value was specified for the client session argument in the call to the **NSPv2Startup** function, then this same client session argument is passed in the *lpvClientSessionArg* parameter to the NSPv2LookupServiceBegin function.

If **LUP_CONTAINERS** is specified in a call, avoid all other restriction values. If any are supplied, the name service provider must decide if it can support this restriction over the containers. If not, it should return an error.

Some name service providers may have other means of finding containers. For example, containers can all be of some well-known type, or of a set of well-known types, and therefore a query restriction could be created for finding them. No matter what other means the name service provider has for locating containers, **LUP_CONTAINERS** and **LUP_NOCONTAINERS** take precedence. Therefore, if a query restriction is given that includes containers, specifying **LUP_NOCONTAINERS** will prevent the container items from being returned. Similarly, no matter what the query restriction, if **LUP_CONTAINERS** is given, only containers should be returned. If a namespace does not support containers and **LUP_CONTAINERS** is specified, it should return WSANO_DATA.

The preferred method of obtaining the containers within another container, is the call:

```
dwStatus = NSPv2LookupServiceBegin(
    lpProviderId,
    lpqsRestrictions,
    LUP_CONTAINERS,
    lpClientSession,
    lphLookup);
```

followed by the requisite number of NSPv2LookupServiceNextEx calls. This will return all containers contained immediately within the starting context; that is, it is not a deep query. With this, one can map the address space structure by walking the hierarchy, perhaps enumerating the content of selected containers. Subsequent uses of NSPv2LookupServiceBegin use the containers returned from a previous call.

Forming Queries

The WSAQUERYSET2 structure is used as an input parameter to NSPv2LookupServiceBegin to qualify the query. The following table lists **WSAQUERYSET2** member names and describes how the **WSAQUERYSET2** is used to construct a query. Members labeled as optional and dependent on the requirements of the NSPv2 provider may be supplied as a **NULL** pointer when unused as a search criteria by the namespace provider. For more information, see Query-Related Data Structures.

WSAQUERYSET2 MEMBER NAME	QUERY INTERPRETATION

dwSize	Will be set to sizeof(WSAQUERYSET2). This is a versioning mechanism.
lpszServiceInstanceName	A string that contains the service name. The semantics for wildcarding within the string are not defined, but can be supported by certain namespace providers. This member is optional, dependent on the requirements of the NSPv2 service provider.
lpVersion	The desired version number that provides version comparison semantics (that is, version must match exactly, or version must be not less than the value supplied). This member is optional, dependent on the requirements of the NSPv2 service provider.
lpszComment	This member is ignored for queries.
dwNameSpace	The identifier of a single namespace in which to constrain the search, or **NS_ALL** to include all namespaces.
lpNSProviderId	The GUID of a specific namespace provider that limits the query to this provider only. This member is optional, dependent on the requirements of the NSPv2 service provider.
lpszContext	The starting point of the query in a hierarchical namespace. This member is optional, dependent on the requirements of the NSPv2 service provider.
dwNumberOfProtocols	The size, in bytes, of the number of entries in the protocol constraint array. This member can be zero.
lpafpProtocols	An array of AFPROTOCOLS structures. Only services that use these protocols will be returned. It is permissible for the value **AF_UNSPEC** to appear as a protocol family value, signifying a wildcard. Namespace providers may supply information about any service that uses the corresponding protocol, regardless of address family. This member is optional, dependent on the requirements of the NSPv2 service provider.
lpszQueryString	Some namespaces (such as whois++) support rich SQL-like queries contained in a simple text string. This parameter is used to specify that string.This member is optional, dependent on the requirements of the NSPv2 service provider.
dwNumberOfCsAddrs	This member is ignored for queries.
lpcsaBuffer	This member is ignored for queries.
dwOutputFlags	This member is ignored for queries.
lpBlob	A pointer to a provider-specific entity. This member is optional, dependent on the requirements of the NSPv2 service provider.

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows
Header	ws2spi.h

See also

AFPROTOCOLS

NSPV2_ROUTINE

NSPv2Cleanup

NSPv2ClientSessionRundown

NSPv2LookupServiceEnd

NSPv2LookupServiceNextEx

NSPv2SetServiceEx

NSPv2Startup

WSAQUERYSET2

WSASERVICECLASSINFO

WSASetLastError

LPNSPV2LOOKUPSERVICEEND callback function (ws2spi.h)

10/22/2022 • 2 minutes to read • Edit Online

The **NSPv2LookupServiceEnd** function is called to free the handle after previous calls to NSPv2LookupServiceBegin and NSPv2LookupServiceNextEx.

Syntax

```
LPNSPV2LOOKUPSERVICEEND Lpnspv2lookupserviceend;

INT Lpnspv2lookupserviceend(
  [in] HANDLE hLookup
)
{...}
```

Parameters

`[in] hLookup`

The handle obtained previously by a call to NSPv2LookupServiceBegin.

Return value

The function should return **NO_ERROR** (zero) if the routine succeeds. It should return **SOCKET_ERROR** (that is, 1) if the routine fails and it must set the appropriate error code using WSASetLastError.

ERROR CODE	MEANING
WSA_INVALID_HANDLE	The handle is not valid.
WSA_NOT_ENOUGH_MEMORY	There is not enough memory available to perform this operation.

Remarks

The **NSPv2LookupServiceEnd** function is used as part of the namespace service provider version-2 (NSPv2) architecture available on Windows Vista and later.

On Windows Vista and Windows Server 2008, the **NSPv2LookupServiceEnd** function can only be used for operations on NS_EMAIL namespace providers.

It is possible to receive an NSPv2LookupServiceBegin function call on another thread while processing an NSPv2LookupServiceNextEx. This indicates that the client has canceled the request and the provider should close the handle and return from the **NSPv2LookupServiceNextEx** function call as well, setting the last error to **WSA_E_CANCELLED**.

In Windows Sockets 2, conflicting error codes are defined for **WSAECANCELLED** and WSA_E_CANCELLED. The

error code **WSAECANCELLED** will be removed in a future version and only WSA_E_CANCELLED will remain. Namespace Providers should use the WSA_E_CANCELLED error code to maintain compatibility with the widest possible range of applications.

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows
Header	ws2spi.h

See also

NSPV2_ROUTINE

NSPv2Cleanup

NSPv2ClientSessionRundown

NSPv2LookupServiceBegin

NSPv2LookupServiceNextEx

NSPv2SetServiceEx

NSPv2Startup

WSAQUERYSET2

WSASetLastError

LPNSPV2LOOKUPSERVICENEXTEX callback function (ws2spi.h)

10/22/2022 • 6 minutes to read • Edit Online

The **NSPv2LookupServiceNextEx** function is called after obtaining a handle from a previous call to NSPv2LookupServiceBegin in order to retrieve the requested information from a namespace version-2 service provider.

Syntax

```
LPNSPV2LOOKUPSERVICENEXTEX Lpnspv2lookupservicenextex;

void Lpnspv2lookupservicenextex(
  [in]      HANDLE hAsyncCall,
  [in]      HANDLE hLookup,
  [in]      DWORD dwControlFlags,
  [in, out] LPDWORD lpdwBufferLength,
  [out]     LPWSAQUERYSET2W lpqsResults
)
{...}
```

Parameters

`[in] hAsyncCall`

A handle returned from the previous call to NSPv2LookupServiceBegin used for asynchronous calls.

`[in] hLookup`

A handle returned from the previous call to NSPv2LookupServiceBegin.

`[in] dwControlFlags`

The flags used to control the next operation. Currently, only **LUP_FLUSHPREVIOUS** is defined as a means to handle a result set that is too large. If an application cannot supply a large enough buffer, setting **LUP_FLUSHPREVIOUS** instructs the provider to discard the last result set, which was too large, and move to the next set for this call.

`[in, out] lpdwBufferLength`

The size, in bytes, on input, that is contained in the buffer pointed to by *lpqsResults*. On output, if the function fails and the error is WSAEFAULT, then it contains the minimum size, in bytes to pass for the *lpqsResults* to retrieve the record.

`[out] lpqsResults`

A pointer to a memory block that will contain, on return, one result set in a WSAQUERYSET2 structure.

Return value

The function should return **NO_ERROR** (zero) if the routine succeeds. It should return **SOCKET_ERROR** (that is, 1) if the routine fails and it must set the appropriate error code using WSASetLastError.

ERROR CODE	MEANING
WSA_E_CANCELLED	A call to NSPv2LookupServiceEnd was made while this call was still processing. The call has been canceled. The data in the *lpqsResults* buffer is undefined. In Windows Sockets 2, conflicting error codes are defined for **WSAECANCELLED** (10103) and **WSA_E_CANCELLED** (10111).The error code **WSAECANCELLED** will be removed in a future version and only WSA_E_CANCELLED will remain. Namespace providers should use the **WSA_E_CANCELLED** error code to maintain compatibility with the widest possible range of applications.
WSA_E_NO_MORE	There is no more data available. 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 use the **WSA_E_NO_MORE** error code to maintain compatibility with the widest possible range of applications.
WSAEFAULT	The *lpqsResults* buffer was too small to contain a WSAQUERYSET set.
WSAEINVAL	One or more parameters are invalid, or missing, for this provider.
WSA_INVALID_HANDLE	The specified lookup handle is invalid.
WSANO_DATA	The name was found in the database, but no data, matching the given restrictions, was located.
WSASERVICE_NOT_FOUND	The service is unknown. The service cannot be found in the specified namespace.
WSA_NOT_ENOUGH_MEMORY	There is not enough memory available to perform this operation.

Remarks

The **NSPv2LookupServiceNextEx** function is used as part of the namespace service provider version-2 (NSPv2) architecture available on Windows Vista and later.

On Windows Vista and Windows Server 2008, the **NSPv2LookupServiceNextEx** function can only be used for operations on NS_EMAIL namespace providers.

The provider will pass a WSAQUERYSET2 structure in the *lpqsResults* buffer. The client should call the **NSPv2LookupServiceNextEx** function until it returns **WSA_E_NOMORE**, indicating that all the **WSAQUERYSET2** structures have been returned.

The *dwControlFlags* specified in this function and the ones specified at the time of NSPv2LookupServiceBegin

are handled as "restrictions" for the purpose of combination. The restrictions are combined between the ones at **NSPv2LookupServiceBegin** time and the ones at **NSPv2LookupServiceNextEx** time. Therefore, the flags at **NSPv2LookupServiceNextEx** can never increase the amount of data returned beyond what was requested at **NSPv2LookupServiceBegin**, although it is not an error to specify more or less flags. The flags specified at a given **NSPv2LookupServiceNextEx** apply only to that call.

The *dwControlFlags* **LUP_FLUSHPREVIOUS** and **LUP_RES_SERVICE** are exceptions to the combined restrictions rule (because they are behavior flags instead of "restriction" flags). If either flag is used in **NSPv2LookupServiceNextEx**, they have their defined effect regardless of the setting of the same flags at NSPv2LookupServiceBegin.

For example, if **LUP_RETURN_VERSION** is specified at NSPv2LookupServiceBegin, the service provider retrieves records including the version. If **LUP_RETURN_VERSION** is not specified at **NSPv2LookupServiceNextEx**, the returned information does not include the version, even though it was available. No error is generated.

Also for example, if **LUP_RETURN_BLOB** is not specified at NSPv2LookupServiceBegin, but is specified at **NSPv2LookupServiceNextEx**, the returned information does not include the private data. No error is generated.

The **NSPv2LookupServiceNextEx** function is typically called at least twice. The first time to get the size of the needed buffer to receive the WSAQUERYSET2 pointed to by the *lpqsResults* parameter, and the second time to get the actual query result set. On the first call, the NSPv2 provider should return the size necessary for the **WSAQUERYSET2** results.

The WSAQUERYSET2 structure pointed to by the *lpqsResults* parameter that is returned is only useful in the same process context, since several of the members in the **WSAQUERYSET2** structure contains pointers to the actual data returned. If the query result needs to be passed to another process (using RPC, for example), then it will be necessary to serialize and marshal the data returned in the **WSAQUERYSET2** structure pointed to by the *lpqsResults* parameter including the data pointed to by members in the **WSAQUERYSET2** structure. The data needs to be serialized in a form that can be passed across process boundaries. Just passing a copy of the **WSAQUERYSET2** structure is insufficient, since only pointers to data will be passed and the actual data will be unavailable to other processes.

Query Results

The following table lists WSAQUERYSET2 and describes how query results are represented in the **WSAQUERYSET2** structure. For more information, see Query-Related Data Structures.

WSAQUERYSET2 MEMBER NAME	RESULT INTERPRETATION
dwSize	The size, in bytes, of WSAQUERYSET2 structure. This is used as a versioning mechanism.
lpszServiceInstanceName	A string that contains the service name.
lpVersion	References version number of the particular service instance.
lpszComment	A comment string supplied by service instance. This member is optional, dependent on the requirements of the NSPv2 service provider.
dwNameSpace	The namespace identifier in which the name or service instance was found.
lpNSProviderId	The specific namespace provider that supplied this query result.

lpszContext	The context point in a hierarchical namespace at which the service is located.
dwNumberOfProtocols	This member is undefined for results.
lpafpProtocols	This member is undefined for results. All needed protocol information is in the CSADDR_INFO structures.
lpszQueryString	When *dwControlFlags* includes **LUP_RETURN_QUERY_STRING**, this member returns the unparsed remainder of the **lpszServiceInstanceName** specified in the original query. For example, in a namespace that identifies services by hierarchical names that specify a host name and a file path within that host, the address returned might be the host address and the unparsed remainder might be the file path. If the **lpszServiceInstanceName** is fully parsed and **LUP_RETURN_QUERY_STRING** is used, this member is null or points to a zero-length string.
dwNumberOfCsAddrs	The number of elements in the array of CSADDR_INFO structures.
lpcsaBuffer	A pointer to an array of CSADDR_INFO structures, with one complete transport address contained within each element.
dwOutputFlags	The **RESULT_IS_ALIAS** flag indicates this is an alias result.
lpBlob	A pointer to a provider-specific entity. This member is optional, dependent on the requirements of the NSPv2 service provider.

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows
Header	ws2spi.h

See also

CSADDR_INFO

NSPV2_ROUTINE

NSPv2Cleanup

NSPv2ClientSessionRundown

NSPv2LookupServiceBegin

NSPv2LookupServiceEnd

NSPv2SetServiceEx

NSPv2Startup

WSAQUERYSET2

WSASetLastError

LPNSPV2SETSERVICEEX callback function (ws2spi.h)

10/22/2022 • 6 minutes to read • Edit Online

The **NSPv2SetServiceEx** function registers or deregisters a name or service instance within a namespace of a namespace service provider version-2 (NSPv2) provider.

Syntax

```
LPNSPV2SETSERVICEEX Lpnspv2setserviceex;

void Lpnspv2setserviceex(
  [in] HANDLE hAsyncCall,
  [in] LPGUID lpProviderId,
  [in] LPWSAQUERYSET2W lpqsRegInfo,
  [in] WSAESETSERVICEOP essOperation,
  [in] DWORD dwControlFlags,
  [in] LPVOID lpvClientSessionArg
)
{...}
```

Parameters

`[in] hAsyncCall`

A handle returned from the previous call to NSPv2LookupServiceBegin used for asynchronous calls.

`[in] lpProviderId`

A pointer to the GUID of the specific namespace provider in which the name or service is registered.

`[in] lpqsRegInfo`

The property information to be updated upon registration.

`[in] essOperation`

The type of operation requested.

This parameter can be one of the values from the **WSAESETSERVICEOP** enumeration type defined in the *Winsock2.h* header file.

VALUE	MEANING
RNRSERVICE_REGISTER 0	Register the service. For the Service Advertising Protocol (SAP) namespace used within a NetWare environment, this means sending a periodic broadcast. This is an NOP for the Domain Name System (DNS) namespace. For persistent data stores this means updating the address information.
RNRSERVICE_DEREGISTER 1	Deregister the service. For the SAP namespace, this means stop sending the periodic broadcast. This is a NOP for the DNS namespace. For persistent data stores this means deleting address information.

RNRSERVICE_DELETE 2	Delete the service from dynamic name and persistent spaces. For services represented by multiple CSADDR_INFO structures (using the SERVICE_MULTIPLE flag), only the supplied address will be deleted, and this must match exactly the corresponding **CSADDR_INFO** structure supplied when the service was registered.

`[in] dwControlFlags`

A set of flags that controls the operation requested.

The possible values for this parameter are defined in the *Winsock2.h* header file.

VALUE	MEANING
SERVICE_MULTIPLE 0x00000001	Control the scope of the operation. When this value is set, the action is only performed on the given address set. A register operation does not invalidate existing addresses and a deregister operation only invalidates the given set of addresses. When this value is absent, service addresses are managed as a group. A register or deregister invalidates all existing addresses before adding the given address set.

`[in] lpvClientSessionArg`

A pointer to the client session.

Return value

The function should return **NO_ERROR** (zero) if the routine succeeds. It should return **SOCKET_ERROR** (that is, 1) if the routine fails and it must set the appropriate error code using WSASetLastError.

ERROR CODE	MEANING
WSA_NOT_ENOUGH_MEMORY	There is not enough memory available to perform this operation.
WSAEACCES	The calling routine does not have sufficient privileges to install the service.
WSAEINVAL	One or more parameters were invalid, or missing, for this provider.
WSAEOPNOTSUPP	The operation is not supported. This error is returned if the namespace provider does not implement this function. This error can also be returned if the specified *dwControlCode* is an unrecognized command.
WSASERVICE_NOT_FOUND	Service is unknown. The service cannot be found in the specified namespace.

Remarks

The **NSPv2SetServiceEx** function is used as part of the namespace service provider version-2 (NSPv2) architecture available on Windows Vista and later.

On Windows Vista and Windows Server 2008, the **NSPv2SetServiceEx** function can only be used for operations on NS_EMAIL namespace providers.

The NSPv2Startup function is called each time a new client process begins using namespace provider. Providers may use the client session argument pointed to by the *ppvClientSessionArg* parameter to store information about this session. This client session argument can be passed to the **NSPv2SetServiceEx** function in the *lpvClientSessionArg* parameter.

The **NSPv2SetServiceEx** function is optional, dependent on the requirements of the NSPv2 provider. If the **NSPv2SetServiceEx** function isn't implemented, then the NSPv2 function pointer can be to a stub function that always returns **NO_ERROR**.

The following table lists the possible combination of values for *essOperation* and *dwControlFlags* parameters.

ESSOPERATION	DWCONTROLFLAGS	SERVICE ALREADY EXISTS	SERVICE DOES NOT EXIST
RNRSERVICE_REGISTER	None	Overwrites the object. Uses only addresses specified. Object is REGISTERED.	Creates a new object. Uses only addresses specified. Object is REGISTERED.
RNRSERVICE_REGISTER	**SERVICE_MULTIPLE**	Updates object. Adds new addresses to existing set. Object is REGISTERED.	Creates a new object. Uses all addresses specified. Object is REGISTERED.
RNRSERVICE_DEREGISTER	None	Removes all addresses, but does not remove object from namespace. Object is DEREGISTERED.	WSASERVICE_NOT_FOUND
RNRSERVICE_DEREGISTER	**SERVICE_MULTIPLE**	Updates object. Removes only addresses that are specified. Only mark object as DEREGISTERED if no addresses are present. Does not remove from the namespace.	WSASERVICE_NOT_FOUND
RNRSERVICE_DELETE	None	Removes object from the namespace.	WSASERVICE_NOT_FOUND
RNRSERVICE_DELETE	**SERVICE_MULTIPLE**	Removes only addresses that are specified. Only removes object from the namespace if no addresses remain.	WSASERVICE_NOT_FOUND

When the *dwControlFlags* parameter is set to **SERVICE_MULTIPLE**, this enables an application to manage its addresses independently. This is useful when the application must manage its protocols individually or when the service resides on more than one computer. For example, when a service uses more than one protocol, one listening socket may abort, but the other sockets remain operational. In this example, the service could deregister the aborted address without affecting the other addresses.

When using **SERVICE_MULTIPLE**, an application must not let old addresses remain in the object. This can happen if the application aborts without issuing a **RNRSERVICE_DEREGISTER** request. When a service

registers, it should store its addresses. On its next call, the service should explicitly deregister these old addresses before registering new addresses.

If the **NSPv2SetServiceEx** function isn't implemented, then calls to that function should be intercepted by a stub function that returns WSAEOPNOTSUPP. The NSPv2 function pointer to the unimplemented **NSPv2SetServiceEx** function in the NSPV2_ROUTINE structure should point be to the stub function.

Service Properties

The following table lists WSAQUERYSET2 member names and describes how service property data is represented. Members labeled as optional and dependent on the requirements of the NSPv2 provider may be supplied as a **NULL** pointer when unused by the namespace provider.

WSAQUERYSET2 MEMBER NAME	SERVICE PROPERTY DESCRIPTION
dwSize	Set to the sizeof(WSAQUERYSET2). This is a versioning mechanism.
lpszServiceInstanceName	A string that contains the service instance name.
lpVersion	The service instance version number. This member is optional, dependent on the requirements of the NSPv2 service provider.
lpszComment	A comment string. This member is optional, dependent on the requirements of the NSPv2 service provider.
dwNameSpace	The namespace identifier. This member is optional, dependent on the requirements of the NSPv2 service provider.
lpNSProviderId	The provider identifier. Note that the namespace provider identifier is also passed in the *lpProviderId* parameter. This member is optional, dependent on the requirements of the NSPv2 service provider.
lpszContext	The starting point of the query in a hierarchical namespace. This member is optional, dependent on the requirements of the NSPv2 service provider.
dwNumberOfProtocols	The size, in bytes, of the number of entries in the protocol constraint array. This member can be zero.This member is optional, dependent on the requirements of the NSPv2 service provider.
lpafpProtocols	An array of AFPROTOCOLS structures. This member is optional, dependent on the requirements of the NSPv2 service provider.
lpszQueryString	Some namespaces (such as whois++) support rich SQL-like queries contained in a simple text string. This parameter is used to specify that string.This member is optional, dependent on the requirements of the NSPv2 service provider.
dwNumberOfCsAddrs	The number of elements in the array of CSADDR_INFO structures referenced by *lpcsaBuffer*.

lpcsaBuffer	A pointer to an array of CSADDR_INFO structures that contain the address or addresses that the service is listening on.
dwOutputFlags	This member is optional, dependent on the requirements of the NSPv2 service provider.
lpBlob	A pointer to a provider-specific entity. This member is required for the NS_EMAIL namespace. This member is optional, dependent on the requirements for other NSPv2 service providers.

> **Note** It is acceptable for the **iProtocol** member of the CSADDR_INFO structure to contain the manifest constant **IPROTOCOL_ANY**, indicating a wildcard value. The namespace provider should substitute an acceptable value for the given address family and socket type.

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows
Header	ws2spi.h

See also

CSADDR_INFO

NSPV2_ROUTINE

NSPv2Cleanup

NSPv2ClientSessionRundown

NSPv2LookupServiceBegin

NSPv2LookupServiceEnd

NSPv2LookupServiceNextEx

NSPv2Startup

WSAQUERYSET2

WSASetLastError

LPNSPV2STARTUP callback function (ws2spi.h)

10/22/2022 • 2 minutes to read • Edit Online

The **NSPv2Startup** function notifies a namespace service provider version-2 (NSPv2) provider that a new client process is to begin using the provider.

Syntax

```
LPNSPV2STARTUP Lpnspv2startup;

INT Lpnspv2startup(
  [in] LPGUID lpProviderId,
  [in] LPVOID *ppvClientSessionArg
)
{...}
```

Parameters

`[in] lpProviderId`

A pointer to the GUID of the specific namespace provider to notify.

`[in] ppvClientSessionArg`

A pointer to the client session.

Return value

The function should return **NO_ERROR** (zero) if the routine succeeds. It should return **SOCKET_ERROR** (that is, 1) if the routine fails and it must set the appropriate error code using WSASetLastError.

ERROR CODE	MEANING
WSA_NOT_ENOUGH_MEMORY	There is not enough memory available to perform this operation.
WSAEACCES	The calling routine does not have sufficient privileges to initialize the service.
WSAEINVAL	One or more parameters were invalid, or missing, for this provider.
WSAEOPNOTSUPP	The operation is not supported. This error is returned if the namespace provider does not implement this function.
WSASERVICE_NOT_FOUND	Service is unknown. The service cannot be found in the specified namespace.

Remarks

The **NSPv2Startup** function is used as part of the namespace service provider version-2 (NSPv2) architecture available on Windows Vista and later.

On Windows Vista and Windows Server 2008, the **NSPv2Startup** function can only be used for operations on NS_EMAIL namespace providers.

The **NSPv2Startup** function is called each time a new client process begins using namespace provider. Providers may use the client session argument pointed to by the *ppvClientSessionArg* parameter to store information about this session. The value in the *ppvClientSessionArg* parameter will be passed to subsequent NSPv2 function calls in the same session. The client session argument may **NULL**, if the namespace provider does not require this information.

The **NSPv2Startup** function is called when a new client session initializes. The **NSPv2Startup** and NSPv2Cleanup functions must be called as pairs.

The **NSPv2Startup** function must be called successfully before calling the NSPv2Cleanup function. It is permissible to make more than one **NSPv2Startup** call. However, for each **NSPv2Startup** call, a corresponding **NSPv2Cleanup** call must also be issued. Only the final **NSPv2Cleanup** for the service provider does the actual cleanup; the preceding calls decrement an internal reference count in the namespace service provider.

The **NSPv2Startup**, NSPv2ClientSessionRundown, and NSPv2Cleanup functions are optional, dependent on the requirements of the NSPv2 provider.

If the **NSPv2Startup** function isn't implemented, then calls to that function should be intercepted by a stub function that returns WSAEOPNOTSUPP. The NSPv2 function pointer to the unimplemented **NSPv2Startup** function in the NSPV2_ROUTINE structure should point be to the stub function.

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows
Header	ws2spi.h

See also

NSPV2_ROUTINE

NSPv2Cleanup

NSPv2ClientSessionRundown

NSPv2LookupServiceBegin

NSPv2LookupServiceEnd

NSPv2LookupServiceNextEx

NSPv2SetServiceEx

WSAQUERYSET2

LPWSPACCEPT callback function (ws2spi.h)

10/22/2022 • 6 minutes to read • Edit Online

The **LPWSPAccept** function conditionally accepts a connection based on the return value of a condition function.

Syntax

```
LPWSPACCEPT Lpwspaccept;

SOCKET Lpwspaccept(
  [in]      SOCKET s,
  [out]     sockaddr *addr,
  [in, out] LPINT addrlen,
  [in]      LPCONDITIONPROC lpfnCondition,
  [in]      DWORD_PTR dwCallbackData,
  [out]     LPINT lpErrno
)
{...}
```

Parameters

`[in] s`

Descriptor identifying a socket that is listening for connections after a LPWSPListen.

`[out] addr`

Optional pointer to a buffer that receives the address of the connecting entity, as known to the service provider. The exact format of the *addr* parameter is determined by the address family established when the socket in the sockaddr structure was created.

`[in, out] addrlen`

Optional pointer to an integer that contains the length of the *addr* parameter, in bytes.

`[in] lpfnCondition`

Procedure instance address of an optional-condition function furnished by Windows Sockets. This function is used in the accept or reject decision based on the caller information passed in as parameters.

`[in] dwCallbackData`

Callback data to be passed back to the Windows Socket 2 client as the value of the *dwCallbackData* parameter of the condition function. This parameter is not interpreted by the service provider.

`[out] lpErrno`

Pointer to the error code.

Return value

If no error occurs, **LPWSPAccept** returns a value of type SOCKET that is a descriptor for the accepted socket. Otherwise, a value of INVALID_SOCKET is returned, and a specific error code is available in *lpErrno*.

ERROR CODE	MEANING
WSAECONNREFUSED	The connection request was forcefully rejected as indicated in the return value of the condition function (CF_REJECT).
WSAECONNRESET	An incoming connection was indicated, but was subsequently terminated by the remote peer prior to accepting the call.
WSAENETDOWN	The network subsystem has failed.
WSAEFAULT	The *addrlen* parameter is too small or the *lpfnCondition* parameter is not part of the user address space.
WSAEINTR	A (blocking) call was canceled through LPWSPCancelBlockingCall.
WSAEINPROGRESS	A blocking Windows Sockets call is in progress.
WSAEINVAL	LPWSPListen was not invoked prior to LPWSPAccept, parameter *g* specified in the condition function is not a valid value, the return value of the condition function is not a valid one, or any case where the specified socket is in an invalid state.
WSAEMFILE	Queue is nonempty upon entry to LPWSPAccept and there are no socket descriptors available.
WSAENOBUFS	No buffer space is available.
WSAENOTSOCK	The descriptor is not a socket.
WSAEOPNOTSUPP	Referenced socket is not a type that supports connection-oriented service.
WSATRY_AGAIN	Acceptance of the connection request was deferred as indicated in the return value of the condition function (CF_DEFER).
WSAEWOULDBLOCK	Socket is marked as nonblocking and no connections are present to be accepted.
WSAEACCES	The connection request that was offered has timed out or been withdrawn.

Remarks

The **LPWSPAccept** function extracts the first connection on the queue of pending connections on socket *s*, and

checks it against the condition function, provided the condition function is specified (that is, not null). The condition function must be executed in the same thread as this routine. If the condition function returns CF_ACCEPT, **LPWSPAccept** creates a new socket.

Newly created sockets have the same properties as the socket *s*, including network events registered with **LPWSPAsyncSelect** or with **LPWSPEventSelect**. As described in **DescriptorAllocation**, when new socket descriptors are allocated, IFS providers must call **WPUModifyIFSHandle** and non-IFS providers must call **WPUCreateSocketHandle**.

If the condition function returns CF_REJECT, **LPWSPAccept** rejects the connection request. If the application's accept/reject decision cannot be made immediately, the condition function will return CF_DEFER to indicate that no decision has been made. No action about this connection request is to be taken by the service provider. When the application is ready to take action on the connection request, it will invoke **LPWSPAccept** again and return either CF_ACCEPT or CF_REJECT as a return value from the condition function.

For sockets that are in the (default) blocking mode, if no pending connections are present on the queue, **LPWSPAccept** blocks the caller until a connection is present. For sockets in nonblocking mode, if this function is called when no pending connections are present on the queue, **LPWSPAccept** returns the error code WSAEWOULDBLOCK. The accepted socket cannot be used to accept more connections. The original socket remains open.

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

The prototype of the condition function is as follows.

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

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

The *lpSQOS* parameter references the flow specifications for socket *s* specified by the caller, one for each direction, followed by any additional provider-specific parameters. The sending or receiving flow specification values will be ignored as appropriate for any unidirectional sockets. A null value for *lpSQOS* indicates that there is no caller-supplied QoS and that no negotiation is possible. A non-**NULL** *lpSQOS* pointer indicates that a QoS negotiation is to occur or that the provider is prepared to accept the QoS request without negotiation.

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

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

The *dwCallbackData* parameter value passed to the condition function is the value passed as the *dwCallbackData* parameter in the original **LPWSPAccept** call. This value is interpreted only by the Windows Sockets 2 client. This allows a client to pass some context information from the **LPWSPAccept** call site through to the condition function, which provides the condition function with any additional information required to determine whether to accept the connection. A typical usage is to pass a (suitably cast) pointer to a data structure containing references to application-defined objects with which this socket is associated.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	ws2spi.h

See also

LPWSPAsyncSelect

LPWSPBind

LPWSPConnect

LPWSPEventSelect

LPWSPGetSockOpt

LPWSPListen

LPWSPSelect

LPWSPSocket

LPWSPADDRESSTOSTRING callback function (ws2spi.h)

10/22/2022 • 2 minutes to read • Edit Online

The **LPWSPAddressToString** function converts all components of a sockaddr structure into a human-readable numeric string representation of the address. This is used mainly for display purposes.

Syntax

```
LPWSPADDRESSTOSTRING Lpwspaddresstostring;

INT Lpwspaddresstostring(
  [in]      LPSOCKADDR lpsaAddress,
  [in]      DWORD dwAddressLength,
  [in]      LPWSAPROTOCOL_INFOW lpProtocolInfo,
  [out]     LPWSTR lpszAddressString,
  [in, out] LPDWORD lpdwAddressStringLength,
  [out]     LPINT lpErrno
)
{...}
```

Parameters

`[in] lpsaAddress`

Pointer to a sockaddr structure to translate into a string.

`[in] dwAddressLength`

Length of the address of sockaddr, in bytes.

`[in] lpProtocolInfo`

(required) WSAProtocol_Info structure associated with the provider that will do the translation.

`[out] lpszAddressString`

Buffer that receives the human-readable address string..

`[in, out] lpdwAddressStringLength`

Length of the *AddressString* buffer, in bytes. Returns the length of the string actually copied into the buffer. If the supplied buffer is not large enough, the function fails with a specific error of WSAEFAULT and this parameter is updated with the required size, in bytes.

`[out] lpErrno`

Pointer to the error code.

Return value

If no error occurs, **LPWSPAddressToString** returns zero. Otherwise, it returns SOCKET_ERROR, and a specific error code is available in *lpErrno*.

ERROR CODE	MEANING
WSAEFAULT	The specified AddressString buffer is too small. Pass in a larger buffer.
WSAEINVAL	The specified address is not a valid socket address, or its address family is not supported by the provider, or the specified *lpProtocolInfo* did not refer to a WSAProtocol_Info structure supported by the provider.

Remarks

A layered service provider supplies an implementation of this function, but it is also a client of this function if and when it calls **LPWSPAddressToString** of the next layer in the protocol chain. Some special considerations apply to the *lpProtocolInfo* parameter as it is propagated down through the layers of the protocol chain.

If the next layer in the protocol chain is another layer, then, when the next layer's **LPWSPAddressToString** is called, this layer must pass to the next layer a *lpProtocolInfo* parameter that references the same unmodified WSAProtocol_Info structure with the same unmodified chain information. However, if the next layer is the base protocol (that is, the last element in the chain), this layer performs a substitution when calling the base provider's **LPWSPAddressToString**. In this case, the base provider's **WSAPROTOCOL_INFO** structure should be referenced by the *lpProtocolInfo* parameter. One vital benefit of this policy is that base service providers do not have to be aware of protocol chains.

This same propagation policy applies when propagating a WSAProtocol_Info structure through a layered sequence of other functions such as LPWSPDuplicateSocket, WSPStartup, LPWSPSocket, or LPWSPStringToAddress.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	ws2spi.h

See also

WSAProtocol_Info

WSPDucplicateSocket

LPWSPSocket

WSPStartup

sockaddr

LPWSPASYNCSELECT callback function (ws2spi.h)

10/22/2022 • 12 minutes to read • Edit Online

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

Syntax

```
LPWSPASYNCSELECT Lpwspasyncselect;

int Lpwspasyncselect(
  [in]  SOCKET s,
  [in]  HWND hWnd,
  [in]  unsigned int wMsg,
  [in]  long lEvent,
  [out] LPINT lpErrno
)
{...}
```

Parameters

`[in] s`

Descriptor identifying the socket for which event notification is required.

`[in] hWnd`

Handle identifying the window that should receive a message when a network event occurs.

`[in] wMsg`

Message to be sent when a network event occurs.

`[in] lEvent`

Bitmask that specifies a combination of network events in which the Windows Sockets service provider interface (SPI) client is interested. Constructed by using the bitwise OR operator with any of these values.

VALUE	MEANING
FD_READ	Issues notification of readiness for reading.
FD_WRITE	Issues notification of readiness for writing.
FD_OOB	Issues notification of the arrival of OOB data.
FD_ACCEPT	Issues notification of incoming connections.

FD_CONNECT	Issues notification of completed connections.
FD_CLOSE	Issues notification of socket closure.
FD_QOS	Issues notification of socket quality of service (QoS) changes.
FD_GROUP_QOS	Reserved.
FD_ROUTING_INTERFACE_CHANGE	Issues notification of routing interface change for the specified destination.
FD_ADDRESS_ LIST_CHANGE	Issues notification of local address list change for the socket's protocol family.

`[out] lpErrno`

Pointer to the error code. See the **Return value** section for more info.

Return value

The return value is zero if the Windows Sockets SPI client's declaration of interest in the network event set was successful. Otherwise, the value SOCKET_ERROR is returned, and a specific error code is available in *lpErrno*.

ERROR CODE	MEANING
WSAENETDOWN	The network subsystem has failed.
WSAEINVAL	Indicates that one of the specified parameters was invalid such as the window handle not referring to an existing window, or the specified socket is in an invalid state.
WSAEINPROGRESS	A blocking Windows Sockets call is in progress, or the service provider is still processing a callback function.
WSAENOTSOCK	The descriptor is not a socket.

See **Remarks** for info about additional error codes that can be set (in the high word of *lParam* within the message) when an application window receives a message.

Remarks

This function is used to request that the service provider send a Windows message to the client's window *hWnd* whenever the service provider detects any of the network events specified by the *lEvent* argument. The service provider should use the WPUPostMessage function to post the message. The message to be sent is specified by the *wMsg* parameter. The socket for which notification is required is identified by *s*.

This function automatically sets socket *s* to nonblocking mode, regardless of the value of *lEvent*. See LPWSPIoctl about how to set the socket back to blocking mode.

Invoking **LPWSPAsyncSelect** for a socket cancels any previous **LPWSPAsyncSelect** or LPWSPEventSelect for the same socket. For example, to receive notification for both reading and writing, the Windows Sockets SPI client must call **LPWSPAsyncSelect** with both FD_READ and FD_WRITE, like this.

```
rc = WSPAsyncSelect(s, hWnd, wMsg, FD_READ | FD_WRITE, &error);
```

It's not possible to specify different messages for different events. The following code won't work; the second call cancels the effects of the first, and the only association will be the FD_WRITE event associated with wMsg2.

```
// Incorrect example.
rc = WSPAsyncSelect(s, hWnd, wMsg1, FD_READ, &error);
rc = WSPAsyncSelect(s, hWnd, wMsg2, FD_WRITE, &error);
```

To cancel all notification (that is, to indicate that the service provider should send no further messages related to network events on the socket), set *lEvent* to zero.

```
rc = WSPAsyncSelect(s, hWnd, 0, 0, &error);
```

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

When one of the nominated network events occurs on the specified socket *s*, the service provider uses WPUPostMessage to send message *wMsg* to the Windows Sockets SPI client's window *hWnd*. In the posted message, the *wParam* argument identifies the socket on which a network event has occurred. The low word of *lParam* specifies the network event that has occurred. The possible network event codes that may be indicated are as follows.

VALUE	MEANING
FD_READ	Socket *s* is ready for reading
FD_WRITE	Socket *s* is ready for writing
FD_OOB	Out-of-band data is ready for reading on socket *s*
FD_ACCEPT	Socket *s* is ready to accept a new incoming connection
FD_CONNECT	The connection that was initiated on socket *s* has completed
FD_CLOSE	The connection identified by socket *s* has been closed
FD_QOS	The quality of service associated with socket *s* has changed

VALUE	MEANING
FD_GROUP_QOS	Reserved for future use with socket groups: Quality of service associated with the socket group to which socket *s* belongs has changed
FD_ROUTING_INTERFACE_CHANGE	The local interface that should be used to send to the specified destination has changed
FD_ADDRESS_LIST_CHANGE	The list of addresses of the socket's protocol family to which the Windows Sockets SPI client can bind has changed

The high word of *lParam* contains any error code (it can be extracted by using the **WSAGETSELECTERROR** macro). The error code be any error as defined in `ws2spi.h`. Possible error codes for each network event are listed in the following table.

Event: FD_CONNECT

ERROR CODE	MEANING
WSAEAFNOSUPPORT	Addresses in the specified family cannot be used with this socket.
WSAECONNREFUSED	The attempt to connect was rejected.
WSAENETUNREACH	The network cannot be reached from this host at this time.
WSAEFAULT	The *namelen* parameter is invalid.
WSAEINVAL	The socket is already bound to an address.
WSAEISCONN	The socket is already connected.
WSAEMFILE	No more file descriptors are available.
WSAENOBUFS	No buffer space is available. The socket cannot be connected.
WSAENOTCONN	The socket is not connected.
WSAETIMEDOUT	Attempt to connect timed out without establishing a connection.

Event: FD_CLOSE

ERROR CODE	MEANING
WSAENETDOWN	The network subsystem failed.
WSAECONNRESET	The connection was reset by the remote side.
WSAECONNABORTED	The connection was terminated due to a time-out or other failure.

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

ERROR CODE	MEANING
WSAENETDOWN	The network subsystem failed.

Event: FD_ROUTING_INTERFACE_CHANGE

ERROR CODE	MEANING
WSAENETUNREACH	The specified destination is no longer reachable.
WSAENETDOWN	The network subsystem failed.

Although **LPWSPAsyncSelect** can be called with interest in multiple events, the service provider issues the same Windows message for each event.

A Windows Sockets 2 provider shouldn't continually flood a Windows Sockets SPI client with messages for a particular network event. Having successfully posted notification of a particular event to a Windows Sockets SPI client window, no further message(s) for that network event will be posted to the Windows Sockets SPI client window until the Windows Sockets SPI client makes the function call that implicitly re-enables notification of that network event.

NETWORK EVENT	RE-ENABLING FUNCTION
FD_READ	LPWSPRecv or LPWSPRecvFrom
FD_WRITE	LPWSPSend or LPWSPSendTo
FD_OOB	LPWSPRecv or LPWSPRecvFrom
FD_ACCEPT	LPWSPAccept, unless the error code returned is WSATRY_AGAIN indicating that the condition function returned CF_DEFER
FD_CONNECT	NONE
FD_CLOSE	NONE

NETWORK EVENT	RE-ENABLING FUNCTION
FD_QOS	LPWSPIoctl with SIO_GET_QOS
FD_GROUP_QOS	Reserved for future use with socket groups: LPWSPIoctl with SIO_GET_GROUP_QOS
FD_ROUTING_INTERFACE_CHANGE	LPWSPIoctl with command SIO_ROUTING_INTERFACE_CHANGE
FD_ADDRESS_LIST_CHANGE	LPWSPIoctl with command SIO_ADDRESS_LIST_CHANGE

Any call to the re-enabling routine, even one that fails, results in re-enabling of message-posting for the relevant event.

For FD_READ, FD_OOB, and FD_ACCEPT events, message-posting is *level-triggered*. This means that if the re-enabling routine is called and the relevant condition is still met after the call, an **LPWSPAsyncSelect** message is posted to the Windows Sockets SPI client.

The FD_QOS and FD_GROUP_QOS events are considered *edge-triggered*. A message will be posted exactly once when a QOS change occurs. Further messages won't be forthcoming until either the provider detects a further change in QOS, or the Windows Sockets SPI client renegotiates the QOS for the socket.

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

If any event has already happened when the Windows Sockets SPI client calls **LPWSPAsyncSelect**, or when the re-enabling function is called, then a message is posted as appropriate. For example, consider the following sequence.

1. A Windows Sockets SPI client calls LPWSPListen.
2. A connect request is received, but not yet accepted.
3. The Windows Sockets SPI client calls **LPWSPAsyncSelect** specifying that it wants to receive FD_ACCEPT messages for the socket. Due to the persistence of events, the WinSock service provider posts an FD_ACCEPT message immediately.

The FD_WRITE event is handled slightly differently. An FD_WRITE message is posted when a socket is first connected with LPWSPConnect (after FD_CONNECT, if also registered) or accepted with LPWSPAccept, and then after an LPWSPSend or LPWSPSendTo fails with WSAEWOULDBLOCK and buffer space becomes available. Therefore, a Windows Sockets SPI client can assume that sends are possible starting from the first FD_WRITE message and lasting until a send returns WSAEWOULDBLOCK. After such a failure, the Windows Sockets SPI client will be notified that sends are again possible with an FD_WRITE message.

The FD_OOB event is used only when a socket is configured to receive out-of-band data separately. If the socket is configured to receive out-of-band data in-line, then the out-of-band (expedited) data is treated as normal data, and the Windows Sockets SPI client must register an interest in FD_READ events, not FD_OOB events.

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

The FD_CLOSE message is posted when a close indication is received for the virtual circuit corresponding to the socket. In TCP terms, this means that the FD_CLOSE is posted when the connection goes into the TIME WAIT or

CLOSE WAIT states. This results from the remote end performing an LPWSPShutdown on the send side or an LPWSPCloseSocket. It's correct for FD_CLOSE to be posted only after all data is read from a socket.

In the case of a graceful close, the service provider should send an FD_CLOSE message to indicate virtual circuit closure only after all the received data has been read. It shouldn't send an FD_READ message to indicate this condition.

The FD_QOS or FD_GROUP_QOS message is posted when there has been a change to any field in the flow spec associated with socket *s*, or the socket group that *s* belongs to, respectively. The service provider must update the QOS information available to the client via LPWSPIoctl with SIO_GET_QOS and/or SIO_GET_GROUP_QOS.

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

The FD_ADDRESS_LIST_CHANGE message is posted when the list of addresses to which the Windows Sockets SPI client can bind changes *after* LPWSPIoctl with SIO_ADDRESS_LIST_CHANGE has been issued.

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

FD_READ

1. When **LPWSPAsyncSelect** is called, if there is data currently available to receive.
2. When data arrives, if FD_READ not already posted.
3. After LPWSPRecv or LPWSPRecvFrom is called (with or without MSG_PEEK), if data is still available to receive.

When LPWSPSetSockOpt SO_OOBINLINE is enabled, *data* includes both normal data and out-of-band (OOB) data in the instances noted above.

FD_WRITE

1. When **LPWSPAsyncSelect** is called, if an LPWSPSend or LPWSPSendTo is possible.
2. After LPWSPConnect or LPWSPAccept is called, when connection established.
3. After LPWSPSend or LPWSPSendTo fail with WSAEWOULDBLOCK, when LPWSPSend or LPWSPSendTo are likely to succeed.
4. After LPWSPBind on a connectionless socket. FD_WRITE may or may not occur at this time (implementation-dependent). In any case, a connectionless socket is always writable immediately after LPWSPBind.

FD_OOB (valid only when LPWSPSetSockOpt SO_OOBINLINE is disabled (default))

1. When **LPWSPAsyncSelect** is called, if there is OOB data currently available to receive with the MSG_OOB flag.
2. When OOB data arrives, if FD_OOB not already posted.
3. After LPWSPRecv or LPWSPRecvFrom is called with or without MSG_OOB flag, if OOB data is still available to receive.

FD_ACCEPT

1. When **LPWSPAsyncSelect** is called, if there is currently a connection request available to accept.
2. When a connection request arrives, if FD_ACCEPT not already posted.
3. After LPWSPAccept is called, if there is another connection request available to accept.

FD_CONNECT

1. When **LPWSPAsyncSelect** is called, if there is currently a connection established.
2. After LPWSPConnect is called, when connection is established (even when LPWSPConnect succeeds immediately, as is typical with a datagram socket), and even when it fails immediately).

3. After WSPJoinLeaf is called, when the join operation completes.
4. After connect, WSAConnect, or WSPJoinLeaf was called with a non-blocking, connection-oriented socket. The initial operation returned with a specific error of WSAEWOULDBLOCK, but the network operation went ahead. Whether the operation eventually succeeds or not, when the outcome has been determined, FD_CONNECT happens. The client should check the error code to determine whether the outcome was a success or failure.

FD_CLOSE (valid only on connection-oriented sockets (for example, SOCK_STREAM))

1. When **LPWSPAsyncSelect** is called, if the socket connection has been closed.
2. After the remote system initiated a graceful close, when no data is currently available to receive (if data has been received and is waiting to be read when the remote system initiates a graceful close, then the FD_CLOSE is not delivered until all pending data has been read).
3. After the local system initiates a graceful close with LPWSPShutdown and the remote system has responded with an *end-of-data* notification (such as TCP FIN), when no data is currently available to receive.
4. When the remote system aborts the connection (for example, sent TCP RST), and *lParam* will contain the WSAECONNRESET error value.

FD_CLOSE is not posted after LPWSPCloseSocket is called.

FD_QOS

1. When **LPWSPAsyncSelect** is called, if the QOS associated with the socket has been changed.
2. After LPWSPIoctl with SIO_GET_QOS is called, when the QOS is changed.

FD_GROUP_QOS

Reserved for future use with socket groups:

1. When **LPWSPAsyncSelect** is called, if the group QOS associated with the socket has been changed.
2. After LPWSPIoctl with SIO_GET_GROUP_QOS is called, when the group QOS is changed.

FD_ROUTING_INTERFACE_CHANGE

1. after LPWSPIoctl with SIO_ROUTING_INTERFACE_CHANGE is called, when the local interface that should be used to reach the destination specified in the IOCTL changes.

FD_ADDRESS_LIST_CHANGE

1. after LPWSPIoctl with SIO_ADDRESS_LIST_CHANGE is called, when the list of local addresses to which the Windows Sockets SPI client can bind changes.

Requirements

Minimum supported client	Windows 10 Build 20348
Minimum supported server	Windows 10 Build 20348
Header	ws2spi.h

See also

LPWSPAsyncSelect callback function

LPWSPBIND callback function (ws2spi.h)

10/22/2022 • 3 minutes to read • Edit Online

The **LPWSPBind** function associates a local address (that is, name) with a socket.

Syntax

```
LPWSPBIND Lpwspbind;

int Lpwspbind(
  [in]  SOCKET s,
  [in]  const sockaddr *name,
  [in]  int namelen,
  [out] LPINT lpErrno
)
{...}
```

Parameters

`[in] s`

A descriptor identifying an unbound socket.

`[in] name`

The address to assign to the socket, in the form of a **sockaddr** structure.

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

`[in] namelen`

The length, in bytes, of structure pointed to by the *name* parameter.

`[out] lpErrno`

A pointer to the error code.

Return value

If no error occurs, **LPWSPBind** returns zero. Otherwise, it returns SOCKET_ERROR, and a specific error code is available in *lpErrno*.

ERROR CODE	MEANING
WSAENETDOWN	The network subsystem has failed.

WSAEADDRINUSE	Some process on the local computer has already bound to the same fully qualified address (for example, IP address and port in the **AF_INET** case) and the socket has not been marked to allow address reuse with SO_REUSEADDR. (See the SO_REUSEADDR socket option under LPWSPSetSockOpt.)
WSAEADDRNOTAVAIL	The specified address is not a valid address for this computer.
WSAEFAULT	The *name* or the *namelen* parameter is not a valid part of the user address space, the *namelen* parameter is too small, the *name* parameter contains incorrect address format for the associated address family, or the first two bytes of the memory block specified by *name* do not match the address family associated with the socket descriptor *s*.
WSAEINPROGRESS	Function is invoked when a callback is in progress.
WSAEINVAL	The socket is already bound to an address.
WSAENOBUFS	There are not enough buffers available, there are too many connections.
WSAENOTSOCK	The descriptor is not a socket.

Remarks

The **LPWSPBind** function is used on an unconnected connectionless or connection-oriented socket, before subsequent calls to the **LPWSPConnect** or **LPWSPListen** functions. When a socket is created with **LPWSPSocket**, it exists in a namespace (address family), but it has no name or local address assigned. The **LPWSPBind** function establishes the local association of the socket by assigning a local name to an unnamed socket.

As an example, in the Internet address family, a name consists of three parts: the address family, a host address, and a port number that identifies the Winsock SPI client. In Windows Sockets 2, the *name* parameter is not strictly interpreted as a pointer to a **sockaddr** structure. Service providers are free to regard it as a pointer to a block of memory of size *namelen*. The first two bytes in this block (corresponding to **sa_family** in the **sockaddr** declaration) must contain the address family that was used to create the socket. Otherwise, the error WSAEFAULT will be indicated.

If a Windows Sockets 2 SPI client does not care what local address is assigned to it, it will specify the manifest constant value **ADDR_ANY** for the **sa_data** member of the *name* parameter. This instructs the service provider to use any appropriate network address. For TCP/IP, if the port is specified as zero, the service provider will assign a unique port to the Winsock SPI client with a value between 1024 and 5000. The SPI client can use **LPWSPGetSockName** after **LPWSPBind** to learn the address and the port that has been assigned to it. However, note that if the Internet address is equal to INADDR_ANY, **LPWSPGetSockOpt** will not necessarily be able to supply the address until the socket is connected, since several addresses can be valid if the host is multihomed.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	ws2spi.h

See also

sockaddr

LPWSPConnect

LPWSPGetSockName

LPWSPListen

WSPSetSockOpt

LPWSPSocket

LPWSPCANCELBLOCKINGCALL callback function (ws2spi.h)

10/22/2022 • 2 minutes to read • Edit Online

The **LPWSPCancelBlockingCall** function cancels a blocking call that is currently in progress.

Syntax

```
LPWSPCANCELBLOCKINGCALL Lpwspcancelblockingcall;

int Lpwspcancelblockingcall(
  [out] LPINT lpErrno
)
{...}
```

Parameters

`[out] lpErrno`

Pointer to the error code.

Return value

The value returned by **LPWSPCancelBlockingCall** is zero if the operation was successfully canceled. Otherwise, the value SOCKET_ERROR is returned, and a specific error code is available in *lpErrno*.

ERROR CODE	MEANING
WSAENETDOWN	The network subsystem has failed.
WSAEINVAL	Indicates there is no outstanding blocking call.

Remarks

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

- A Windows Sockets SPI client is processing a message that has been received while a service provider is implementing pseudoblocking. In this case, **WSAIsBlocking** will be true.
- A blocking call is in progress and the Winsock service provider has called back to the Winsock SPI client's blocking hook function (through the callback function retrieved from **WPUQueryBlockingCallback**), which in turn is invoking this function. Such a situation might arise, for instance, in implementing a **Cancel** option for an operation that requires an extended time to complete.

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

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

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

Canceling any operation other than **LPWSPAccept** and **LPWSPSelect** can leave the socket in an indeterminate state. If a Winsock SPI client cancels a blocking operation on a socket, the only operation the Winsock SPI client will be able to perform on the socket is a call to **LPWSPCloseSocket**, although other operations can work on some Winsock service providers. If a Winsock SPI client requires maximum portability, it must be careful not to depend on performing operations after a cancel operation. A Winsock SPI client can reset the connection by setting the time-out on SO_LINGER to zero and calling **LPWSPCloseSocket**.

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

It is acceptable for **LPWSPCancelBlockingCall** to return successfully if the blocking network operation completes prior to being canceled. In this case, the blocking operation will return successfully as if **LPWSPCancelBlockingCall** had never been called. The only way for the Winsock SPI client to confirm that an operation was actually canceled is to check for a return code of WSAEINTR from the blocking call.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	ws2spi.h

See also

WPUQueryBlockingCallback

WSAIsBlocking

LPWSPAccept

LPWSPCloseSocket

LPWSPSelect

LPWSPSocket

LPWSPCLEANUP callback function (ws2spi.h)

10/22/2022 • 2 minutes to read • Edit Online

The **LPWSPCleanup** function terminates use of the Windows Sockets service provider.

Syntax

```
LPWSPCLEANUP Lpwspcleanup;

int Lpwspcleanup(
  [out] LPINT lpErrno
)
{...}
```

Parameters

`[out] lpErrno`

Pointer to the error code.

Return value

The return value is zero if the operation has been successfully initiated. Otherwise, the value SOCKET_ERROR is returned, and a specific error number is available in *lpErrno*.

ERROR CODE	MEANING
WSANOTINITIALISED	A successful **WSPStartup** call must occur before using this function.
WSAENETDOWN	The network subsystem has failed.
WSAEINVAL	Provider identifier given to the name-space provider is not managed by the name-space provider.

Remarks

The Windows Sockets 2 SPI client is required to perform a successful **WSPStartup** call before it can use Winsock service providers. When it has completed the use of Winsock service providers, the SPI client will call **LPWSPCleanup** to deregister itself from a Winsock service provider and allow the service provider to free any resources allocated on behalf of the Windows Sockets 2 client. It is permissible for SPI clients to make more than one **WSPStartup** call. For each **WSPStartup** call, a corresponding **LPWSPCleanup** call will also be issued. Only the final **LPWSPCleanup** for the service provider does the actual cleanup; the preceding calls simply decrement an internal reference count in the Winsock service provider.

When the internal reference count reaches zero and actual cleanup operations commence, any pending blocking or asynchronous calls issued by any thread in this process are canceled without posting any notification messages or signaling any event objects. Any pending overlapped send and receive operations (**LPWSPSend**,

LPWSPSendTo, **LPWSPRecv**, **LPWSPRecvFrom** with an overlapped socket) issued by any thread in this process are also canceled without setting the event object or invoking the completion routine, if specified. In this case, the pending overlapped operations fail with the error status WSA_OPERATION_ABORTED. Any sockets open when **LPWSPCleanup** is called are reset and automatically deallocated as if **LPWSPCloseSocket** was called; sockets that have been closed with **LPWSPCloseSocket** but still have pending data to be sent are not affected, but the pending data is still sent.

This function should not return until the service provider DLL is prepared to be unloaded from memory. In particular, any data remaining to be transmitted must either already have been sent or be queued for transmission by portions of the transport stack that will not be unloaded from memory along with the service provider's DLL.

A Winsock service provider must be prepared to deal with a process that terminates without invoking **LPWSPCleanup** (for example, as a result of an error). A Winsock service provider must ensure that **LPWSPCleanup** leaves things in a state in which the Ws2_32.dll can immediately invoke **WSPStartup** to reestablish Winsock usage.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	ws2spi.h

See also

LPWSPCloseSocket

LPWSPShutdown

WSPStartup

LPWSPCLOSESOCKET callback function (ws2spi.h)

10/22/2022 • 3 minutes to read • Edit Online

The **LPWSPCloseSocket** function closes a socket.

Syntax

```
LPWSPCLOSESOCKET Lpwspclosesocket;

int Lpwspclosesocket(
  [in]  SOCKET s,
  [out] LPINT lpErrno
)
{...}
```

Parameters

`[in] s`

Descriptor identifying a socket.

`[out] lpErrno`

Pointer to the error code.

Return value

If no error occurs, **LPWSPCloseSocket** returns zero. Otherwise, a value of SOCKET_ERROR is returned, and a specific error code is available in *lpErrno*.

ERROR CODE	MEANING
WSAENETDOWN	The network subsystem has failed.
WSAEINPROGRESS	Blocking Windows Sockets call is in progress, or the service provider is still processing a callback function.
WSAENOTSOCK	The descriptor is not a socket.
WSAEWOULDBLOCK	Socket is marked as nonblocking and SO_LINGER is set to a nonzero time-out value.

Remarks

This function closes a socket. More precisely, it releases the socket descriptor *s*, so further references to *s* should fail with the error WSAENOTSOCK. If this is the last reference to an underlying socket, the associated naming information and queued data are discarded. Any blocking or asynchronous calls pending on the socket (issued by any thread in this process) are canceled without posting any notification messages. Any pending overlapped

operations issued by any thread in this process are also canceled. Whatever completion action was specified for these overlapped operations is performed (for example, event, completion routine, or completion port). In this case, the pending overlapped operations fail with the error status WSA_OPERATION_ABORTED . FD_CLOSE will not be posted after **LPWSPCloseSocket** is called.

LPWSPCloseSocket behavior is summarized as follows:

- If SO_DONTLINGER is enabled (the default setting), **LPWSPCloseSocket** returns immediately and connection is gracefully closed in the background.
- If SO_LINGER is enabled with a zero time-out, **LPWSPCloseSocket** returns immediately and the connection is reset/terminated.

  or

- If SO_LINGER is enabled with a nonzero time-out with a blocking socket, **LPWSPCloseSocket** blocks until all data is sent or the time-out expires.
- If SO_LINGER is enabled with a nonzero time-out with a nonblocking socket, **LPWSPCloseSocket** returns immediately, thus indicating failure.

The semantics of **LPWSPCloseSocket** are affected by the socket options SO_LINGER and SO_DONTLINGER as follows.

OPTION	INTERVAL	TYPE OF CLOSE	WAIT FOR CLOSE?
SO_DONTLINGER	Do not care	Graceful	No
SO_LINGER	Zero	Hard	No
SO_LINGER	Nonzero	Graceful	Yes

If SO_LINGER is set (that is, the **l_onoff** member of the linger structure is nonzero) and the time-out interval, **l_linger**, is zero, **LPWSPCloseSocket** is not blocked even if queued data has not yet been sent or acknowledged. This is called a hard or abortive close, because the socket's virtual circuit is reset immediately, and any unsent data is lost. Any **LPWSPRecv** call on the remote side of the circuit will fail with WSAECONNRESET.

If SO_LINGER is set with a nonzero time-out interval on a blocking socket, the **LPWSPCloseSocket** call blocks on a blocking socket until the remaining data has been sent or until the time-out expires. This is called a graceful disconnect. If the time-out expires before all data has been sent, the service provider should terminate the connection before **LPWSPCloseSocket** returns.

Enabling SO_LINGER with a nonzero time-out interval on a nonblocking socket is not recommended. In this case, the call to **LPWSPCloseSocket** will fail with an error of WSAEWOULDBLOCK if the close operation cannot be completed immediately. If **LPWSPCloseSocket** fails with WSAEWOULDBLOCK, the socket handle is still valid and a disconnect is not initiated.

The Winsock SPI client must call **LPWSPCloseSocket** again to close the socket, although **LPWSPCloseSocket** can continue to fail unless the Winsock SPI client does one of the following:

- Disables SO_DONTLINGER.
- Enables SO_LINGER with a zero time-out.
- Calls **LPWSPShutdown** to initiate closure.

If SO_DONTLINGER is set on a stream socket (that is, the **l_onoff** member of the linger structure is zero), the **LPWSPCloseSocket** call will return immediately and does not get WSAEWOULDBLOCK, whether the socket is blocking or nonblocking. However, any data queued for transmission will be sent if possible before the underlying socket is closed. This is called a graceful disconnect and is the default behavior.

Note that in this case the Winsock provider is allowed to retain any resources associated with the socket until such time as the graceful disconnect has completed or the provider terminates the connection due to an inability to complete the operation in a provider-determined amount of time. This can affect Winsock clients that expect to use all available sockets. This is the default behavior; SO_DONTLINGER is set by default.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	ws2spi.h

See also

LPWSPAccept

LPWSPIoctl

WSPSetSockOpt

LPWSPSocket

LPWSPCONNECT callback function (ws2spi.h)

10/22/2022 • 6 minutes to read • Edit Online

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

Syntax

```
LPWSPCONNECT Lpwspconnect;

int Lpwspconnect(
  [in]  SOCKET s,
  [in]  const sockaddr *name,
  [in]  int namelen,
  [in]  LPWSABUF lpCallerData,
  [out] LPWSABUF lpCalleeData,
  [in]  LPQOS lpSQOS,
  [in]  LPQOS lpGQOS,
  [out] LPINT lpErrno
)
{...}
```

Parameters

`[in] s`

Descriptor identifying an unconnected socket.

`[in] name`

Name of the peer to which the socket in the **sockaddr** is to be connected.

`[in] namelen`

Length of the *name*, in bytes.

`[in] lpCallerData`

Pointer to the user data that is to be transferred to the peer during connection establishment.

`[out] lpCalleeData`

Pointer to a buffer into which any user data received from the peer during connection establishment can be copied.

`[in] lpSQOS`

Pointer to the flow specifications for socket *s*, one for each direction.

`[in] lpGQOS`

Reserved.

`[out] lpErrno`

Pointer to the error code.

Return value

If no error occurs, **LPWSPConnect** returns zero. Otherwise, it returns SOCKET_ERROR, and a specific error code is available in *lpErrno*.

On a blocking socket, the return value indicates success or failure of the connection attempt. If the return error code indicates the connection attempt failed (that is, WSAECONNREFUSED, WSAENETUNREACH, WSAETIMEDOUT) the Winsock SPI client can call **LPWSPConnect** again for the same socket.

ERROR CODE	MEANING
WSAENETDOWN	The network subsystem has failed.
WSAEADDRINUSE	Local address of the socket is already in use and the socket was not marked to allow address reuse with SO_REUSEADDR. This error usually occurs at the time of bind, but could be delayed until this function if the bind was to a partially wildcard address (involving ADDR_ANY) and if a specific address needs to be committed at the time of this function.
WSAEINTR	(Blocking) call was canceled through **LPWSPCancelBlockingCall**.
WSAEINPROGRESS	Blocking Winsock call is in progress or the service provider is still processing a callback function.
WSAEALREADY	Nonblocking **LPWSPConnect** call is in progress on the specified socket. In order to preserve backward compatibility, this error is reported as WSAEINVAL to Windows Sockets 1.1 applications that link to either Winsock.dll or Wsock32.dll.
WSAEADDRNOTAVAIL	Remote address is not a valid address (for example, ADDR_ANY).
WSAEAFNOSUPPORT	Addresses in the specified family cannot be used with this socket.
WSAECONNREFUSED	An attempt to connect was rejected.
WSAEFAULT	The *name* or the *namelen* parameter is not a valid part of the user address space, the *namelen* parameter is too small, the buffer length for *lpCalleeData*, *lpSQOS*, and *lpGQOS* is too small, or the buffer length for *lpCallerData* is too large.
WSAEINVAL	Parameter *s* is a listening socket.
WSAEISCONN	Socket is already connected (connection-oriented sockets only).

WSAENETUNREACH	The network cannot be reached from this host at this time.
WSAENOBUFS	No buffer space is available. The socket cannot be connected.
WSAENOTSOCK	The descriptor is not a socket.
WSAEOPNOTSUPP	Flow specifications specified in *lpSQOS* cannot be satisfied.
WSAEPROTONOSUPPORT	The *lpCallerData* augment is not supported by the service provider.
WSAETIMEDOUT	An attempt to connect timed out without establishing a connection.
WSAEWOULDBLOCK	Socket is marked as nonblocking and the connection cannot be completed immediately. It is possible to select the socket using the **LPWSPSelect** function while it is connecting by using the **WSPSelect** function to select it for writing.
WSAEACCES	An attempt to connect datagram socket to broadcast address failed because **WSPSetSockOpt** SO_BROADCAST is not enabled.

Remarks

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

For connection-oriented sockets (for example, type SOCK_STREAM), an active connection is initiated to the specified host using *name* (an address in the namespace of the socket. For a detailed description, see **LPWSPBind**. When this call completes successfully, the socket is ready to send and receive data. If the address member of the **name** structure is all zeroes, **LPWSPConnect** will return the error WSAEADDRNOTAVAIL. Any attempt to reconnect an active connection will fail with the error code WSAEISCONN.

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

For a connectionless socket (for example, type SOCK_DGRAM), the operation performed by **LPWSPConnect** is to establish a default destination address so the socket can be used with subsequent connection-oriented send and receive operations (**LPWSPSend**, **LPWSPRecv**). Any datagrams received from an address other than the destination address specified will be discarded. If the address member of the *name* structure is all zeroes, the socket will be disconnected— the default remote address will be indeterminate, so **LPWSPSend** and

LPWSPRecv calls will return the error code WSAENOTCONN. However, **LPWSPSendTo** and **LPWSPRecvFrom** can still be used. The default destination can be changed by simply calling **LPWSPConnect** again, even if the socket is already connected. Any datagrams queued for receipt are discarded if *name* is different from the previous **LPWSPConnect**.

For connectionless sockets, *name* can indicate any valid address, including a broadcast address. However, to connect to a broadcast address, a socket must have **WSPSetSockOpt** SO_BROADCAST enabled. Otherwise, **LPWSPConnect** will fail with the error code WSAEACCES.

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

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

The *lpCallerData* is a value parameter that contains any user data to be sent along with the connection request. If *lpCallerData* is null, no user data will be passed to the peer. The *lpCalleeData* is a result parameter that will reference any user data passed back from the peer as part of the connection establishment. The *lpCalleeData*->**len** initially contains the length of the buffer allocated by the Winsock SPI client and pointed to by *lpCalleeData*->**buf**. The *lpCalleeData*->**len** will be set to zero if no user data has been passed back. The *lpCalleeData* information will be valid when the connection operation is complete. For blocking sockets, this will be when the **LPWSPConnect** function returns. For nonblocking sockets, this will be after the FD_CONNECT notification has occurred. If *lpCalleeData* is null, no user data will be passed back. The exact format of the user data is specific to the address family the socket belongs to and/or the applications involved.

At connect time, a Winsock SPI client can use the *lpSQOS* parameter to override any previous QoS specification made for the socket through **LPWSPIoctl** with the SIO_SET_QOS opcode.

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

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

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	ws2spi.h

See also

LPWSPAccept

LPWSPBind

LPWSPEnumNetworkEvents

LPWSPEventSelect

LPWSPGetSockName

LPWSPGetSockopt

LPWSPSelect

LPWSPSocket

LPWSPDUPLICATESOCKET callback function (ws2spi.h)

10/22/2022 • 4 minutes to read • Edit Online

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

Syntax

```
LPWSPDUPLICATESOCKET Lpwspduplicatesocket;

int Lpwspduplicatesocket(
  [in]  SOCKET s,
  [in]  DWORD dwProcessId,
  [out] LPWSAPROTOCOL_INFOW lpProtocolInfo,
  [out] LPINT lpErrno
)
{...}
```

Parameters

`[in] s`

Local socket descriptor.

`[in] dwProcessId`

Identifier of the target process for which the shared socket will be used.

`[out] lpProtocolInfo`

Pointer to a buffer allocated by the client that is large enough to contain a WSAPROTOCOL_INFO structure. The service provider copies the protocol information structure contents to this buffer.

`[out] lpErrno`

Pointer to the error code.

Return value

If no error occurs, **LPWSPDuplicateSocket** returns zero. Otherwise, the value of SOCKET_ERROR is returned, and a specific error number is available in *lpErrno*.

ERROR CODE	MEANING
WSAENETDOWN	The network subsystem has failed.
WSAEINVAL	Indicates that one of the specified parameters was invalid.

WSAEINPROGRESS	Blocking Windows Sockets call is in progress or the service provider is still processing a callback function.
WSAEMFILE	No more socket descriptors are available.
WSAENOBUFS	No buffer space is available. The socket cannot be created.
WSAENOTSOCK	The descriptor is not a socket.

Remarks

A source process calls **LPWSPDuplicateSocket** to obtain a special WSAPROTOCOL_INFO structure. It uses some interprocess communications (IPC) mechanism to pass the contents of this structure to a target process, which in turn uses it in a call to LPWSPSocket to obtain a descriptor for the duplicated socket. Note that the special **WSAPROTOCOL_INFO** structure can only be used once by the target process.

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 LPWSPSocket in the target processes' context. The provider must then return a socket descriptor that references a common underlying socket. The **dwProviderReserved** member of the **WSAPROTOCOL_INFO** structure is available for the service provider's use and can be used to store any useful context information, including a duplicated handle.

When a new socket descriptor is allocated, an installable file system (IFS) provider must call WPUModifyIFSHandle, and a non-IFS provider must call WPUCreateSocketHandle. An IFS provider can use the DuplicateHandle function. To ensure proper execution of socket duplication, a non-IFS service provider must use the **LPWSPDuplicateSocket** function.

One possible scenario for establishing and using a shared socket in handoff mode is illustrated in the following.

SOURCE PROCESS	IPC	MEANING
1) LPWSPSocket, LPWSPConnect		
2) Requests target process identifier.	==>	
		3) Receives process identifier request and respond.
4) Receives process identifier.	<==	
5) Calls **LPWSPDuplicateSocket** to get a special WSAPROTOCOL_INFO structure.		

6) Sends WSAPROTOCOL_INFO structure to target.		
	==>	7) Receives WSAPROTOCOL_INFO structure.
		8) Calls LPWSPSocket to create shared socket descriptor.
		9) Uses shared socket for data exchange.
10) LPWSPCloseSocket	<==	

The descriptors that reference a shared socket can be used independently as far as I/O is concerned. However, the Windows Sockets interface does not implement any type of access control, so 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.

Since what is duplicated are the socket descriptors and not the underlying socket, all 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 **LPWSPGetSockopt** from any or all descriptors. A process can call LPWSPCloseSocket on a duplicated socket and the descriptor will become deallocated. The underlying socket, however, will remain open until **LPWSPClosesocket** is called by the last remaining descriptor.

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

A layered service provider supplies an implementation of this function, but it is also a client of this function if and when it calls **LPWSPDuplicateSocket** of the next layer in the protocol chain. Some special considerations apply to this function's *lpProtocolInfo* parameter as it is propagated down through the layers of the protocol chain.

If the next layer in the protocol chain is another layer then when the next layer's **LPWSPDuplicateSocket** is called, this layer must pass to the next layer a *lpProtocolInfo* that references the same unmodified **WSAPROTOCOL_INFO** structure with the same unmodified chain information. However, if the next layer is the base protocol (that is, the last element in the chain), this layer performs a substitution when calling the base provider's **LPWSPDuplicateSocket**. In this case, the base provider's **WSAPROTOCOL_INFO** structure should be referenced by the *lpProtocolInfo* parameter.

One vital benefit of this policy is that base service providers do not have to be aware of protocol chains. This same policy applies when propagating a **WSAPROTOCOL_INFO** structure through a layered sequence of other functions such as **LPWSPAddressToString**, **WSPStartup**, **LPWSPSocket**, or **LPWSPStringToAddress**.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	ws2spi.h

See also

WPUCreateSocketHandle

WPUModifyIFSHandle

LPWSPENUMNETWORKEVENTS callback function (ws2spi.h)

10/22/2022 • 3 minutes to read • Edit Online

The **LPWSPEnumNetworkEvents** function reports occurrences of network events for the indicated socket.

Syntax

```
LPWSPENUMNETWORKEVENTS Lpwspenumnetworkevents;

int Lpwspenumnetworkevents(
  [in]  SOCKET s,
  [in]  WSAEVENT hEventObject,
  [out] LPWSANETWORKEVENTS lpNetworkEvents,
  [out] LPINT lpErrno
)
{...}
```

Parameters

`[in] s`

Descriptor identifying the socket.

`[in] hEventObject`

Optional handle identifying an associated event object to be reset.

`[out] lpNetworkEvents`

Pointer to a WSANETWORKEVENTS structure that is filled with a record of occurred network events and any associated error codes. The **WSANETWORKEVENTS** structure is defined in the following text.

`[out] lpErrno`

Pointer to the error code.

Return value

The return value is zero if the operation was successful. Otherwise, the value SOCKET_ERROR is returned, and a specific error number is available in *lpErrno*.

ERROR CODE	MEANING
WSAENETDOWN	The network subsystem has failed.
WSAEINVAL	Indicates that one of the specified parameters was invalid.

WSAEINPROGRESS	A blocking Windows Sockets call is in progress, or the service provider is still processing a callback function.
WSAENOTSOCK	The descriptor is not a socket.

Remarks

This function is used to report which network events have occurred for the indicated socket since the last invocation of this function. It is intended for use in conjunction with LPWSPEventSelect and LPWSPAsyncSelect, which associate an event object with one or more network events. Recording of network events commences when **LPWSPEventSelect** or **LPWSPAsyncSelect** is called with a nonzero *lNetworkEvents* argument, and remains in effect until another corresponding call is made to **LPWSPEventSelect** or **LPWSPAsyncSelect** with the *lNetworkEvents* argument set to zero.

LPWSPEnumNetworkEvents only reports network activity and errors nominated through **LPWSPEventSelect**. See the descriptions of **LPWSPSelect** and **LPWSPAsyncSelect** to find out how those functions report network activity and errors.

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

The WSANETWORKEVENTS structure is defined on the WSANETWORKEVENTS reference page.

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

Note that only those elements of the *iErrorCode* array are set that correspond to the bits set in the **lNetworkEvents** member. Other members are not modified (this is important for backward compatibility with the Windows Socket 2 SPI clients that are not aware of new FD_ROUTING_INTERFACE_CHANGE and FD_ADDRESS_LIST_CHANGE events).

The following error codes can be returned along with the respective network event.

Event: FD_CONNECT

ERROR CODE	MEANING
WSAEAFNOSUPPORT	Addresses in the specified family cannot be used with this socket.
WSAECONNREFUSED	An attempt to connect was forcefully rejected.
WSAENETUNREACH	The network cannot be reached from this host at this time.

WSAENOBUFS	No buffer space is available. The socket cannot be connected.
WSAETIMEDOUT	An attempt to connect timed out without establishing a connection.

Event: FD_CLOSE

ERROR CODE	MEANING
WSAENETDOWN	The network subsystem has failed.
WSAECONNRESET	The connection was reset by the remote side.
WSAECONNABORTED	The connection was terminated due to a time-out or other failure.

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

ERROR CODE	MEANING
WSAENETDOWN	The network subsystem has failed.

Event: FD_ROUTING_INTERFACE_CHANGE

ERROR CODE	MEANING
WSAENETUNREACH	The specified destination is no longer reachable.
WSAENETDOWN	The network subsystem has failed.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	ws2spi.h

See also

LPWSPEventSelect

LPWSPEVENTSELECT callback function (ws2spi.h)

10/22/2022 • 8 minutes to read • Edit Online

The **LPWSPEventSelect** function specifies an event object to be associated with the supplied set of network events.

Syntax

```
LPWSPEVENTSELECT Lpwspeventselect;

int Lpwspeventselect(
  [in]  SOCKET s,
  [in]  WSAEVENT hEventObject,
  [in]  long lNetworkEvents,
  [out] LPINT lpErrno
)
{...}
```

Parameters

`[in] s`

A descriptor identifying the socket.

`[in] hEventObject`

The handle identifying the event object to be associated with the supplied set of network events.

`[in] lNetworkEvents`

A bitmask that specifies the combination of network events in which the Windows Sockets SPI client has interest. Constructed by using the bitwise OR operator with any of these values.

VALUE	MEANING
FD_READ	Issues notification of readiness for reading.
FD_WRITE	Issues notification of readiness for writing.
FD_OOB	Issues notification of the arrival of OOB data.
FD_ACCEPT	Issues notification of incoming connections.
FD_CONNECT	Issues notification of completed connection.

FD_CLOSE	Issues notification of socket closure.
FD_QOS	Issues notification of socket (QoS) changes.
FD_GROUP_QOS	Reserved.
FD_ROUTING_INTERFACE_CHANGE	Issues notification of routing interface changes for the specified destination(s).
FD_ADDRESS_LIST_CHANGE	Issues notification of local address list changes for the socket's address family.

`[out] lpErrno`

A pointer to the error code. See the **Return value** section for more info.

Return value

The return value is zero if the Windows Sockets SPI client's specification of the network events and the associated event object was successful. Otherwise, the value **SOCKET_ERROR** is returned, and a specific error number is available in *lpErrno*.

ERROR CODE	MEANING
WSAENETDOWN	The network subsystem has failed.
WSAEINVAL	Indicates that one of the specified parameters was invalid, or the specified socket is in an invalid state.
WSAEINPROGRESS	Blocking Windows Sockets call is in progress or the service provider is still processing a callback function.
WSAENOTSOCK	The descriptor is not a socket.

Remarks

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

LPWSPEventSelect operates very similarly to LPWSPAsyncSelect, the difference being in the actions taken when a nominated network event occurs. Whereas **WSPAsyncSelect** causes a Windows Sockets SPI client-specified Windows message to be posted, **LPWSPEventSelect** sets the associated event object and records the occurrence of this event in an internal network event record. A Windows Sockets SPI client can use LPWSPEnumNetworkEvents to retrieve the contents of the internal network event record, and thus determine which of the nominated network events have occurred.

LPWSPEventSelect is the only function that causes network activity and errors to be recorded and retrievable through **LPWSPEnumNetworkEvents**. See the descriptions of **LPWSPSelect** and **LPWSPAsyncSelect** to find out how those functions report network activity and errors.

This function automatically sets socket *s* to nonblocking mode, regardless of the value of *lNetworkEvents*.

Issuing an **LPWSPEventSelect** for a socket cancels any previous **LPWSPAsyncSelect** or **LPWSPEventSelect** for the same socket, and clears the internal network event record. For example, to associate an event object with both reading and writing network events, the Windows Sockets SPI client must call **LPWSPEventSelect** with both FD_READ and FD_WRITE, like this.

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

It's not possible to specify different event objects for different network events. The following code won't work; the second call cancels the effects of the first, and the only association will be the FD_WRITE network event associated with hEventObject2.

```
// Incorrect example.
rc = WSPEventSelect(s, hEventObject1, FD_READ);
rc = WSPEventSelect(s, hEventObject2, FD_WRITE);
```

To cancel the association and selection of network events on a socket, you should set *lNetworkEvents* to zero, in which case the *hEventObject* parameter is ignored.

```
rc = WSPEventSelect(s, hEventObject, 0);
```

Closing a socket with **LPWSPCloseSocket** also cancels the association and selection of network events specified in **LPWSPEventSelect** for the socket. The Windows Sockets SPI client, however, still must call **WSACloseEvent** to explicitly close the event object, and free any resources.

Since an **LPWSPAccept**'ed socket has the same properties as the listening socket used to accept it, any **LPWSPEventSelect** association and network events selection set for the listening socket apply to the accepted socket. For example, if a listening socket has **LPWSPEventSelect** association of *hEventObject* with FD_ACCEPT, FD_READ, and FD_WRITE, then any socket accepted on that listening socket will also have FD_ACCEPT, FD_READ, and FD_WRITE network events associated with the same *hEventObject*. If a different *hEventObject* or network events are desired, then the Windows Sockets SPI client should call **LPWSPEventSelect**, passing the accepted socket, and the desired new information.

Having successfully recorded the occurrence of the network event and signaled the associated event object, no further actions are taken for that network event until the Windows Sockets SPI client makes the function call that implicitly re-enables the setting of that network event and signaling of the associated event object.

NETWORK EVENT	RE-ENABLING FUNCTION
FD_READ	LPWSPRecv or LPWSPRecvFrom
FD_WRITE	LPWSPSend or LPWSPSendTo
FD_OOB	LPWSPRecv or LPWSPRecvFrom
FD_ACCEPT	LPWSPAccept, unless the error code returned is WSATRY_AGAIN indicating that the condition function returned CF_DEFER

NETWORK EVENT	RE-ENABLING FUNCTION
FD_CONNECT	NONE
FD_CLOSE	NONE
FD_QOS	LPWSPIoctl with SIO_GET_QOS
FD_GROUP_QOS	Reserved for future use with socket groups: LPWSPIoctl with SIO_GET_GROUP_QOS
FD_ROUTING_INTERFACE_CHANGE	LPWSPIoctl with command SIO_ROUTING_INTERFACE_CHANGE
FD_ADDRESS_LIST_CHANGE	LPWSPIoctl with command SIO_ADDRESS_LIST_CHANGE

Any call to the re-enabling routine, even one that fails, results in re-enabling of recording and signaling for the relevant network event and event object, respectively.

For FD_READ, FD_OOB, and FD_ACCEPT network events, network event recording and event object signaling are *level-triggered*. This means that if the re-enabling routine is called, and the relevant network condition is still valid after the call, then the network event is recorded and the associated event object is signaled. This allows a Windows Sockets SPI client to be event-driven while also being unconcerned with the amount of data that arrives at any one time. Consider the following sequence.

1. The service provider receives 100 bytes of data on socket *s*, records the FD_READ network event, and signals the associated event object.
2. The Windows Sockets SPI client issues `WSPRecv(s, buffptr, 50, 0)` to read 50 bytes.
3. The service provider records the FD_READ network event, and signals the associated event object again since there is still data to be read.

With these semantics, a Windows Sockets SPI client need not read all available data in response to an FD_READ network event. Rather, a single LPWSPRecv in response to each FD_READ network event is appropriate.

The FD_QOS and FD_GROUP_QOS events are considered *edge-triggered*. A message will be posted exactly once when a quality of service (QOS) change occurs. Further indications won't be issued until either the service provider detects a further change in QOS, or the Windows Sockets SPI client renegotiates the QOS for the socket.

The FD_ROUTING_INTERFACE_CHANGE and FD_ADDRESS_LIST_CHANGE events are considered *edge-triggered* as well. A message will be posted exactly once when a change occurs *after* the Windows Sockets SPI client has request the notification by issuing WSAIoctl with SIO_ROUTING_INTERFACE_CHANGE or SIO_ADDRESS_LIST_CHANGE correspondingly. Further messages won't be forthcoming until the Windows Sockets SPI client reissues the IOCTL *and* another change is detected since the IOCTL was issued.

If a network event has already happened when the Windows Sockets SPI client calls **LPWSPEventSelect**, or when the re-enabling function is called, then a network event is recorded and the associated event object is signaled, as appropriate. For example, consider the following sequence.

1. A Windows Sockets SPI client calls LPWSPListen.
2. A connect request is received, but not yet accepted.
3. The Windows Sockets SPI client calls **LPWSPEventSelect** specifying that it is interested in the FD_ACCEPT network event for the socket. The service provider records the FD_ACCEPT network event, and signals the associated event object immediately.

The FD_WRITE network event is handled slightly differently. An FD_WRITE network event is recorded when a socket is first connected with LPWSPConnect or accepted with LPWSPAccept, and then after an LPWSPSend or LPWSPSendTo fails with WSAEWOULDBLOCK and buffer space becomes available. Therefore, a Windows Sockets SPI client can assume that sends are possible starting from the first FD_WRITE network event setting, and lasting until a send returns WSAEWOULDBLOCK. After such a failure, the Windows Sockets SPI client will find that sends are again possible when an FD_WRITE network event is recorded and the associated event object is signaled.

The FD_OOB network event is used only when a socket is configured to receive out-of-band data separately. If the socket is configured to receive out-of-band data in-line, then the out-of-band (expedited) data is treated as normal data, and the Windows Sockets SPI client should register an interest in, and will get, FD_READ network event, not FD_OOB network event. A Windows Sockets SPI client may set or inspect the way in which out-of-band data is to be handled by using LPWSPSetSockOpt or LPWSPGetSockOpt for the SO_OOBINLINE option.

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

The FD_CLOSE network event is recorded when a close indication is received for the virtual circuit corresponding to the socket. In TCP terms, this means that the FD_CLOSE is recorded when the connection goes into the FIN WAIT or CLOSE WAIT states. This results from the remote end performing an LPWSPShutdown on the send side, or an LPWSPCloseSocket.

A service providers should record *only* an FD_CLOSE network event to indicate closure of a virtual circuit; it should *not* record an FD_READ network event to indicate that condition.

The FD_QOS or FD_GROUP_QOS network event is recorded when there has been a change to any field in the flow spec associated with socket *s*, or the socket group that *s* belongs to, respectively. This change must be made available to Windows Sockets SPI clients via the LPWSPIoctl function with SIO_GET_QOS and/or SIO_GET_GROUP_QOS to retrieve the current QOS for socket *s*, or for the socket group *s* belongs to, respectively.

The FD_ROUTING_INTERFACE_CHANGE network event is recorded when the local interface that should be used to reach the destination specified in WSAIoctl with SIO_ROUTING_INTERFACE_CHANGE changes *after* such IOCTL has been issued.

The FD_ADDRESS_LIST_CHANGE network event is recorded when the list of addresses of sockets' protocol family to which the Windows Sockets SPI client can bind changes *after* WSAIoctl with SIO_ADDRESS_LIST_CHANGE has been issued.

Requirements

Minimum supported client	Windows 10 Build 20348
Minimum supported server	Windows 10 Build 20348
Header	ws2spi.h

See also

LPWSPEnumNetworkEvents

LPWSPGETOVERLAPPEDRESULT callback function (ws2spi.h)

10/22/2022 • 4 minutes to read • Edit Online

The **LPWSPGetOverlappedResult** function returns the results of an overlapped operation on the specified socket.

Syntax

```
LPWSPGETOVERLAPPEDRESULT Lpwspgetoverlappedresult;

BOOL Lpwspgetoverlappedresult(
  [in]  SOCKET s,
  [in]  LPWSAOVERLAPPED lpOverlapped,
  [out] LPDWORD lpcbTransfer,
  [in]  BOOL fWait,
  [out] LPDWORD lpdwFlags,
  [out] LPINT lpErrno
)
{...}
```

Parameters

`[in] s`

Identifies the socket. This is the same socket that was specified when the overlapped operation was started by a call to **LPWSPRecv**, **LPWSPRecvFrom**, **LPWSPSend**, **LPWSPSendTo**, or **LPWSPIoctl**.

`[in] lpOverlapped`

Pointer to a **WSAOverlapped** structure that was specified when the overlapped operation was started.

`[out] lpcbTransfer`

Pointer to a 32-bit variable that receives the number of bytes that were actually transferred by a send or receive operation, or by **LPWSPIoctl**.

`[in] fWait`

Specifies whether the function should wait for the pending overlapped operation to complete. If **TRUE**, the function does not return until the operation has been completed. If **FALSE** and the operation is still pending, the function returns **FALSE** and *lpErrno* is WSA_IO_INCOMPLETE. The *fWait* parameter may be set to **TRUE** only if the overlapped operation selected event-based completion notification.

`[out] lpdwFlags`

Pointer to a 32-bit variable that will receive one or more flags that supplement the completion status. If the overlapped operation was initiated through **LPWSPRecv** or **LPWSPRecvFrom**, this parameter will contain the results value for *lpFlags* parameter.

`[out] lpErrno`

Pointer to the error code.

Return value

If **LPWSPGetOverlappedResult** succeeds, the return value is **TRUE**. This means the overlapped operation has completed successfully and the value pointed to by *lpcbTransfer* has been updated. If **LPWSPGetOverlappedResult** returns **FALSE**, this means that the overlapped operation has not completed or the overlapped operation completed but with errors, or completion status could not be determined due to errors in one or more parameters to **LPWSPGetOverlappedResult**. On failure, the value pointed to by *lpcbTransfer* will not be updated. The *lpErrno* parameter indicates the cause of the failure (either of **LPWSPGetOverlappedResult** or of the associated overlapped operation).

ERROR CODE	MEANING
WSAENETDOWN	The network subsystem has failed.
WSAENOTSOCK	The descriptor is not a socket.
WSA_INVALID_HANDLE	The **hEvent** member of the **WSAOverlapped** structure does not contain a valid event object handle.
WSAEINVAL	One of the parameters is unacceptable.
WSA_IO_INCOMPLETE	The *fWait* parameter is **FALSE** and the I/O operation has not yet completed.

Remarks

The results reported by the **LPWSPGetOverlappedResult** function are those of the specified socket's last overlapped operation to which the specified **WSAOverlapped** structure was provided, and for which the operation's results were pending. A pending operation is indicated when the function that started the operation returns SOCKET_ERROR, and the *lpErrno* is WSA_IO_PENDING. When an I/O operation is pending, the function that started the operation resets the **hEvent** member of the **WSAOVERLAPPED** structure to the nonsignaled state. Then, when the pending operation has been completed, the system sets the event object to the signaled state.

If the *fWait* parameter is **TRUE**, **LPWSPGetOverlappedResult** determines whether the pending operation has been completed by blocking and waiting for the event object to be in the signaled state. A client may set the *fWait* parameter to **TRUE** only if it selected event-based completion notification when the I/O operation was requested. If another form of notification was selected, the usage of the **hEvent** member of the **WSAOverlapped** structure is different, and setting *fWait* to **TRUE** causes unpredictable results.

> **NOTE**
>
> All I/O initiated by a given thread is canceled when that thread exits. For overlapped sockets, pending asynchronous operations can fail if the thread is closed before the operations complete. See **ExitThread** for more information.

Interaction with WPUCompleteOverlappedRequest

The behavior of **WPUCompleteOverlappedRequest** places some constraints on how a service provider

implements **LPWSPGetOverlappedResult** since only the **Offset** and **OffsetHigh** members of the **WSAOverlapped** structure are exclusively controlled by the service provider even though three values (byte count, flags, and error) must be retrieved from the structure by **LPWSPGetOverlappedResult**. A service provider may accomplish this any way it chooses as long as it interacts with the behavior of **WPUCompleteOverlappedRequest** properly. The following description presents a typical implementation:

At the start of overlapped processing, the service provider sets **Internal** to WSS_OPERATION_IN_PROGRESS.

When the I/O operation is complete, the provider sets **OffsetHigh** to the Windows Sockets 2 error code resulting from the operation, sets **Offset** to the flags resulting from the I/O operation, and calls **WPUCompleteOverlappedRequest**, passing the transfer byte count as one of the parameters. **WPUCompleteOverlappedRequest** eventually sets **InternalHigh** to the transfer byte count, then sets **Internal** to a value other than WSS_OPERATION_IN_PROGRESS.

When **LPWSPGetOverlappedResult** is called, the service provider checks Internal. If it is WSS_OPERATION_IN_PROGRESS, the provider waits on the event handle in the **hEvent** member or returns an error, based on the setting of the *fWait* flag of **LPWSPGetOverlappedResult**. If not in progress, or after completion of waiting, the provider returns the values from **InternalHigh**, **OffsetHigh**, and **Offset** as the transfer count, operation result error code, and flags respectively.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	ws2spi.h

See also

WPUCompleteOverlappedRequest

LPWSPAccept

LPWSPConnect

LPWSPIoctl

LPWSPRecv

LPWSPRecvFrom

LPWSPSend

LPWSPSendTo

LPWSPGETPEERNAME callback function (ws2spi.h)

10/22/2022 • 2 minutes to read • Edit Online

The **LPWSPGetPeerName** function gets the address of the peer to which a socket is connected.

Syntax

```
LPWSPGETPEERNAME Lpwspgetpeername;

int Lpwspgetpeername(
  [in]      SOCKET s,
  [out]     sockaddr *name,
  [in, out] LPINT namelen,
  [out]     LPINT lpErrno
)
{...}
```

Parameters

`[in] s`

Descriptor identifying a connected socket.

`[out] name`

Pointer to the **sockaddr** structure to receive the name of the peer.

`[in, out] namelen`

On input, pointer to an integer that indicates the size of the structure pointed to by *name*, in bytes. On output, indicates the size of the returned name, in bytes.

`[out] lpErrno`

Pointer to the error code.

Return value

If no error occurs, **LPWSPGetPeerName** returns zero. Otherwise, a value of SOCKET_ERROR is returned, and a specific error code is available in *lpErrno*.

ERROR CODE	MEANING
WSAENETDOWN	The network subsystem has failed.
WSAEFAULT	The *name* or the *namelen* parameter is not a valid part of the user address space, or the *namelen* parameter is too small.
WSAEINPROGRESS	Function is invoked when a callback is in progress.

WSAENOTSOCK	The descriptor is not a socket.
WSAENOTCONN	Socket is not connected.
WSAENOTSOCK	The descriptor is not a socket.

Remarks

The **LPWSPGetPeerName** function supplies the name of the peer connected to the socket *s* and stores it in the structure **sockaddr** referenced by *name*. It can be used only on a connected socket. For datagram sockets, only the name of a peer specified in a previous **LPWSPConnect** call will be returned and any name specified by a previous **LPWSPSendTo** call will not be returned by **LPWSPGetPeerName**.

On return, the *namelen* parameter contains the actual size of the name returned in bytes.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	ws2spi.h

See also

LPWSPBind

LPWSPGetSockName

LPWSPSocket

LPWSPGETQOSBYNAME callback function (ws2spi.h)

10/22/2022 • 2 minutes to read • Edit Online

The **WSPGetQOSByName** function initializes a QOS structure based on a named template, or retrieves an enumeration of the available template names.

Syntax

```
LPWSPGETQOSBYNAME Lpwspgetqosbyname;

BOOL Lpwspgetqosbyname(
  [in]      SOCKET s,
  [in, out] LPWSABUF lpQOSName,
  [out]     LPQOS lpQOS,
  [out]     LPINT lpErrno
)
{...}
```

Parameters

`[in] s`

Descriptor identifying a socket.

`[in, out] lpQOSName`

Specifies the QOS template name, or supplies a buffer to retrieve an enumeration of the available template names.

`[out] lpQOS`

Pointer to the QOS structure to be filled.

`[out] lpErrno`

Pointer to the error code.

Return value

If the function succeeds, the return value is **TRUE**. If the function fails, the return value is **FALSE**, and a specific error code is available in *lpErrno*.

ERROR CODE	MEANING
WSAENETDOWN	The network subsystem has failed.
WSAENOTSOCK	The descriptor is not a socket.

WSAENOTSOCK	The *lpQOS* argument is not a valid part of the user address space, or the buffer length for *lpQOS* is too small.
WSAEINVAL	The specified QOS template name is invalid.

Remarks

Clients can use *WSPGetQOSByName* to initialize a QOS structure to a set of known values appropriate for a particular service class or media type. These values are stored in a template that is referenced by a well-known name. The client may retrieve these values by setting the **buf** member of the **WSABUF** indicated by *lpQOSName* to point to a Unicode string of nonzero length specifying a template name. In this case the usage of *lpQOSName* is IN only, and results are returned through *lpQOS*.

Alternatively, the client may use **LPWSPGetQOSByName** to retrieve an enumeration of available template names. The client may do this by setting the **buf** member of the **WSABUF** indicated by *lpQOSName* to a zero-length null-terminated Unicode string. In this case, the buffer indicated by **buf** is overwritten with a sequence of as many null-terminated Unicode template name strings as are available up to the number of bytes available in **buf** as indicated by the **len** member of the **WSABUF** indicated by *lpQOSName*. The list of names itself is terminated by a zero-length Unicode name string. When **LPWSPGetQOSByName** is used to retrieve template names, the *lpQOS* parameter is ignored.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	ws2spi.h

See also

LPWSPAccept

LPWSPConnect

LPWSPGetSockopt

LPWSPGETSOCKNAME callback function (ws2spi.h)

10/22/2022 • 2 minutes to read • Edit Online

The **LPWSPGetSockName** function gets the local name for a socket.

Syntax

```
LPWSPGETSOCKNAME Lpwspgetsockname;

int Lpwspgetsockname(
  [in]      SOCKET s,
  [out]     sockaddr *name,
  [in, out] LPINT namelen,
  [out]     LPINT lpErrno
)
{...}
```

Parameters

`[in] s`

Descriptor identifying a bound socket.

`[out] name`

Pointer to a **sockaddr** structure used to supply the address (name) of the socket.

`[in, out] namelen`

On input, pointer to an integer that indicates the size of the structure pointed to by *name*, in bytes. On output indicates the size of the returned name, in bytes.

`[out] lpErrno`

Pointer to the error code.

Return value

If no error occurs, **LPWSPGetSockName** returns zero. Otherwise, a value of SOCKET_ERROR is returned, and a specific error code is available in *lpErrno*.

ERROR CODE	MEANING
WSAENETDOWN	The network subsystem has failed.
WSAEFAULT	The *name* or the *namelen* parameter is not a valid part of the user address space, or the *namelen* parameter is too small.
WSAEINPROGRESS	Function is invoked when a callback is in progress.

WSAENOTSOCK	The descriptor is not a socket.
WSAEINVAL	Socket has not been bound to an address with LPWSPBind, or ADDR_ANY is specified in **LPWSPBind** but connection has not yet occurred.

Remarks

LPWSPGetSockName retrieves the current name for the specified socket descriptor in *name*. It is used on a bound and/or connected socket specified by the *s* parameter. The local association is returned. This call is especially useful when a LPWSPConnect call has been made without doing a LPWSPBind first; as this call provides the only means by which the local association that has been set by the service provider can be determined.

If a socket was bound to an unspecified address (for example, ADDR_ANY), indicating that any of the host's addresses within the specified address family should be used for the socket, **LPWSPGetSockName** will *not* necessarily return information about the host address, unless the socket has been connected with LPWSPConnect or LPWSPAccept. The Windows Sockets SPI client must not assume that an address will be specified unless the socket is connected. This is because for a multihomed host, the address that will be used for the socket is unknown until the socket is connected.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	ws2spi.h

See also

LPWSPBind

LPWSPGetPeerName

LPWSPSocket

LPWSPGETSOCKOPT callback function (ws2spi.h)

10/22/2022 • 6 minutes to read • Edit Online

The **LPWSPGetSockOpt** function retrieves a socket option.

Syntax

```
LPWSPGETSOCKOPT Lpwspgetsockopt;

int Lpwspgetsockopt(
  SOCKET s,
  int level,
  int optname,
  char *optval,
  LPINT optlen,
  LPINT lpErrno
)
{...}
```

Parameters

`s`

A descriptor identifying a socket.

`level`

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

`optname`

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

`optval`

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

`optlen`

A pointer to the size, in bytes, of the *optval* buffer.

`lpErrno`

A pointer to the error code.

Return value

If no error occurs, **LPWSPGetSockOpt** returns zero. Otherwise, a value of SOCKET_ERROR is returned, and a specific error code is available in *lpErrno*.

ERROR CODE	MEANING

WSAENETDOWN	The network subsystem has failed.
WSAEFAULT	One of the *optval* or the *optlen* parameters is not a valid part of the user address space, or the *optlen* parameter is too small.
WSAEINVAL	The *level* is unknown or invalid.
WSAEINPROGRESS	Function is invoked when a callback is in progress.
WSAENOPROTOOPT	Option is unknown or unsupported by the indicated protocol family.
WSAENOTSOCK	The descriptor is not a socket.

Remarks

The **LPWSPGetSockOpt** function retrieves the current value for a socket option associated with a socket of any type, in any state, and stores the result in *optval*. Options can exist at multiple protocol levels, but they are always present at the uppermost socket' level. Options affect socket operations, such as the routing of packets and OOB data transfer.

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

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

If the option was never set with **LPWSPSetSockOpt**, then **LPWSPGetSockOpt** returns the default value for the option.

For more information on socket options, see **Socket Options**.

level = SOL_SOCKET

VALUE	TYPE	MEANING	DEFAULT
SO_ACCEPTCONN	BOOL	The socket is listening through **LPWSPListen**.	**FALSE** unless a **LPWSPListen** has been performed.
SO_BROADCAST	BOOL	The socket is configured for the transmission and receipt of broadcast messages.	**FALSE**
SO_DEBUG	BOOL	Debugging is enabled.	**FALSE**
SO_DONTLINGER	BOOL	If true, the SO_LINGER option is disabled.	**TRUE**

VALUE	TYPE	MEANING	DEFAULT
SO_DONTROUTE	BOOL	Routing is disabled. Setting this socket option succeeds but is ignored on AF_INET sockets; fails on AF_INET6 sockets with WSAENOPROTOOPT . This option is not supported on ATM sockets (results in an error).	**FALSE**
SO_ERROR	integer	Retrieves error status and clears.	0
SO_GROUP_ID	GROUP	Reserved.	Null
SO_GROUP_PRIORITY	integer	Reserved.	0
SO_KEEPALIVE	BOOL	Keepalives are being sent. Not supported on ATM sockets (results in an error).	**FALSE**
SO_LINGER	**LINGER** structure	Returns the current linger options.	1 is on (default), 0 is off
SO_MAX_MSG_SIZE	unsigned integer	The maximum size of a message for message-oriented socket types (for example, SOCK_DGRAM). Has no meaning for stream oriented sockets.	Implementation dependent
SO_OOBINLINE	BOOL	OOB data is being received in the normal data stream.	**FALSE**
SO_PROTOCOL_INFO	**WSAPROTOCOL_INFO** structure	A description of the protocol information for the protocol that is bound to this socket.	Protocol dependent
SO_RCVBUF	integer	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
SO_REUSEADDR	BOOL	The socket can be bound to an address that is already in use. This option is not applicable on ATM sockets.	**FALSE**.

VALUE	TYPE	MEANING	DEFAULT
SO_SNDBUF	integer	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
SO_TYPE	integer	The type of socket (for example, SOCK_STREAM).	As created with LPWSPSocket">**LPWSPSocket**
PVD_CONFIG	Service Provider Dependent	An opaque data structure object from the service provider associated with socket *s*. This object stores the current configuration information of the service provider. The exact format of this data structure is service provider-specific.	Implementation dependent

Calling **LPWSPGetSockOpt** with an unsupported option will result in an error code of WSAENOPROTOOPT being returned in *lpErrno*.

SO_DEBUG

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

SO_ERROR

The SO_ERROR option returns and resets the per-socket-based error code (that is not necessarily the same as the per-thread-error code that is maintained by the WS2_32.DLL). A successful Windows Sockets call on the socket does not reset the socket-based error code returned by the SO_ERROR option.

SO_GROUP_ID

Reserved. This value should be **NULL**.

SO_GROUP_PRIORITY

Reserved.

SO_KEEPALIVE

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

SO_LINGER

SO_LINGER controls the action taken when unsent data is queued on a socket and a **LPWSPCloseSocket** is performed. See **LPWSPCloseSocket** for a description of the way in which the SO_LINGER settings affect the semantics of **LPWSPCloseSocket**. The Windows Sockets SPI client obtains the desired behavior by creating a **LINGER** structure (pointed to by the *optval* parameter) with the following elements:

```
}
```

SO_MAX_MSG_SIZE

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

SO_PROTOCOL_INFOW

This is a get-only option that supplies the **WSAPROTOCOL_INFO** structure associated with this socket. See **WSCEnumProtocols** for more information about this structure.

SO_SNDBUF

When a Windows Sockets service provider supports the SO_RCVBUF and SO_SNDBUF options, a Windows Sockets SPI client can use **LPWSPSetSockOpt** to request different buffer sizes (larger or smaller). The call can succeed even though the service provider did not make available the entire amount requested. A Windows Sockets SPI client must call this function with the same option to check the buffer size actually provided.

SO_REUSEADDR

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

PVD_CONFIG

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

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	ws2spi.h

See also

LPWSPSetSockOpt

LPWSPSocket

LPWSPIOCTL callback function (ws2spi.h)

10/22/2022 • 35 minutes to read • Edit Online

The **LPWSPIoctl** function controls the mode of a socket.

Syntax

```
LPWSPIOCTL Lpwspioctl;

int Lpwspioctl(
  [in]  SOCKET s,
  [in]  DWORD dwIoControlCode,
  [in]  LPVOID lpvInBuffer,
  [in]  DWORD cbInBuffer,
  [out] LPVOID lpvOutBuffer,
  [in]  DWORD cbOutBuffer,
  [out] LPDWORD lpcbBytesReturned,
  [in]  LPWSAOVERLAPPED lpOverlapped,
  [in]  LPWSAOVERLAPPED_COMPLETION_ROUTINE lpCompletionRoutine,
  [in]  LPWSATHREADID lpThreadId,
  [in]  LPINT lpErrno
)
{...}
```

Parameters

`[in] s`

A descriptor identifying a socket.

`[in] dwIoControlCode`

The control code of the operation to perform.

`[in] lpvInBuffer`

A pointer to the input buffer.

`[in] cbInBuffer`

The size, in bytes, of the input buffer.

`[out] lpvOutBuffer`

A pointer to the output buffer.

`[in] cbOutBuffer`

The size, in bytes, of the output buffer.

`[out] lpcbBytesReturned`

A pointer to actual number of bytes of output.

`[in] lpOverlapped`

A pointer to a **WSAOverlapped** structure (ignored for non-overlapped sockets).

`[in] 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). See Remarks.

`[in] 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) until after the **WPUQueueApc** function returns.

`[in] lpErrno`

A pointer to the error code.

Return value

If no error occurs and the operation has completed immediately, **LPWSPIoctl** returns zero. Note that in this case the completion routine, if specified, will have already been queued. Otherwise, a value of SOCKET_ERROR is returned, and a specific error code is available in *lpErrno*. The error code WSA_IO_PENDING indicates that an overlapped operation has been successfully initiated and that completion will be indicated at a later time. Any other error code indicates that no overlapped operation was initiated and no completion indication will occur.

ERROR CODE	MEANING
WSA_IO_PENDING	An overlapped operation was successfully initiated and completion will be indicated at a later time.
WSAEFAULT	The *lpvInBuffer*, *lpvOutBuffer* or *lpcbBytesReturned* parameter is not totally contained in a valid part of the user address space, or the *cbInBuffer* or *cbOutBuffer* parameter is too small.
WSAEINVAL	The *dwIoControlCode* is not a valid command, or a supplied input parameter is not acceptable, or the command is not applicable to the type of socket supplied.
WSAEINPROGRESS	The function is invoked when a callback is in progress.
WSAENETDOWN	The network subsystem has failed.
WSAENOTSOCK	The descriptor *s* is not a socket.
WSAEOPNOTSUPP	The specified IOCTL command cannot be realized. For example, the flow specifications specified in **SIO_SET_QOS** cannot be satisfied.
WSAEWOULDBLOCK	The socket is marked as nonblocking and the requested operation would block.

Remarks

This routine is used to set or retrieve operating parameters associated with the socket, the transport protocol, or the communications subsystem. If both *lpOverlapped* and *lpCompletionRoutine* are **NULL**, the socket in this function will be treated as a nonoverlapped socket.

For non-overlapped sockets, *lpOverlapped* and *lpCompletionRoutine* parameters are ignored and this function can block if socket *s* is in blocking mode. Note that if socket *s* is in nonblocking mode, this function can return WSAEWOULDBLOCK if the specified operation cannot be finished immediately. In this case, the Windows Sockets SPI client may change the socket to blocking mode and reissue the request or wait for the corresponding network event (such as FD_ROUTING_INTERFACE_CHANGE or FD_ADDRESS_LIST_CHANGE in case of **SIO_ROUTING_INTERFACE_CHANGE** or **SIO_ADDRESS_LIST_CHANGE**) using Windows message (through **LPWSPAsyncSelect** or event (using **LPWSPEventSelect**) based notification mechanism.

For overlapped sockets, operations that cannot be completed immediately will be initiated, and completion will be indicated at a later time. The **DWORD** value pointed to by the *lpcbBytesReturned* parameter that is returned may be ignored. The final completion status and bytes returned can be retrieved when the appropriate completion method is signaled when the operation has completed.

Any IOCTL may block indefinitely, depending on the implementation of the service provider. If the Windows Sockets SPI client cannot tolerate blocking in a **LPWSPIoctl** call, overlapped I/O would be advised for IOCTLs that are most likely to block including:

- **SIO_ADDRESS_LIST_CHANGE**
- **SIO_FINDROUTE**
- **SIO_FLUSH**
- **SIO_GET_QOS**
- **SIO_GET_GROUP_QOS**
- **SIO_ROUTING_INTERFACE_CHANGE**
- **SIO_SET_QOS**
- **SIO_SET_GROUP_QOS**

Some protocol-specific IOCTLs may also be particularly likely to block. Check the relevant protocol-specific annex for available information.

The prototype for the completion routine pointed to by the *lpCompletionRoutine* parameter is as follows.

```
void CALLBACK
CompletionRoutine(
  IN DWORD           dwError,
  IN DWORD           cbTransferred,
  IN LPWSAOVERLAPPED lpOverlapped,
  IN DWORD           dwFlags
);
```

The CompletionRoutine is a placeholder for an application-supplied function name. The *dwError* parameter specifies the completion status for the overlapped operation as indicated by *lpOverlapped* parameter. The *cbTransferred* parameter specifies the number of bytes received. The *dwFlags* parameter is not used for this IOCTL. The completion routine does not return a value.

In as much as the *dwIoControlCode* parameter is now a 32-bit entity, it is possible to adopt an encoding scheme that provides a convenient way to partition the opcode identifier space. The *dwIoControlCode* parameter is constructed to allow for protocol and vendor independence when adding new control codes, while retaining backward compatibility with Windows Sockets 1.1 and UNIX control codes. The *dwIoControlCode* parameter has the following form.

BIT 31	BIT 30	BIT 29	BITS 28 AND 27	BITS 26 THRU 16	BITS 15 THRU 0
I	O	V	T	Vendor/Address family	Code

I is set if the input buffer is valid for the code, as with **IOC_IN**.

O is set if the output buffer is valid for the code, as with **IOC_OUT**. Note that for codes with both input and output parameters, both **I** and **O** will be set.

V is set if there are no parameters for the code, as with **IOC_VOID**.

T is a two-bit quantity that defines the type of IOCTL. The following values are defined.

- **0** indicates that the IOCTL is a standard UNIX IOCTL code, as with **FIONREAD**, **FIONBIO**, and so on.
- **1** indicates that the IOCTL is a generic Windows Sockets 2 IOCTL code. New IOCTL codes defined for Windows Sockets 2 will have **T** == **1**.
- **2** indicates that the IOCTL applies only to a specific address family.
- **3** The IOCTL applies only to a specific vendor's provider. This type allows companies to be assigned a vendor number that appears in the **Vendor/Address family** member. 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 is 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 member has the same value as the code on UNIX. If this is a generic Windows Sockets 2 IOCTL (**T** == **1**) then this member can be used as an extension of the code member to provide additional code values.

Code is the specific IOCTL code for the operation.

The following UNIX commands are supported:

FIONBIO

Enables or disables nonblocking mode on socket *s*. The *lpvInBuffer* parameter points at an unsigned long, which is nonzero if nonblocking mode is to be enabled and zero if it is to be disabled. When a socket is created, it operates in blocking mode (that is, nonblocking mode is disabled). This is consistent with Berkeley Software Distribution (BSD) sockets.

The **LPWSPAsyncSelect** or **LPWSPEventSelect** routine automatically sets a socket to nonblocking mode. If **LPWSPAsyncSelect** or **LPWSPEventSelect** has been issued on a socket, then any attempt to use **LPWSPIoctl** to set the socket back to blocking mode will fail with WSAEINVAL. To set the socket back to blocking mode, a Windows Sockets SPI client must first disable **LPWSPAsyncSelect** by calling **LPWSPAsyncSelect** with the *lEvent* parameter equal to zero, or disable **LPWSPEventSelect** by calling **LPWSPEventSelect** with the *lNetworkEvents* parameter equal to zero.

FIONREAD

Determine the amount of data that can be read atomically from socket *s*. The *lpvOutBuffer* parameter points at an **unsigned long** in which **WSAIoctl** 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

Determines 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 Windows Sockets SPI client should use the **SIOCATMARK** operation to determine whether any remains. If there is any normal data preceding the urgent (OOB) data, it will be received in order. (Note that receive operations will never mix OOB and normal data in the same call.) *lpvOutBuffer* points at a **BOOL** in which **LPWSPIoctl** stores the result.

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: T==1)

To receive notification of changes in the list of local transport addresses of the socket's protocol family to which the Windows Sockets SPI client can bind. No output information will be provided upon completion of this IOCTL; the completion merely indicates that the list of available local addresses has changed and should be queried again through **SIO_ADDRESS_LIST_QUERY**.

It is assumed (although not required) that the Windows Sockets SPI client 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 nonblocking socket and without overlapped parameters (*lpOverlapped* and *lpCompletionRoutine* are set to **NULL**), it will complete immediately with error WSAEWOULDBLOCK. The Windows Sockets SPI client can then wait for address list change events through a call to **LPWSPEventSelect** or **LPWSPAsyncSelect** with the 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_ASSOCIATE_HANDLE (opcode setting: I, T==1)

Associates this socket with the specified handle of a companion interface. The input buffer contains the integer value corresponding to the manifest constant for the companion interface (for example, TH_NETDEV and TH_TAPI), followed by a value that is a handle of the specified companion interface, along with any other required information. Refer to the appropriate section in the *Windows Sockets 2 Protocol-Specific Annex* and/or documentation for the particular companion interface for additional details. (These resources may only be available in English.) 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:

- A great deal of additional control over the behavior of a socket.
- Provider-specific controls that do not map to existing Windows Socket functions (or those likely for 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 backward 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 should 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 Software Development Kit (SDK).

SIO_ENABLE_CIRCULAR_QUEUEING (opcode setting: V, T==1)

Indicates to a 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), that 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 **LPWSPSendTo**.

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 **AcceptEx** extension function.
WSAID_CONNECTEX	The **ConnectEx** extension function.
WSAID_DISCONNECTEX	The **DisconnectEx** extension function.
WSAID_GETACCEPTEXSOCKADDRS	The **GetAcceptExSockaddrs** extension function.
WSAID_TRANSMITFILE	The **TransmitFile** extension function.
WSAID_TRANSMITPACKETS	The **TransmitPackets** extension function.
WSAID_WSARECVMSG	The **LPFN_WSARECVMSG (WSARecvMsg)** extension function.
WSAID_WSASENDMSG	The **WSASendMsg** extension function.

SIO_GET_GROUP_QOS (opcode setting: O, T==1)

Reserved.

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 ro receive the list of configured interfaces.

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)

Retrieves the QOS structure associated with the socket. The input buffer is optional. Some protocols (for example, RSVP) allow the input buffer to be used to qualify a **QOS** request. The **QOS** structure will be copied into the output buffer. The output buffer must be sized large enough to be able to contain the full **QOS** structure. The WSAENOPROTOOPT error code is indicated for service providers that do not support quality of service.

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 Service Pack 1 (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.

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 acknowledgment 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_MULTIPOINT_LOOPBACK (opcode setting: I, 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 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 can be crossed. Note that this corresponds to the time-to-live (TTL) parameter in IP multicasting.

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.

```
void CALLBACK
CompletionRoutine(
  IN DWORD           dwError,
  IN DWORD           cbTransferred,
  IN LPWSAOVERLAPPED lpOverlapped,
  IN DWORD           dwFlags
);
```

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_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. In Windows Vista, WFP focuses on scenarios where the host machine is the communication endpoint. In 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_PNP_TARGET_HANDLE (opcode setting: O, T==1)

To obtain the socket descriptor of the next provider in the chain on which the current socket depends in PnP sense. This IOCTL is invoked by the Windows Sockets 2 DLL only on sockets of non-IFS service providers created through **WPUCreateSocketHandle** call. The provider should return in the output buffer the socket handle of the next provider in the chain on which a given socket handle depends in PnP sense (for example, the removal of the device that supports the underlying handle will result in the invalidation of the handle above it in the chain).

If an overlapped operation completes immediately, this function returns a value of zero and the *lpcbBytesReturned* parameter is updated with the number of bytes in the output buffer. If the overlapped operation is successfully initiated and will complete later, this function returns SOCKET_ERROR and indicates error code WSA_IO_PENDING. In this case, *lpcbBytesReturned* is not updated. When the overlapped operation completes, the amount of data in the output buffer is indicated either through the *cbTransferred* parameter in the completion routine (if specified), or through the *lpcbTransfer* parameter in LPWSPGetOverlappedResult.

SIO_RCVALL (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 **WSAIoctl** 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.

TERM	DESCRIPTION
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_RCVALL** reference.

SIO_RCVALL 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 with error WSAEWOULDBLOCK and the Windows Socket SPI client can then wait for routing change events using a call to **LPWSPEventSelect** or **LPWSPAsyncSelect** with the 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 Windows Sockets SPI client 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 Windows Sockets SPI client 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.

The service provider can opt to ignore the information supplied by the Windows Sockets SPI client in the input buffer of the **SIO_ROUTING_INTERFACE_CHANGE** (as though the Windows Sockets SPI client 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) that 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, Windows Socket SPI clients cannot rely on the information returned by **SIO_ROUTING_INTERFACE_QUERY** to be persistent. SPI clients may register for routing change notifications using 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 Windows Socket SPI client 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 WinSock SPI client 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 output buffer is not large enough to contain the interface address, SOCKET_ERROR is returned as the result of this IOCTL and **WSAGetLastError** returns WSAEFAULT. The required size of the output buffer will be returned in *lpcbBytesReturned* in this case. Note the WSAEFAULT error code is also returned if the *lpvInBuffer*, *lpvOutBuffer*, or *lpcbBytesReturned* parameter is not totally contained in a valid part of the user address space.

If the destination address specified in the input buffer cannot be reached 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:

```
} 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 **WSA_COMPATIBILITY_BEHAVIOR_ID**.

TERM	DESCRIPTION
WsaBehaviorReceiveBuffering	When the **TargetOsVersion** member is set to a value for Windows Vista or later, reductions to the TCP receive buffer size on this socket using the **SO_RCVBUF** socket option are allowed even after a TCP connection has been establishment. When the **TargetOsVersion** member is set to a value earlier than Windows Vista, reductions to the TCP receive buffer size on this socket using the **SO_RCVBUF** socket option are not allowed after connection establishment.
WsaBehaviorAutoTuning	When the **TargetOsVersion** 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 **TargetOsVersion** 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 **SO_RCVBUF** 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_QOS (opcode setting: I, T==1)

Associate the supplied **QOS** structure with the socket. No output buffer is required, the **QOS** structure will be obtained from the input buffer. The WSAENOPROTOOPT error code is indicated for service providers that do not support quality of service.

SIO_TRANSLATE_HANDLE (opcode setting: I, O, T==1)

To obtain a corresponding handle for socket *s* that is valid in the context of a companion interface (for example, TH_NETDEV and TH_TAPI). A manifest constant identifying the companion interface along with any other needed parameters are specified in the input buffer. The corresponding handle will be available in the output buffer upon completion of this function. Refer to the appropriate section in the *Windows Sockets 2 Protocol-Specific Annex* and/or documentation for the particular companion interface for additional details. 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 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.

When called with an overlapped socket, the *lpOverlapped* parameter must be valid for the duration of the overlapped operation.

If the *lpCompletionRoutine* parameter is **NULL**, the service provider signals the **hEvent** member of *lpOverlapped* when the overlapped operation completes if it contains a valid event object handle. The Windows Sockets SPI client can use LPWSPGetOverlappedResult to poll or wait on the event object.

If *lpCompletionRoutine* is not **NULL**, the **hEvent** member is ignored and can be used by the Windows Sockets SPI client to pass context information to the completion routine. A client that passes a non-**NULL** *lpCompletionRoutine* and later calls **WSAGetOverlappedResult** for the same overlapped I/O request may not set the *fWait* parameter for that invocation of **WSAGetOverlappedResult** to **TRUE**. In this case, the usage of the **hEvent** member is undefined, and attempting to wait on the **hEvent** member would produce unpredictable results.

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 and process context 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 (supplied to the provider through the *lpThreadId* input parameter), a pointer to an APC function to be invoked, and a 32-bit context value that is subsequently passed to the APC function. Because only a single 32-bit context value is available, the APC function itself cannot be the client specified–completion routine. The service provider must instead supply a pointer to its own APC function that uses the supplied context value to access the needed result information for the overlapped operation, and then invokes the client specified–completion routine.

The prototype for the client-supplied completion routine is as follows:

```
);
```

CompletionRoutine is a placeholder for a client supplied function. The *dwError* specifies the completion status for the overlapped operation as indicated by *lpOverlapped*. The *cbTransferred* specifies the number of bytes returned. Currently, there are no flag values defined and *dwFlags* will be zero. This function does not return a value.

Returning from this function allows invocation of another pending completion routine for this socket. The completion routines can be called in any order, though not necessarily in the same order that the overlapped operations are completed.

Compatibility

The IOCTL codes with T == 0 are a subset of the IOCTL codes used in Berkeley sockets. In particular, there is no command that is equivalent to FIOASYNC.

> **NOTE**
>
> All I/O initiated by a given thread is canceled when that thread exits. For overlapped sockets, pending asynchronous operations can fail if the thread is closed before the operations complete. See **ExitThread** for more information.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	ws2spi.h

See also

WPUQueueApc

LPWSPGetSockopt

LPWSPSetSockOpt

LPWSPSocket

LPWSPJOINLEAF callback function (ws2spi.h)

10/22/2022 • 8 minutes to read • Edit Online

The **WSPJoinLeaf** function joins a leaf node into a multipoint session, exchanges connect data, and specifies needed quality of service based on the supplied flow specifications.

Syntax

```
LPWSPJOINLEAF Lpwspjoinleaf;

SOCKET Lpwspjoinleaf(
  [in]  SOCKET s,
  [in]  const sockaddr *name,
  [in]  int namelen,
  [in]  LPWSABUF lpCallerData,
  [out] LPWSABUF lpCalleeData,
  [in]  LPQOS lpSQOS,
  [in]  LPQOS lpGQOS,
  [in]  DWORD dwFlags,
  [out] LPINT lpErrno
)
{...}
```

Parameters

`[in] s`

Descriptor identifying a multipoint socket.

`[in] name`

Name of the peer to which the socket in the sockaddr structure is to be joined.

`[in] namelen`

Length of the *name*, in bytes.

`[in] lpCallerData`

Pointer to the user data that is to be transferred to the peer during multipoint session establishment.

`[out] lpCalleeData`

Pointer to the user data that is to be transferred back from the peer during multipoint session establishment.

`[in] lpSQOS`

Pointer to the flow specifications for socket *s*, one for each direction.

`[in] lpGQOS`

Reserved.

`[in] dwFlags`

Flags to indicate that the socket is acting as a sender, receiver, or both.

`[out] lpErrno`

Pointer to the error code.

Return value

If no error occurs, **WSPJoinLeaf** returns a value of type **SOCKET** that is a descriptor for the newly created multipoint socket. Otherwise, a value of INVALID_SOCKET is returned, and a specific error code is available in *lpErrno*.

On a blocking socket, the return value indicates success or failure of the join operation.

With a nonblocking socket, successful initiation of a join operation is indicated by a return value of a valid socket descriptor. Subsequently, an FD_CONNECT indication is given when the join operation completes, either successfully or otherwise. The error code associated with the FD_CONNECT indicates the success or failure of the **WSPJoinLeaf**.

Also, until the multipoint session join attempt completes all subsequent calls to **WSPJoinLeaf** on the same socket will fail with the error code WSAEALREADY. After the **WSPJoinLeaf** completes successfully a subsequent attempt will usually fail with the error code WSAEISCONN. An exception to the WSAEISCONN rule occurs for a c_root socket that allows root-initiated joins. In such a case another join may be initiated after a prior **WSPJoinLeaf** completes.

If the return error code indicates the multipoint session join attempt failed (that is, WSAECONNREFUSED, WSAENETUNREACH, WSAETIMEDOUT) the Windows Sockets SPI client can call **WSPJoinLeaf** again for the same socket.

ERROR CODE	MEANING
WSAENETDOWN	The network subsystem has failed.
WSAEADDRINUSE	Socket's local address is already in use and the socket was not marked to allow address reuse with SO_REUSEADDR. This error usually occurs at the time of bind, but could be delayed until this function if the **bind** was to a partially wild-card address (involving ADDR_ANY) and if a specific address needs to be "committed" at the time of this function.
WSAEINTR	(Blocking) call was canceled through WSPCancelBlockingCall.
WSAEINPROGRESS	Blocking Windows Sockets call is in progress, or the service provider is still processing a callback function.
WSAEALREADY	Nonblocking WSPJoinLeaf call is in progress on the specified socket.
WSAEADDRNOTAVAIL	Remote address is not a valid address (for example, ADDR_ANY).
WSAEAFNOSUPPORT	Addresses in the specified family cannot be used with this socket.

WSAECONNREFUSED	The attempt to join was forcefully rejected.
WSAEFAULT	The *name* or the *namelen* parameter is not a valid part of the user address space, the *namelen* parameter is too small, the buffer length for *lpCalleeData*, *lpSQOS*, and *lpGQOS* are too small, or the buffer length for *lpCallerData* is too large.
WSAEISCONN	Socket is already member of the multipoint session.
WSAENETUNREACH	The network cannot be reached from this host at this time.
WSAENOBUFS	No buffer space is available. The socket cannot be joined.
WSAENOTSOCK	The descriptor is not a socket.
WSAEOPNOTSUPP	Flow specifications specified in *lpSQOS* cannot be satisfied.
WSAEPROTONOSUPPORT	The *lpCallerData* augment is not supported by the service provider.
WSAETIMEDOUT	An attempt to join timed out without establishing a multipoint session.

Remarks

This function is used to join a leaf node to a multipoint session, and to perform a number of other ancillary operations that occur at session join time as well. If the socket, *s*, is unbound, unique values are assigned to the local association by the system, and the socket is marked as bound.

WSPJoinLeaf has the same parameters and semantics as LPWSPConnect except that it returns a socket descriptor (as in LPWSPAccept), and it has an additional *dwFlags* parameter. Only multipoint sockets created using LPWSPSocket with appropriate multipoint flags set can be used for input parameter *s* in this function. If the socket is in the nonblocking mode, the returned socket descriptor will not be usable until after a corresponding FD_CONNECT indication on the original socket *s* has been received, except that closesocket can be invoked on this new socket descriptor to cancel a pending join operation. A root node in a multipoint session can call **WSPJoinLeaf** one or more times in order to add a number of leaf nodes, however at most one multipoint connection request can be outstanding at a time. Refer to Protocol-Independent Multicast and Multipoint in the SPI for additional information.

For nonblocking sockets it is often not possible to complete the connection immediately. In such a case, this function returns an as-yet unusable socket descriptor and the operation proceeds. There is no error code such as WSAEWOULDBLOCK in this case, since the function has effectively returned a "successful start" indication. When the final outcome success or failure becomes known, it may be reported through LPWSPAsyncSelect or

LPWSPEventSelect depending on how the client registers for notification on the original socket *s*. In either case, the notification is announced with FD_CONNECT and the error code associated with the FD_CONNECT indicates either success or a specific reason for failure. Note that LPWSPSelect cannot be used to detect completion notification for **WSPJoinLeaf**.

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. (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). The newly created socket has the same properties as *s* including asynchronous events registered with LPWSPAsyncSelect or with LPWSPEventSelect. It is not intended to be used for exchange of multipoint data, but rather is used to receive network event indications (for example, FD_CLOSE) for the connection that exists to the particular c_leaf. Some multipoint implementations can also allow this socket to be used for "side chats" between the root and an individual leaf node. An FD_CLOSE indication will be received for this socket if the corresponding leaf node calls LPWSPCloseSocket 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 node (for a rooted control scheme) or an existing multipoint session (nonrooted control scheme), and the returned socket descriptor is the same as the input socket descriptor. In other words, a new socket descriptor is not allocated. In a rooted control scheme, the root application would put its c_root socket in the listening mode by calling LPWSPListen. The standard FD_ACCEPT notification will be delivered when the leaf node requests to join itself to the multipoint session. The root application uses the usual LPWSPAccept functions 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 an input to **WSPJoinLeaf**.

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

The *lpCallerData* is a value parameter that contains any user data that is to be sent along with the multipoint session join request. If *lpCallerData* is **NULL**, no user data will be passed to the peer. The *lpCalleeData* is a result parameter that will contain any user data passed back from the peer as part of the multipoint session establishment. *lpCalleeData*->**len** initially contains the length of the buffer allocated by the Windows Sockets SPI client and pointed to by *lpCalleeData*->**buf**. *lpCalleeData*->**len** will be set to zero if no user data has been passed back. The *lpCalleeData* information will be valid when the multipoint join operation is complete. For blocking sockets, this will be when the **WSPJoinLeaf** function returns. For nonblocking sockets, this will be after the FD_CONNECT notification has occurred on the original socket *s*. If *lpCalleeData* is **NULL**, no user data will be passed back. The exact format of the user data is specific to the address family to which the socket belongs and/or the applications involved.

At multipoint session establishment time, a Windows Sockets SPI client can use the *lpSQOS* parameters to override any previous QoS specification made for the socket through LPWSPIoctl with the SIO_SET_QOS opcode.

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

The *dwFlags* parameter is used to indicate whether the socket will be acting only as a sender

(JL_SENDER_ONLY), only as a receiver (JL_RECEIVER_ONLY), or both (JL_BOTH).

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

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	ws2spi.h

See also

LPWSPAccept

LPWSPAsyncSelect

LPWSPBind

LPWSPEventSelect

LPWSPSelect

LPWSPSocket

LPWSPLISTEN callback function (ws2spi.h)

10/22/2022 • 2 minutes to read • Edit Online

The **LPWSPListen** function establishes a socket to listen for incoming connections.

Syntax

```
LPWSPLISTEN Lpwsplisten;

int Lpwsplisten(
  [in]  SOCKET s,
  [in]  int backlog,
  [out] LPINT lpErrno
)
{...}
```

Parameters

`[in] s`

Descriptor identifying a bound, unconnected socket.

`[in] backlog`

Maximum length to which the queue of pending connections can grow. If this value is SOMAXCONN, then the service provider should set the backlog to a maximum "reasonable" value. There is no standard provision to find out the actual backlog value.

`[out] lpErrno`

Pointer to the error code.

Return value

If no error occurs, **LPWSPListen** returns zero. Otherwise, a value of SOCKET_ERROR is returned, and a specific error code is available in *lpErrno*.

ERROR CODE	MEANING
WSAENETDOWN	The network subsystem has failed.
WSAEADDRINUSE	Socket's local address is already in use and the socket was not marked to allow address reuse with SO_REUSEADDR. This error usually occurs at the time of **Bind**, but could be delayed until this function if the **bind** was to a partially wildcard address (involving ADDR_ANY) and if a specific address needs to be committed at the time of this function.
WSAEINPROGRESS	Function is invoked when a callback is in progress.

WSAEINVAL	Socket has not been bound with **LPWSPBind**.
WSAEISCONN	Socket is already connected.
WSAEMFILE	No more socket descriptors are available.
WSAENOBUFS	No buffer space is available.
WSAENOTSOCK	The descriptor is not a socket.
WSAEOPNOTSUPP	Referenced socket is not of a type that supports the **LPWSPListen** operation.

Remarks

To accept connections, a socket is first created with LPWSPSocket bound to a local address with **LPWSPBind**, a backlog for incoming connections is specified with **LPWSPListen**, and then the connections are accepted with **LPWSPAccept**. **LPWSPListen** applies only to sockets that are connection oriented (for example, SOCK_STREAM). The socket *s* is put into passive mode where incoming connection requests are acknowledged and queued pending acceptance by the Windows Sockets SPI client.

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

LPWSPListen should continue to function rationally when there are no available descriptors. It should accept connections until the queue is emptied. If descriptors become available, a later call to **LPWSPListen** or **LPWSPAccept** will refill the queue to the current or most recent backlog, if possible, and resume listening for incoming connections.

A Windows Sockets SPI client can call **LPWSPListen** more than once on the same socket. This has the effect of updating the current backlog for the listening socket. Should there be more pending connections than the new *backlog* value, the excess pending connections will be reset and dropped.

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

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	ws2spi.h

See also

LPWSPAccept

LPWSPConnect

LPWSPSocket

LPWSPRECV callback function (ws2spi.h)

10/22/2022 • 11 minutes to read • Edit Online

The **LPWSPRecv** function receives data on a socket.

Syntax

```
LPWSPRECV Lpwsprecv;

int Lpwsprecv(
  [in]         SOCKET s,
  \[in\, out\] LPWSABUF lpBuffers,
  [in]         DWORD dwBufferCount,
  [out]        LPDWORD lpNumberOfBytesRecvd,
  \[in\, out\] LPDWORD lpFlags,
  [in]         LPWSAOVERLAPPED lpOverlapped,
  [in]         LPWSAOVERLAPPED_COMPLETION_ROUTINE lpCompletionRoutine,
  [in]         LPWSATHREADID lpThreadId,
  [out]        LPINT lpErrno
)
{...}
```

Parameters

`[in] s`

A descriptor identifying a connected socket.

`\\[in\\, out\\] lpBuffers`

A pointer to an array of WSABUF structures. Each **WSABUF** structure contains a pointer to a buffer and the length of the buffer, in bytes.

`[in] dwBufferCount`

The number of WSABUF structures in the *lpBuffers* array.

`[out] lpNumberOfBytesRecvd`

A pointer to the number of bytes received by this call.

`\\[in\\, out\\] lpFlags`

A pointer to flags that specify the way in which the call is made.

`[in] lpOverlapped`

A pointer to a **WSAOverlapped** structure (ignored for nonoverlapped structures).

`[in] lpCompletionRoutine`

Type: _In_opt_ **LPWSAOVERLAPPED_COMPLETION_ROUTINE**

A pointer to the completion routine called when the receive operation has been completed (ignored for nonoverlapped structures).

`[in] 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.

`[out] lpErrno`

A pointer to the error code.

Return value

If no error occurs and the receive operation has completed immediately, **LPWSPRecv** returns zero. Note that in this case the completion routine, if specified, will have already been queued. Otherwise, a value of SOCKET_ERROR is returned, and a specific error code is available in *lpErrno*. The error code WSA_IO_PENDING indicates that the overlapped operation has been successfully initiated and that completion will be indicated at a later time. Any other error code indicates that no overlapped operations was initiated and no completion indication will occur.

ERROR CODE	MEANING
WSAENETDOWN	The network subsystem has failed.
WSAENOTCONN	Socket is not connected.
WSAEINTR	(Blocking) call was canceled through **LPWSPCancelBlockingCall**.
WSAEINPROGRESS	Blocking Windows Sockets call is in progress, or the service provider is still processing a callback function.
WSAENETRESET	The connection has been broken due to keep-alive activity detecting a failure while the operation was in progress.
WSAEFAULT	The *lpBuffers* parameter is not totally contained in a valid part of the user address space.
WSAENOTSOCK	The descriptor is not a socket.
WSAEOPNOTSUPP	MSG_OOB was specified, but the socket is not stream-style such as type SOCK_STREAM, OOB data is not supported in the communication domain associated with this socket, or the socket is unidirectional and supports only send operations.
WSAESHUTDOWN	Socket has been shut down; it is not possible to receive through **LPWSPRecv** on a socket after **LPWSPShutdown** has been invoked with *how* set to SD_RECEIVE or SD_BOTH.
WSAEWOULDBLOCK	Windows NT: Overlapped sockets: there are too many outstanding overlapped I/O requests. Nonoverlapped sockets: The socket is marked as nonblocking and the receive operation cannot be completed immediately.

WSAEMSGSIZE	Message was too large to fit into the specified buffer and (for unreliable protocols only) any trailing portion of the message that did not fit into the buffer has been discarded.
WSAEINVAL	Socket has not been bound (for example, with **LPWSPBind**) or the socket is not created with the overlapped flag.
WSAECONNABORTED	Virtual circuit was terminated due to a time-out or other failure.
WSAECONNRESET	Virtual circuit was reset by the remote side.
WSAEDISCON	Socket *s* is message oriented and the virtual circuit was gracefully closed by the remote side.
WSA_IO_PENDING	An overlapped operation was successfully initiated and completion will be indicated at a later time.
WSA_OPERATION_ABORTED	Overlapped operation has been canceled due to the closure of the socket.

Remarks

LPWSPRecv is used on connected sockets or bound connectionless sockets specified by the *s* parameter and is used to read incoming data. The socket's local address must be known. This may be done explicitly through **LPWSPBind** or implicitly through **LPWSPAccept**, **LPWSPConnect**, **LPWSPSendTo**, or **LPWSPJoinLeaf**.

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

For overlapped sockets **LPWSPRecv** is used to post one or more buffers into which incoming data will be placed as it becomes available, after which the Windows Sockets SPI client-specified completion indication (invocation of the completion routine or setting of an event object) occurs. If the operation does not complete immediately, the final completion status is retrieved through the completion routine or LPWSPGetOverlappedResult.

If both *lpOverlapped* and *lpCompletionRoutine* are null, the socket in this function will be treated as a nonoverlapped socket.

For nonoverlapped sockets, the *lpOverlapped*, *lpCompletionRoutine*, and *lpThreadId* parameters are ignored. Any data that has already been received and buffered by the transport will be copied into the supplied user buffers. For the case of a blocking socket with no data currently having been received and buffered by the transport, the call will block until data is received. Windows Sockets 2 does not define any standard blocking timeout mechanism for this function. For protocols acting as byte-stream protocols the stack tries to return as much data as possible subject to the supplied buffer space and amount of received data available. However, receipt of a single byte is sufficient to unblock the caller. There is no guarantee that more than a single byte will be returned. For protocols acting as message-oriented, a full message is required to unblock the caller.

Whether or not a protocol is acting as byte-stream is determined by the setting of XP1_MESSAGE_ORIENTED and XP1_PSEUDO_STREAM in its **WSAPROTOCOL_INFO** structure and the setting of the MSG_PARTIAL flag passed in to this function (for protocols that support it). The relevant combinations are summarized in the

following table (an asterisk (*) indicates that the setting of this bit does not matter in this case).

XP1_MESSAGE_ORIENTED	XP1_PSEUDO_STREAM	MSG_PARTIAL	ACTS AS
not set	*	*	byte stream
*	set	*	byte stream
set	not set	set	byte stream
set	not set	not set	message oriented

The supplied buffers are filled in the order in which they appear in the array pointed to by *lpBuffers*, and the buffers are packed so that no holes are created.

The array of WSABUF structures pointed to by the *lpBuffers* parameter is transient. If this operation completes in an overlapped manner, it is the service provider's responsibility to capture this array of pointers to **WSABUF** structures before returning from this call. This enables Windows Sockets SPI clients to build stack-based **WSABUF** arrays.

For byte stream-style sockets (for example, type SOCK_STREAM), incoming data is placed into the buffers until the buffers are filled, the connection is closed, or internally buffered data is exhausted. Regardless of whether or not the incoming data fills all the buffers, the completion indication occurs for overlapped sockets. For message-oriented sockets (for example, type SOCK_DGRAM), an incoming message is placed into the supplied buffers, up to the total size of the buffers supplied, and the completion indication occurs for overlapped sockets. If the message is larger than the buffers supplied, the buffers are filled with the first part of the message. If the MSG_PARTIAL feature is supported by the service provider, the MSG_PARTIAL flag is set in *lpFlags* and subsequent receive operation(s) can be used to retrieve the rest of the message. If MSG_PARTIAL is not supported but the protocol is reliable, **LPWSPRecv** generates the error WSAEMSGSIZE and a subsequent receive operation with a larger buffer can be used to retrieve the entire message. Otherwise, (that is, the protocol is unreliable and does not support MSG_PARTIAL), the excess data is lost, and **LPWSPRecv** generates the error WSAEMSGSIZE.

For connection-oriented sockets, **LPWSPRecv** can indicate the graceful termination of the virtual circuit in one of two ways, depending on whether the socket is a byte stream or message oriented. For byte streams, zero bytes having been read indicates graceful closure and that no more bytes will ever be read. For message-oriented sockets, where a zero byte message is often allowable, a return error code of WSAEDISCON is used to indicate graceful closure. In any case a return error code of WSAECONNRESET indicates an abortive close has occurred.

The *lpFlags* parameter can be used to influence the behavior of the function invocation beyond the options specified for the associated socket. That is, the semantics of this function are determined by the socket options and the *lpFlags* parameter. The latter is constructed by using the bitwise OR operator with any of the following values.

VALUE	MEANING
MSG_PEEK	Peeks at the incoming data. The data is copied into the buffer but is not removed from the input queue. This flag is valid only for nonoverlapped sockets.
MSG_OOB	Processes Out Of Band (OOB) data.

VALUE	MEANING
MSG_PARTIAL	This flag is for message-oriented sockets only. On output, indicates that the data supplied is a portion of the message transmitted by the sender. Remaining portions of the message will be supplied in subsequent receive operations. A subsequent receive operation with MSG_PARTIAL flag cleared indicates end of sender's message. As an input parameter, MSG_PARTIAL indicates that the receive operation should complete even if only part of a message has been received by the service provider.

If an overlapped operation completes immediately, **LPWSPRecv** returns a value of zero and the *lpNumberOfBytesRecvd* parameter is updated with the number of bytes received and the flag bits pointed by the *lpFlags* parameter are also updated. If the overlapped operation is successfully initiated and will complete later, **LPWSPRecv** returns SOCKET_ERROR and indicates error code WSA_IO_PENDING. In this case, *lpNumberOfBytesRecvd* and *lpFlags* are not updated. When the overlapped operation completes the amount of data transferred is indicated either through the *cbTransferred* parameter in the completion routine (if specified), or through the *lpcbTransfer* parameter in LPWSPGetOverlappedResult. Flag values are obtained either through the *dwFlags* parameter of the completion routine, or by examining the *lpdwFlags* parameter of **WSAGetOverlappedResult**.

Providers must allow this function to be called from within the completion routine of a previous **LPWSPRecv**, **LPWSPRecvFrom**, **LPWSPSend** or **LPWSPSendTo** function. However, for a given socket, I/O completion routines cannot be nested. This permits time-sensitive data transmissions to occur entirely within a preemptive context.

The *lpOverlapped* parameter must be valid for the duration of the overlapped operation. If multiple I/O operations are simultaneously outstanding, each must reference a separate overlapped structure. The **WSAOverlapped** structure is defined in its own reference page.

If the *lpCompletionRoutine* parameter is null, the service provider signals the **hEvent** member of *lpOverlapped* when the overlapped operation completes if it contains a valid event object handle. The Windows Sockets SPI client can use LPWSPGetOverlappedResult to wait or poll on the event object.

If *lpCompletionRoutine* is not null, the **hEvent** member is ignored and can be used by the Windows Sockets SPI client to pass context information to the completion routine. A client that passes a null *lpCompletionRoutine* and later calls **WSAGetOverlappedResult** for the same overlapped I/O request may not set the *fWait* parameter for that invocation of **WSAGetOverlappedResult** to **TRUE**. In this case the usage of the **hEvent** member is undefined, and attempting to wait on the **hEvent** member would produce unpredictable results.

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 and process context by calling **WPUQueueApc**, which was used to initiate the overlapped operation. 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 (supplied to the provider through the *lpThreadId* input parameter), a pointer to an APC function to be invoked, and a context value that is subsequently passed to the APC function. Because only a single context value is available, the APC function itself cannot be the client specified–completion routine. The service provider must instead supply a pointer to its own

APC function that uses the supplied context value to access the needed result information for the overlapped operation, and then invokes the client specified–completion routine.

The prototype for the client-supplied completion routine is as follows.

```
void CALLBACK
CompletionRoutine(
  IN DWORD           dwError,
  IN DWORD           cbTransferred,
  IN LPWSAOVERLAPPED lpOverlapped,
  IN DWORD           dwFlags
);
```

The *CompletionRoutine* parameter is a placeholder for a client-supplied function name. *dwError* specifies the completion status for the overlapped operation as indicated by *lpOverlapped*. The *cbTransferred* parameter specifies the number of bytes received. *dwFlags* contains information that would have appeared in *lpFlags* if the receive operation had completed immediately. This function does not return a value.

The completion routines can be called in any order, but not necessarily the same order in which the overlapped operations are completed. However, the posted buffers are guaranteed to be filled in the same order in which they are supplied.

> **NOTE**
> All I/O initiated by a given thread is canceled when that thread exits. For overlapped sockets, pending asynchronous operations can fail if the thread is closed before the operations complete. See **ExitThread** for more information.

Requirements

Minimum supported client	Windows 10 Build 20348
Minimum supported server	Windows 10 Build 20348
Header	ws2spi.h

See also

WPUCloseEvent

WPUCreateEvent

WPUQueueApc

LPWSPGetOverlappedResult

LPWSPSocket

LPWSPRECVDISCONNECT callback function (ws2spi.h)

10/22/2022 • 2 minutes to read • Edit Online

The **LPWSPRecvDisconnect** function terminates reception on a socket and retrieves the disconnect data, if the socket is connection oriented.

Syntax

```
LPWSPRECVDISCONNECT Lpwsprecvdisconnect;

int Lpwsprecvdisconnect(
  [in]  SOCKET s,
  [out] LPWSABUF lpInboundDisconnectData,
  [out] LPINT lpErrno
)
{...}
```

Parameters

`[in] s`

Descriptor identifying a socket.

`[out] lpInboundDisconnectData`

Pointer to a buffer into which disconnect data is to be copied.

`[out] lpErrno`

Pointer to the error code.

Return value

If no error occurs, **LPWSPRecvDisconnect** returns zero. Otherwise, a value of SOCKET_ERROR is returned, and a specific error code is available in *lpErrno*.

ERROR CODE	MEANING
WSAENETDOWN	The network subsystem has failed.
WSAEFAULT	Buffer referenced by the parameter *lpInboundDisconnectData* is too small.
WSAENOPROTOOPT	Disconnect data is not supported by the indicated protocol family.
WSAEINPROGRESS	Blocking Windows Sockets call is in progress, or the service provider is still processing a callback function.

WSAENOTCONN	Socket is not connected (connection-oriented sockets only).
WSAENOTSOCK	The descriptor is not a socket.

Remarks

LPWSPRecvDisconnect is used on connection-oriented sockets to disable reception, and retrieve any incoming disconnect data from the remote party.

After this function has been successfully issued, subsequent receives on the socket will be disallowed. This has no effect on the lower protocol layers. For TCP, the TCP window is not changed and incoming data will be accepted (but not acknowledged) until the window is exhausted. For UDP, incoming datagrams are accepted and queued. In no case will an ICMP error packet be generated.

To successfully receive incoming disconnect data, a Windows Sockets SPI client must use other mechanisms to determine that the circuit has been closed. For example, a client needs to receive an FD_CLOSE notification, or get a zero return value, or a **WSAEDISCON** error code from **LPWSPRecv**.

Note that **LPWSPRecvDisconnect** does not close the socket, and resources attached to the socket will not be freed until **LPWSPCloseSocket** is invoked.

> **NOTE**
>
> **LPWSPRecvDisconnect** does not block regardless of the SO_LINGER setting on the socket. A Windows Sockets SPI client should not rely on being able to reuse a socket after it has been **LPWSPRecvDisconnect**ed. In particular, a Windows Sockets provider is not required to support the use of **LPWSPConnect** on such a socket.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	ws2spi.h

See also

LPWSPConnect

LPWSPSocket

LPWSPRECVFROM callback function (ws2spi.h)

10/22/2022 • 10 minutes to read • Edit Online

The **LPWSPRecvFrom** function receives a datagram and stores the source address.

Syntax

```
LPWSPRECVFROM Lpwsprecvfrom;

int Lpwsprecvfrom(
  [in]      SOCKET s,
  [in, out] LPWSABUF lpBuffers,
  [in]      DWORD dwBufferCount,
  [out]     LPDWORD lpNumberOfBytesRecvd,
  [in, out] LPDWORD lpFlags,
  [out]     sockaddr *lpFrom,
  [in, out] LPINT lpFromlen,
  [in]      LPWSAOVERLAPPED lpOverlapped,
  [in]      LPWSAOVERLAPPED_COMPLETION_ROUTINE lpCompletionRoutine,
  \[in\]    LPWSATHREADID lpThreadId,
  [in, out] LPINT lpErrno
)
{...}
```

Parameters

`[in] s`

Descriptor identifying a socket.

`[in, out] lpBuffers`

Pointer to an array of WSABUF structures. Each **WSABUF** structure contains a pointer to a buffer and the length of the buffer, in bytes.

`[in] dwBufferCount`

Number of WSABUF structures in the *lpBuffers* array.

`[out] lpNumberOfBytesRecvd`

Pointer to the number of bytes received by this call.

`[in, out] lpFlags`

Pointer to flags.

`[out] lpFrom`

Optional pointer to a buffer in the **sockaddr** structure that will hold the source address upon the completion of the overlapped operation.

`[in, out] lpFromlen`

Pointer to the size of the *lpFrom* buffer, in bytes, required only if *lpFrom* is specified.

`[in] lpOverlapped`

Pointer to a **WSAOverlapped** structure (ignored for nonoverlapped sockets).

`[in] lpCompletionRoutine`

Type: _In_opt_ **LPWSAOVERLAPPED_COMPLETION_ROUTINE**

Pointer to the completion routine called when the receive operation has been completed (ignored for nonoverlapped sockets).

`\\[in\\] lpThreadId`

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.

`[in, out] lpErrno`

Pointer to the error code.

Return value

If no error occurs and the receive operation has completed immediately, **LPWSPRecvFrom** returns zero. Note that in this case the completion routine, if specified will have already been queued. Otherwise, a value of SOCKET_ERROR is returned, and a specific error code is available in *lpErrno*. The error code WSA_IO_PENDING indicates that the overlapped operation has been successfully initiated and that completion will be indicated at a later time. Any other error code indicates that no overlapped operations was initiated and no completion indication will occur.

ERROR CODE	MEANING
WSAENETDOWN	The network subsystem has failed.
WSAEFAULT	The *lpFromlen* parameter was invalid: the *lpFrom* buffer was too small to accommodate the peer address or *lpbuffers* is not totally contained within a valid part of the user address space.
WSAEINTR	(Blocking) call was canceled through **LPWSPCancelBlockingCall**.
WSAEINPROGRESS	Blocking Windows Sockets call is in progress, or the service provider is still processing a callback function.
WSAEINVAL	Socket has not been bound (for example, with **LPWSPBind**) or the socket is not created with the overlapped flag.
WSAEISCONN	Socket is connected. This function is not permitted with a connected socket, whether the socket is connection-oriented or connectionless.
WSAENETRESET	The connection has been broken due to keep-alive activity detecting a failure while the operation was in progress.

WSAENOTSOCK	The descriptor is not a socket.
WSAEOPNOTSUPP	MSG_OOB was specified, but the socket is not stream-style such as type SOCK_STREAM, OOB data is not supported in the communication domain associated with this socket, or the socket is unidirectional and supports only send operations.
WSAESHUTDOWN	Socket has been shut down; it is not possible to run **LPWSPRecvFrom** on a socket after **LPWSPShutdown** has been invoked with *how* set to SD_RECEIVE or SD_BOTH.
WSAEWOULDBLOCK	**Windows NT:** Overlapped sockets: There are too many outstanding overlapped I/O requests.Nonoverlapped sockets: The socket is marked as nonblocking and the receive operation cannot be completed immediately.
WSAEMSGSIZE	Message was too large to fit into the specified buffer and (for unreliable protocols only) any trailing portion of the message that did not fit into the buffer has been discarded.
WSAECONNRESET	The virtual circuit was reset by the remote side executing a hard or abortive close. The application should close the socket as it is no longer usable. On a UDP datagram socket this error would indicate that a previous send operation resulted in an ICMP "Port Unreachable" message.
WSAEDISCON	Socket *s* is message oriented and the virtual circuit was gracefully closed by the remote side.
WSA_IO_PENDING	An overlapped operation was successfully initiated and completion will be indicated at a later time.
WSA_OPERATION_ABORTED	Overlapped operation has been canceled due to the closure of the socket.

Remarks

The **LPWSPRecvFrom** function is used primarily on a connectionless socket specified by *s* The socket must not be connected. The local address of the socket must be known. This may be done explicitly through **LPWSPBind** or implicitly through **LPWSPSendTo** or **LPWSPJoinLeaf**.

For overlapped sockets, this function is used to post one or more buffers into which incoming data will be placed as it becomes available on a (possibly connected) socket, after which the client-specified completion indication (invocation of the completion routine or setting of an event object) occurs. If the operation does not complete immediately, the final completion status is retrieved through the completion routine or LPWSPGetOverlappedResult. Also note that the values pointed to by *lpFrom* and *lpFromlen* are not updated until completion is indicated. Applications must not use or disturb these values until they have been updated, therefore the client must not use automatic (that is, stack-based) variables for these parameters.

If both *lpOverlapped* and *lpCompletionRoutine* are null, the socket in this function will be treated as a nonoverlapped socket.

For nonoverlapped sockets, the *lpOverlapped*, *lpCompletionRoutine*, and *lpThreadId* parameters are ignored. Any data that has already been received and buffered by the transport will be copied into the supplied user buffers. For the case of a blocking socket with no data currently having been received and buffered by the transport, the call will block until data is received according to the assigned blocking semantics for **LPWSPRecv**.

The supplied buffers are filled in the order in which they appear in the array pointed to by *lpBuffers*, and the buffers are packed so that no holes are created.

The array of WSABUF structures pointed to by the *lpBuffers* parameter is transient. If this operation completes in an overlapped manner, it is the service provider's responsibility to capture this array of pointers to **WSABUF** structures before returning from this call. This enables Windows Sockets SPI clients to build stack-based **WSABUF** arrays.

For connectionless socket types, the address from which the data originated is copied to the buffer pointed by *lpFrom*. On input, the value pointed to by *lpFromlen* is initialized to the size of this buffer, and is modified on completion to indicate the actual size of the address stored there.

As noted previously for overlapped sockets, the *lpFrom* and *lpFromlen* parameters are not updated until after the overlapped I/O has completed. The memory pointed to by these parameters must, therefore, remain available to the service provider and cannot be allocated on the Windows Sockets SPI client's stack frame. The *lpFrom* and *lpFromlen* parameters are ignored for connection-oriented sockets.

For byte stream-style sockets (for example, type SOCK_STREAM), incoming data is placed into the buffers until the buffers are filled, the connection is closed, or internally buffered data is exhausted. Regardless of whether or not the incoming data fills all the buffers, the completion indication occurs for overlapped sockets.

For message-oriented sockets, a single incoming message is placed into the supplied buffers, up to the total size of the buffers supplied, and the completion indication occurs for overlapped sockets. If the message is larger than the buffers supplied, the buffers are filled with the first part of the message. If the MSG_PARTIAL feature is supported by the service provider, the MSG_PARTIAL flag is set in *lpFlags* for the socket and subsequent receive operation(s) will retrieve the rest of the message. If MSG_PARTIAL is not supported but the protocol is reliable, **LPWSPRecvFrom** generates the error **WSAEMSGSIZE** and a subsequent receive operation with a larger buffer can be used to retrieve the entire message. Otherwise, (that is, the protocol is unreliable and does not support MSG_PARTIAL), the excess data is lost, and **LPWSPRecvFrom** generates the error WSAEMSGSIZE.

The *lpFlags* parameter can be used to influence the behavior of the function invocation beyond the options specified for the associated socket. That is, the semantics of this function are determined by the socket options and the *lpFlags* parameter. The latter is constructed by using the bitwise OR operator with any of the following values.

VALUE	MEANING
MSG_PEEK	Peeks at the incoming data. The data is copied into the buffer but is not removed from the input queue. This flag is valid only for nonoverlapped sockets.
MSG_OOB	Processes Out of Band (OOB) data.
MSG_PARTIAL	This flag is for message-oriented sockets only. On output, indicates that the data supplied is a portion of the message transmitted by the sender. Remaining portions of the message will be supplied in subsequent receive operations. A subsequent receive operation with MSG_PARTIAL flag cleared indicates end of sender's message. As an input parameter, MSG_PARTIAL indicates that the receive operation should complete even if only part of a message has been received by the service provider.

For message-oriented sockets, the MSG_PARTIAL bit is set in the *lpFlags* parameter if a partial message is received. If a complete message is received, MSG_PARTIAL is cleared in *lpFlags*. In the case of delayed completion, the value pointed to by *lpFlags* is not updated. When completion has been indicated the Windows Sockets SPI client should call LPWSPGetOverlappedResult and examine the flags pointed to by the *lpdwFlags* parameter.

If an overlapped operation completes immediately, **LPWSPRecv** returns a value of zero and the *lpNumberOfBytesRecvd* parameter is updated with the number of bytes received and the flag bits pointed by the *lpFlags* parameter are also updated. If the overlapped operation is successfully initiated and will complete later, **LPWSPRecv** returns SOCKET_ERROR and indicates error code WSA_IO_PENDING. In this case, *lpNumberOfBytesRecvd* and *lpFlags* is not updated. When the overlapped operation completes, the amount of data transferred is indicated either through the *cbTransferred* parameter in the completion routine (if specified), or through the *lpcbTransfer* parameter in **LPWSPGetOverlappedResult**. Flag values are obtained by examining the *lpdwFlags* parameter of **LPWSPGetOverlappedResult**.

Providers must allow this function to be called from within the completion routine of a previous **LPWSPRecv**, **LPWSPRecvFrom**, **LPWSPSend**, or **LPWSPSendTo** function. However, for a given socket, I/O completion routines cannot be nested. This permits time-sensitive data transmissions to occur entirely within a preemptive context.

The *lpOverlapped* parameter must be valid for the duration of the overlapped operation. If multiple I/O operations are simultaneously outstanding, each must reference a separate overlapped structure. The **WSAOverlapped** structure is defined in its own reference page.

If the *lpCompletionRoutine* parameter is null, the service provider signals the **hEvent** member of *lpOverlapped* when the overlapped operation completes if it contains a valid event object handle. A Windows Sockets SPI client can use LPWSPGetOverlappedResult to wait or poll on the event object.

If *lpCompletionRoutine* is not null, the **hEvent** member is ignored and can be used by the Windows Sockets SPI client to pass context information to the completion routine. 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 and process context 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 (supplied to the provider through the *lpThreadId* input parameter), a pointer to an APC function to be invoked, and a context value that is subsequently passed to the APC function. Because only a single context value is available, the APC function itself cannot be the client specified–completion routine. The service provider must instead supply a pointer to its own APC function that uses the supplied context value to access the needed result information for the overlapped operation, and then invokes the client specified–completion routine.

The prototype for the client-supplied completion routine is as follows:

```
void CALLBACK
CompletionRoutine(
  IN DWORD           dwError,
  IN DWORD           cbTransferred,
  IN LPWSAOVERLAPPED lpOverlapped,
  IN DWORD           dwFlags
);
```

The CompletionRoutine is a placeholder for a client-supplied function name. *dwError* specifies the completion status for the overlapped operation as indicated by *lpOverlapped*. *cbTransferred* specifies the number of bytes received. *dwFlags* contains information that would have appeared in *lpFlags* if the receive operation had completed immediately. This function does not return a value.

The completion routines can be called in any order, though not necessarily in the same order that the overlapped operations are completed. However, the posted buffers are guaranteed to be filled in the same order they are supplied.

> **NOTE**
> All I/O initiated by a given thread is canceled when that thread exits. For overlapped sockets, pending asynchronous operations can fail if the thread is closed before the operations complete. See **ExitThread** for more information.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	ws2spi.h

See also

WPUQueueApc

LPWSPGetOverlappedResult

LPWSPSocket

LPWSPSELECT callback function (ws2spi.h)

10/22/2022 • 5 minutes to read • Edit Online

The **LPWSPSelect** function determines the status of one or more sockets.

Syntax

```
LPWSPSELECT Lpwspselect;

int Lpwspselect(
  [in]      int nfds,
  [in, out] fd_set *readfds,
  [in, out] fd_set *writefds,
  [in, out] fd_set *exceptfds,
  [in]      const timeval *timeout,
  [out]     LPINT lpErrno
)
{...}
```

Parameters

`[in] nfds`

Ignored and included only for the sake of compatibility.

`[in, out] readfds`

Optional pointer to a set of sockets to be checked for readability.

`[in, out] writefds`

Optional pointer to a set of sockets to be checked for writability.

`[in, out] exceptfds`

Optional pointer to a set of sockets to be checked for errors.

`[in] timeout`

Maximum time for **LPWSPSelect** to wait, or **null** for a blocking operation, in the form of a **timeval** structure.

`[out] lpErrno`

Pointer to the error code.

Return value

The **LPWSPSelect** function returns the total number of descriptors that are ready and contained in the fd_set structures, or SOCKET_ERROR if an error occurred. If the return value is SOCKET_ERROR, a specific error code is available in *lpErrno*.

ERROR CODE	MEANING

WSAEFAULT	Windows Sockets service provider was unable to allocated needed resources for its internal operations, or the *readfds*, *writefds*, *exceptfds* or *timeval* parameters are not part of the user address space.
WSAENETDOWN	The network subsystem has failed.
WSAEINVAL	The *timeout* value is not valid, or all three descriptor parameters were **NULL**.
WSAEINTR	(Blocking) call was canceled through **LPWSPCancelBlockingCall**.
WSAEINPROGRESS	Blocking Windows Sockets call is in progress, or the service provider is still processing a callback function.
WSAENOTSOCK	One of the descriptor sets contains an entry that is not a socket.

Remarks

This function is used to determine the status of one or more sockets. For each socket, the caller can request information on read, write, or error status. The set of sockets for which a given status is requested is indicated by an fd_set structure. All entries in an **fd_set** correspond to sockets created by the service provider (that is, the **WSAPROTOCOL_INFO** structures describing their protocols have the same *providerId* value). Upon return, the structures are updated to reflect the subset of these sockets that meet the specified condition, and **LPWSPSelect** returns the total number of sockets meeting the conditions. A set of macros is provided for manipulating an **fd_set**. These macros are compatible with those used in the Berkeley software, but the underlying representation is completely different.

The parameter *readfds* identifies those sockets that are to be checked for readability. If the socket is currently listening through **LPWSPListen**, it will be marked as readable if an incoming connection request has been received, so that a LPWSPAccept is guaranteed to complete without blocking. For other sockets, readability means that queued data is available for reading so that a **LPWSPRecv** or LPWSPRecvFrom is guaranteed not to block.

For connection-oriented sockets, readability can also indicate that a close request has been received from the peer. If the virtual circuit was closed gracefully, then a **LPWSPRecv** will return immediately with zero bytes read. If the virtual circuit was reset, then a **LPWSPRecv** will complete immediately with an error code, such as WSAECONNRESET. The presence of OOB data will be checked if the socket option SO_OOBINLINE has been enabled (see **LPWSPSetSockOpt**).

The parameter *writefds* identifies those sockets that are to be checked for writability:

- If a socket is connecting through **LPWSPConnect**, writability means that the connection establishment successfully completed.
- If the socket is not in the process of listening through **LPWSPConnect**, writability means that a **LPWSPSend** or **LPWSPSendTo** are guaranteed to succeed.

However, they can block on a blocking socket if the **len** exceeds the amount of outgoing system buffer space available. It is not specified how long these guarantees can be assumed to be valid, particularly in a multithreaded environment.

The parameter *exceptfds* identifies those sockets that are to be checked for the presence of OOB data or any exceptional error conditions. Note that OOB data will only be reported in this way if the option SO_OOBINLINE is **FALSE**. If a socket is making a **LPWSPConnect** (nonblocking) connection, failure of the connect attempt is indicated in *exceptfds*. This specification does not define which other errors will be included.

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

Summary: A socket will be identified in a particular set when **LPWSPSelect** returns according to the following.

PARAMETER	DESCRIPTION
readfds:	If **LPWSPListen** is called, a connection is pending, **LPWSPAccept** will succeed.Data is available for reading (includes OOB data if SO_OOBINLINE is enabled).The connection has been closed/reset/terminated.
writefds:	If **LPWSPConnect** (nonblocking), connection has succeeded.Data can be sent.
exceptfds:	If **LPWSPConnect** (nonblocking), connection attempt failed.OOB data is available for reading (only if SO_OOBINLINE is disabled).

Three macros and one upcall function are defined in the header file Ws2spi.h for manipulating and checking the descriptor sets. The variable FD_SETSIZE determines the maximum number of descriptors in a set. (The default value of FD_SETSIZE is 64, which can be modified by #defining FD_SETSIZE to another value before #including Ws2spi.h.) Internally, socket handles in a fd_set are not represented as bit flags as in Berkeley UNIX. Their data representation is opaque. Use of these macros will maintain software portability between different socket environments.

The macros to manipulate and check fd_set contents are:

FD_CLR(*s*, **set*)

Removes the descriptor *s* from *set*.

FD_SET(*s*, **set*)

Adds descriptor *s* to *set*.

FD_ZERO(set*)**

Initializes the *set* to the **null** set.

The upcall function used to check the membership is:

intWPUFDIsSet (**SOCKET** *s*, **FD_SET FAR** **set*);

which will return nonzero if *s* is a member of the *set* or otherwise zero.

The parameter *timeout* controls how long the **LPWSPSelect** can take to complete. If *timeout* is a **null** pointer, **LPWSPSelect** will block indefinitely until at least one descriptor meets the specified criteria. Otherwise, *timeout* points to a **timeval** structure that specifies the maximum time that **LPWSPSelect** should wait before returning. When **LPWSPSelect** returns, the contents of the **timeval** structure are not altered. If **timeval** is initialized to {0,

0}, **LPWSPSelect** will return immediately; this is used to poll the state of the selected sockets. If this is the case, then the **LPWSPSelect** call is considered nonblocking and the standard assumptions for nonblocking calls apply. For example, the blocking hook will not be called, and the Windows Sockets provider will not yield.

> **NOTE**
> The **LPWSPSelect** function has no effect on the persistence of socket events registered with **LPWSPAsyncSelect** or **LPWSPEventSelect**.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	ws2spi.h

See also

LPWSPAccept

LPWSPConnect

LPWSPEventSelect

LPWSPRecv

LPWSPRecvFrom

LPWSPSend

LPWSPSendTo

LPWSPSEND callback function (ws2spi.h)

10/22/2022 • 8 minutes to read • Edit Online

The **LPWSPSend** function sends data on a connected socket.

Syntax

```
LPWSPSEND Lpwspsend;

int Lpwspsend(
  [in]  SOCKET s,
  [in]  LPWSABUF lpBuffers,
  [in]  DWORD dwBufferCount,
  [out] LPDWORD lpNumberOfBytesSent,
  [in]  DWORD dwFlags,
  [in]  LPWSAOVERLAPPED lpOverlapped,
  [in]  LPWSAOVERLAPPED_COMPLETION_ROUTINE lpCompletionRoutine,
  [in]  LPWSATHREADID lpThreadId,
  [out] LPINT lpErrno
)
{...}
```

Parameters

`[in] s`

A descriptor identifying a connected socket.

`[in] lpBuffers`

A pointer to an array of WSABUF structures. Each **WSABUF** structure contains a pointer to a buffer and the length of the buffer, in bytes. For a Winsock application, once the **LPWSPSend** function is called, the system owns these buffers and the application may not access them. Data buffers referenced in each WSABUF structure are owned by the system and your application may not access them for the lifetime of the call.

`[in] dwBufferCount`

The number of WSABUF structures in the *lpBuffers* array.

`[out] lpNumberOfBytesSent`

A pointer to the number of bytes sent by this call.

`[in] dwFlags`

A set of flags that specifies the way in which the call is made.

`[in] lpOverlapped`

A pointer to a **WSAOverlapped** structure (ignored for nonoverlapped sockets).

`[in] lpCompletionRoutine`

Type: _In_opt_ **LPWSAOVERLAPPED_COMPLETION_ROUTINE**

A pointer to the completion routine called when the send operation has been completed (ignored for

nonoverlapped sockets).

`[in] 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.

`[out] lpErrno`

A pointer to the error code.

Return value

If no error occurs and the send operation has completed immediately, **LPWSPSend** returns zero. Note that in this case the completion routine, if specified, will have already been queued. Otherwise, a value of SOCKET_ERROR is returned, and a specific error code is available in *lpErrno*. The error code WSA_IO_PENDING indicates that the overlapped operation has been successfully initiated and that completion will be indicated at a later time. Any other error code indicates that no overlapped operation was initiated and no completion indication will occur.

ERROR CODE	MEANING
WSAENETDOWN	The network subsystem has failed.
WSAEACCES	Requested address is a broadcast address, but the appropriate flag was not set.
WSAEINPROGRESS	Blocking Windows Sockets call is in progress, or the service provider is still processing a callback function.
WSAEFAULT	The *lpBuffers* parameter is not totally contained in a valid part of the user address space.
WSAENETRESET	The connection has been broken due to the remote host resetting.
WSAENOBUFS	The connection has been broken due to keep-alive activity detecting a failure while the operation was in progress.
WSAENOTCONN	Socket is not connected.
WSAENOTSOCK	The descriptor is not a socket.
WSAEOPNOTSUPP	MSG_OOB was specified, but the socket is not stream-style such as type SOCK_STREAM, OOB data is not supported in the communication domain associated with this socket, MSG_PARTIAL is not supported, or the socket is unidirectional and supports only receive operations.

WSAESHUTDOWN	Socket has been shut down; it is not possible to **LPWSPSend** on a socket after **LPWSPShutdown** has been invoked with how set to SD_SEND or SD_BOTH.
WSAEWOULDBLOCK	**Windows NT:** Overlapped sockets: There are too many outstanding overlapped I/O requests.Nonoverlapped sockets: The socket is marked as nonblocking and the send operation cannot be completed immediately.
WSAEMSGSIZE	Socket is message oriented, and the message is larger than the maximum supported by the underlying transport.
WSAEINVAL	Socket has not been bound with **LPWSPBind**, or the socket is not created with the overlapped flag.
WSAECONNABORTED	Virtual circuit was terminated due to a time-out or other failure.
WSAECONNRESET	Virtual circuit was reset by the remote side.
WSA_OPERATION_ABORTED	Overlapped operation has been canceled due to the closure of the socket, or the execution of the SIO_FLUSH command in **LPWSPIoctl**.

Remarks

The **LPWSPSend** function is used to write outgoing data from one or more buffers on a connection-oriented socket specified by *s*. It can also be used, however, on connectionless sockets that have a stipulated default peer address established through the **LPWSPConnect** function.

For overlapped sockets (created using LPWSPSocket with flag WSA_FLAG_OVERLAPPED) this will occur using overlapped I/O, unless both *lpOverlapped* and *lpCompletionRoutine* are null in which case the socket is treated as a nonoverlapped socket. A completion indication will occur (invocation of the completion routine or setting of an event object) when the supplied buffer(s) have been consumed by the transport. If the operation does not complete immediately, the final completion status is retrieved through the completion routine or LPWSPGetOverlappedResult.

For nonoverlapped sockets, the parameters *lpOverlapped*, *lpCompletionRoutine*, and *lpThreadId* are ignored and **LPWSPSend** adopts the regular synchronous semantics. Data is copied from the supplied buffer(s) into the transport's buffer. If the socket is nonblocking and stream oriented, and there is not sufficient space in the transport's buffer, **LPWSPSend** will return with only part of the supplied buffers having been consumed. Given the same buffer situation and a blocking socket, **LPWSPSend** will block until all of the supplied buffer contents have been consumed.

The array of WSABUF structures pointed to by the *lpBuffers* parameter is transient. If this operation completes in an overlapped manner, it is the service provider's responsibility to capture these **WSABUF** structures before returning from this call. This enables applications to build stack-based **WSABUF** arrays.

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

Note that the successful completion of a **LPWSPSend** does not indicate that the data was successfully

delivered.

dwFlags can be used to influence the behavior of the function invocation beyond the options specified for the associated socket. That is, the semantics of this function are determined by the socket options and the *dwFlags* parameter. The latter is constructed by using the bitwise OR operator with any of the following values.

VALUE	MEANING
MSG_DONTROUTE	Specifies that the data should not be subject to routing. A Windows Sockets service provider can choose to ignore this flag;.
MSG_OOB	Sends OOB data (stream-style socket such as SOCK_STREAM only).
MSG_PARTIAL	Specifies that *lpBuffers* only contains a partial message. Note that the error code WSAEOPNOTSUPP will be returned for messages that do not support partial message transmissions.

If an overlapped operation completes immediately, **LPWSPSend** returns a value of zero and the *lpNumberOfBytesSent* parameter is updated with the number of bytes sent. If the overlapped operation is successfully initiated and will complete later, **LPWSPSend** returns SOCKET_ERROR and indicates error code WSA_IO_PENDING. In this case, *lpNumberOfBytesSent* is not updated. When the overlapped operation completes the amount of data transferred is indicated either through the *cbTransferred* parameter in the completion routine (if specified), or through the *lpcbTransfer* parameter in LPWSPGetOverlappedResult.

Providers must allow this function to be called from within the completion routine of a previous **LPWSPRecv**, **LPWSPRecvFrom**, **LPWSPSend** or **LPWSPSendTo** function. However, for a given socket, I/O completion routines cannot be nested. This permits time-sensitive data transmissions to occur entirely within a preemptive context.

The *lpOverlapped* parameter must be valid for the duration of the overlapped operation. If multiple I/O operations are simultaneously outstanding, each must reference a separate overlapped structure. The **WSAOverlapped** structure is defined in its own reference page.

If the *lpCompletionRoutine* parameter is null, the service provider signals the **hEvent** member of *lpOverlapped* when the overlapped operation completes if it contains a valid event object handle. The Windows Sockets SPI client can use LPWSPGetOverlappedResult to wait or poll on the event object.

If *lpCompletionRoutine* is not null, the **hEvent** member is ignored and can be used by the Windows Sockets SPI client to pass context information to the completion routine. A client that passes a non-null *lpCompletionRoutine* and later calls **WSAGetOverlappedResult** for the same overlapped I/O request may not set the *fWait* parameter for that invocation of **WSAGetOverlappedResult** to **TRUE**. In this case the usage of the **hEvent** member is undefined, and attempting to wait on the **hEvent** member would produce unpredictable results.

A service provider arranges for a function to be executed in the proper thread and process context 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.

A service provider arranges for a function to be executed in the proper thread by calling **WPUQueueApc**. Note that this function must be invoked while in the context of the same process (but not necessarily the same

thread) that was used to initiate the overlapped operation. It is the service provider's responsibility to arrange for this process context to be active prior to calling **WPUQueueApc**.

The **WPUQueueApc** function takes as input parameters a pointer to a **WSATHREADID** structure (supplied to the provider through the *lpThreadId* input parameter), a pointer to an APC function to be invoked, and a context value that is subsequently passed to the APC function. Because only a single context value is available, the APC function itself cannot be the client specified–completion routine. The service provider must instead supply a pointer to its own APC function that uses the supplied context value to access the needed result information for the overlapped operation, and then invokes the client specified–completion routine.

The prototype for the client-supplied completion routine is as follows.

```
void CALLBACK
CompletionRoutine(
  IN DWORD           dwError,
  IN DWORD           cbTransferred,
  IN LPWSAOVERLAPPED lpOverlapped,
  IN DWORD           dwFlags
);
```

The CompletionRoutine is a placeholder for a client supplied function name. *dwError* specifies the completion status for the overlapped operation as indicated by *lpOverlapped*. *cbTransferred* specifies the number of bytes sent. No flag values are currently defined and the *dwFlags* value will be zero. This function does not return a value.

The completion routines can be called in any order, though not necessarily in the same order that the overlapped operations are completed. However, the service provider guarantees to the client that posted buffers are sent in the same order they are supplied.

> **NOTE**
> All I/O initiated by a given thread is canceled when that thread exits. For overlapped sockets, pending asynchronous operations can fail if the thread is closed before the operations complete. See **ExitThread** for more information.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	ws2spi.h

See also

WPUQueueApc

LPWSPGetOverlappedResult

LPWSPSocket

LPWSPSENDDISCONNECT callback function (ws2spi.h)

10/22/2022 • 2 minutes to read • Edit Online

The **LPWSPSendDisconnect** function initiates termination of the connection for the socket and sends disconnect data.

Syntax

```
LPWSPSENDDISCONNECT Lpwspsenddisconnect;

int Lpwspsenddisconnect(
  [in]  SOCKET s,
  [in]  LPWSABUF lpOutboundDisconnectData,
  [out] LPINT lpErrno
)
{...}
```

Parameters

`[in] s`

Descriptor identifying a socket.

`[in] lpOutboundDisconnectData`

Pointer to the outgoing disconnect data.

`[out] lpErrno`

Pointer to the error code.

Return value

If no error occurs, **LPWSPSendDisconnect** returns zero. Otherwise, a value of SOCKET_ERROR is returned, and a specific error code is available in *lpErrno*.

ERROR CODE	MEANING
WSAENETDOWN	The network subsystem has failed.
WSAENOPROTOOPT	Parameter *lpOutboundDisconnectData* is not null, and the disconnect data is not supported by the service provider.
WSAEINPROGRESS	Blocking Windows Sockets call is in progress, or the service provider is still processing a callback function.
WSAENOTCONN	Socket is not connected (connection-oriented sockets only).

WSAENOTSOCK	The descriptor is not a socket.
WSAEFAULT	The *lpOutboundDisconnectData* parameter is not totally contained in a valid part of the user address space.

Remarks

The **LPWSPSendDisconnect** function is used on connection-oriented sockets to disable transmission, and to initiate termination of the connection along with the transmission of disconnect data, if any.

After this function has been successfully issued, subsequent sends are disallowed.

The *lpOutboundDisconnectData* parameter, if not null, points to a buffer containing the outgoing disconnect data to be sent to the remote party.

Note that **LPWSPSendDisconnect** does not close the socket, and that resources attached to the socket will not be freed until **LPWSPCloseSocket** is invoked.

> **NOTE**
>
> The **LPWSPSendDisconnect** function does not block regardless of the SO_LINGER setting on the socket. A Windows Sockets SPI client should not rely on being able to reuse a socket after it has been disconnected. In particular, a Windows Sockets provider is not required to support the use of **LPWSPConnect** on such a socket.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	ws2spi.h

See also

LPWSPConnect

LPWSPSocket

LPWSPSENDTO callback function (ws2spi.h)

10/22/2022 • 9 minutes to read • Edit Online

The **LPWSPSendTo** function sends data to a specific destination using overlapped I/O.

Syntax

```
LPWSPSENDTO Lpwspsendto;

int Lpwspsendto(
  [in]  SOCKET s,
  [in]  LPWSABUF lpBuffers,
  [in]  DWORD dwBufferCount,
  [out] LPDWORD lpNumberOfBytesSent,
  [in]  DWORD dwFlags,
  [in]  const sockaddr *lpTo,
  [in]  int iTolen,
  [in]  LPWSAOVERLAPPED lpOverlapped,
  [in]  LPWSAOVERLAPPED_COMPLETION_ROUTINE lpCompletionRoutine,
  [in]  LPWSATHREADID lpThreadId,
  [out] LPINT lpErrno
)
{...}
```

Parameters

`[in] s`

A descriptor identifying a socket.

`[in] lpBuffers`

A pointer to an array of WSABUF structures. Each **WSABUF** structure contains a pointer to a buffer and the length of the buffer, in bytes. For a Winsock application, once the **LPWSPSendTo** function is called, the system owns these buffers and the application may not access them. Data buffers referenced in each WSABUF structure are owned by the system and your application may not access them for the lifetime of the call.

`[in] dwBufferCount`

The number of WSABUF structures in the *lpBuffers* array.

`[out] lpNumberOfBytesSent`

A pointer to the number of bytes sent by this call.

`[in] dwFlags`

A set of flags that specifies the way in which the call is made.

`[in] lpTo`

An optional pointer to the address of the target socket in the **sockaddr** structure.

`[in] iTolen`

The size, in bytes, of the address pointed to by the *lpTo* parameter.

`[in] lpOverlapped`

A pointer to a **WSAOverlapped** structure (ignored for nonoverlapped sockets).

`[in] lpCompletionRoutine`

Type: _In_opt_ **LPWSAOVERLAPPED_COMPLETION_ROUTINE**

A pointer to the completion routine called when the send operation has been completed (ignored for nonoverlapped sockets).

`[in] 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.

`[out] lpErrno`

A pointer to the error code.

Return value

If no error occurs and the receive operation has completed immediately, **LPWSPSendTo** returns zero. Note that in this case the completion routine, if specified, will have already been queued. Otherwise, a value of SOCKET_ERROR is returned, and a specific error code is available in *lpErrno*. The error code WSA_IO_PENDING indicates that the overlapped operation has been successfully initiated and that completion will be indicated at a later time. Any other error code indicates that no overlapped operation was initiated and no completion indication will occur.

ERROR CODE	MEANING
WSAENETDOWN	The network subsystem has failed.
WSAEACCES	Requested address is a broadcast address, but the appropriate flag was not set.
WSAEINTR	(Blocking) call was canceled through **LPWSPCancelBlockingCall**.
WSAEINPROGRESS	Blocking Windows Sockets call is in progress, or the service provider is still processing a callback function.
WSAEFAULT	The *lpBuffers* or *lpTo* parameters are not part of the user address space, or the *lpTo* parameter is too small.
WSAENETRESET	The connection has been broken due to keep-alive activity detecting a failure while the operation was in progress.
WSAENOBUFS	Windows Sockets provider reports a buffer deadlock.
WSAENOTCONN	Socket is not connected (connection-oriented sockets only).

WSAENOTSOCK	The descriptor is not a socket.
WSAEOPNOTSUPP	MSG_OOB was specified, but the socket is not stream-style such as type SOCK_STREAM, OOB data is not supported in the communication domain associated with this socket, MSG_PARTIAL is not supported, or the socket is unidirectional and supports only receive operations.
WSAESHUTDOWN	Socket has been shut down; it is not possible to use **LPWSPSendTo** on a socket after **LPWSPShutdown** has been invoked with *how* set to SD_SEND or SD_BOTH.
WSAEWOULDBLOCK	Windows NT: Overlapped sockets: there are too many outstanding overlapped I/O requests.Nonoverlapped sockets: The socket is marked as nonblocking and the send operation cannot be completed immediately.
WSAEMSGSIZE	Socket is message oriented, and the message is larger than the maximum supported by the underlying transport.
WSAEINVAL	Socket has not been bound with **LPWSPBind**, or the socket is not created with the overlapped flag.
WSAECONNABORTED	Virtual circuit was terminated due to a time-out or other failure.
WSAECONNRESET	Virtual circuit was reset by the remote side.
WSAEADDRNOTAVAIL	Remote address is not a valid address (for example, ADDR_ANY).
WSAEAFNOSUPPORT	Addresses in the specified family cannot be used with this socket.
WSAEDESTADDRREQ	Destination address is required.
WSAENETUNREACH	The network cannot be reached from this host at this time.
WSA_OPERATION_ABORTED	Overlapped operation has been canceled due to the closure of the socket, or the execution of the SIO_FLUSH command in **LPWSPIoctl**.

Remarks

The **LPWSPSendTo** function is normally used on a connectionless socket specified by *s* to send a datagram contained in one or more buffers to a specific peer socket identified by the *lpTo* parameter. Even if the connectionless socket has been previously connected to a specific address with the LPWSPConnect function, *lpTo* overrides the destination address for that particular datagram only. On a connection-oriented socket, the *lpTo* and *iToLen* parameters are ignored; in this case the **LPWSPSendTo** function is equivalent to **LPWSPSend**.

For overlapped sockets (created using **LPWSPSocket** with flag WSA_FLAG_OVERLAPPED) this will occur using overlapped I/O, unless both *lpOverlapped* and *lpCompletionRoutine* are **NULL** in which case the socket is treated as a nonoverlapped socket. A completion indication will occur (invocation of the completion routine or setting of an event object) when the supplied buffer(s) have been consumed by the transport. If the operation does not complete immediately, the final completion status is retrieved through the completion routine or LPWSPGetOverlappedResult.

For nonoverlapped sockets, the parameters *lpOverlapped*, *lpCompletionRoutine*, and *lpThreadId* are ignored and **LPWSPSendTo** adopts the regular synchronous semantics. Data is copied from the supplied buffer(s) into the transport's buffer. If the socket is nonblocking and stream oriented, and there is not sufficient space in the transport's buffer, **LPWSPSendTo** will return with only part of the Windows Sockets SPI client's buffers having been consumed. Given the same buffer situation and a blocking socket, **LPWSPSendTo** will block until all of the Windows Sockets SPI client's buffer contents have been consumed.

The array of WSABUF structures pointed to by the *lpBuffers* parameter is transient. If this operation completes in an overlapped manner, it is the service provider's responsibility to capture these **WSABUF** structures before returning from this call. This enables applications to build stack-based **WSABUF** arrays.

For message-oriented sockets, care must be taken not to exceed the maximum message size of the underlying transport, which can be obtained by getting the value of socket option SO_MAX_MSG_SIZE. If the data is too long to pass atomically through the underlying protocol the error **WSAEMSGSIZE** is returned, and no data is transmitted.

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

The *iFlags* parameter can be used to influence the behavior of the function invocation beyond the options specified for the associated socket. That is, the semantics of this function are determined by the socket options and the *dwFlags* parameter. The latter is constructed by using the bitwise OR operator with any of the following values.

VALUE	MEANING
MSG_DONTROUTE	Specifies that the data should not be subject to routing. A Windows Sockets service provider can choose to ignore this flag.
MSG_OOB	Sends OOB data (stream-style socket such as SOCK_STREAM only).
MSG_PARTIAL	Specifies that *lpBuffers* only contains a partial message. Note that the error code WSAEOPNOTSUPP will be returned by transports that do not support partial message transmissions.

If an overlapped operation completes immediately, **LPWSPSendTo** returns a value of zero and the *lpNumberOfBytesSent* parameter is updated with the number of bytes sent. If the overlapped operation is successfully initiated and will complete later, **LPWSPSendTo** returns SOCKET_ERROR and indicates error code WSA_IO_PENDING. In this case, *lpNumberOfBytesSent* is not updated. When the overlapped operation completes the amount of data transferred is indicated either through the *cbTransferred* parameter in the completion routine (if specified), or through the *lpcbTransfer* parameter in LPWSPGetOverlappedResult.

Providers must allow this function to be called from within the completion routine of a previous **LPWSPRecv**,

LPWSPRecvFrom, **LPWSPSend** or **LPWSPSendTo** function. However, for a given socket, I/O completion routines cannot be nested. This permits time-sensitive data transmissions to occur entirely within a preemptive context.

The *lpOverlapped* parameter must be valid for the duration of the overlapped operation. If multiple I/O operations are simultaneously outstanding, each must reference a separate overlapped structure. The **WSAOverlapped** structure is defined in its own reference page.

If the *lpCompletionRoutine* parameter is **null**, the service provider signals the **hEvent** member of *lpOverlapped* when the overlapped operation completes if it contains a valid event object handle. Windows Sockets SPI clients can use LPWSPGetOverlappedResult to wait or poll on the event object.

If *lpCompletionRoutine* is not **null**, the **hEvent** member is ignored and can be used by the Windows Sockets SPI client to pass context information to the completion routine. A client that passes a non-**null** *lpCompletionRoutine* and later calls **WSAGetOverlappedResult** for the same overlapped I/O request may not set the *fWait* parameter for that invocation of **WSAGetOverlappedResult** to **TRUE**. In this case the usage of the **hEvent** member is undefined, and attempting to wait on the **hEvent** member would produce unpredictable results.

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.

The **WPUQueueApc** function takes as input parameters a pointer to a **WSATHREADID** structure (supplied to the provider through the *lpThreadId* input parameter), a pointer to an APC function to be invoked, and a context value that is subsequently passed to the APC function. Because only a single context value is available, the APC function itself cannot be the client specified–completion routine. 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.

The prototype for the client-supplied completion routine is as follows.

```
void CALLBACK
CompletionRoutine(
  IN DWORD           dwError,
  IN DWORD           cbTransferred,
  IN LPWSAOVERLAPPED lpOverlapped,
  IN DWORD           dwFlags
);
```

The CompletionRoutine is a placeholder for a client-supplied function name. *dwError* specifies the completion status for the overlapped operation as indicated by *lpOverlapped*. *cbTransferred* specifies the number of bytes sent. No flag values are currently defined and the *dwFlags* value will be zero. This function does not return a value.

The completion routines can be called in any order, though not necessarily in the same order that the overlapped operations are completed. However, the service provider guarantees to the client that posted buffers are sent in the same order they are supplied.

> **NOTE**
> All I/O initiated by a given thread is canceled when that thread exits. For overlapped sockets, pending asynchronous operations can fail if the thread is closed before the operations complete. See **ExitThread** for more information.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	ws2spi.h

See also

WPUQueueApc

LPWSPGetOverlappedResult

LPWSPSocket

LPWSPSETSOCKOPT callback function (ws2spi.h)

10/22/2022 • 6 minutes to read • Edit Online

The **LPWSPSetSockOpt** function sets a socket option.

Syntax

```
LPWSPSETSOCKOPT Lpwspsetsockopt;

int Lpwspsetsockopt(
  [in]  SOCKET s,
  [in]  int level,
  [in]  int optname,
  [in]  const char *optval,
  [in]  int optlen,
  [out] LPINT lpErrno
)
{...}
```

Parameters

`[in] s`

The descriptor that identifies a socket.

`[in] level`

The level at which the option is defined; the supported levels include **SOL_SOCKET**. For more information, see Winsock Annexes.

`[in] optname`

The socket option for which the value is to be set.

`[in] optval`

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

`[in] optlen`

The size, in bytes, of the *optval* buffer.

`[out] lpErrno`

A pointer to the error code.

Return value

If no error occurs, **LPWSPSetSockOpt** returns zero. Otherwise, a value of **SOCKET_ERROR** is returned, and a specific error code is available in *lpErrno*.

ERROR CODE	MEANING

WSAENETDOWN	The network subsystem has failed.
WSAEFAULT	The *optval* is not in a valid part of the process address space or *optlen* parameter is too small.
WSAEINPROGRESS	Function is invoked when a callback is in progress.
WSAEINPROGRESS	Blocking Windows Sockets call is in progress, or the service provider is still processing a callback function.
WSAEINVAL	The *level* is not valid, or the information in *optval* is not valid.
WSAENETRESET	The connection has been broken due to keep-alive activity detecting a failure while the operation was in progress.
WSAENOPROTOOPT	Option is unknown or unsupported for the specified provider.
WSAENOTCONN	The connection has been reset when SO_KEEPALIVE is set.
WSAENOTSOCK	The descriptor is not a socket.

Remarks

The **LPWSPSetSockOpt** function sets the current value for a socket option associated with a socket of any type, in any state. Although options can exist at multiple protocol levels, they are always present at the uppermost socket level. Options affect socket operations, such as whether broadcast messages can be sent on the socket.

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

For more information about socket options, see **Socket Options**.

level = SOL_SOCKET

VALUE	TYPE	MEANING
SO_BROADCAST	BOOL	Enables transmission and receipt of broadcast messages on the socket.
SO_DEBUG	BOOL	Records debugging information.
SO_DONTLINGER	BOOL	Reserved.

VALUE	TYPE	MEANING
SO_DONTROUTE	BOOL	Disabled routing: send directly to an interface. Setting this socket option succeeds but is ignored on AF_INET sockets; fails on AF_INET6 sockets with WSAENOPROTOOPT . This option is not supported on ATM sockets (results in an error).
SO_GROUP_PRIORITY	int	Reserved.
SO_KEEPALIVE	BOOL	Sends keep-alives. Not supported on ATM sockets (results in an error).
SO_LINGER	struct linger	Lingers on close if unsent data is present.
SO_OOBINLINE	BOOL	Receives OOB data in the normal data stream.
SO_RCVBUF	int	Specifies 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_REUSEADDR	BOOL	Allows the socket to be bound to an address that is already in use. (See **Bind**.) Not applicable on ATM sockets.
SO_SNDBUF	int	Specifies 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.
PVD_CONFIG	Service Provider Dependent	This object stores the configuration information for the service provider associated with socket *s*. The exact format of this data structure is service provider specific.

Calling **LPWSPGetSockopt** with an unsupported option will result in an error code of WSAENOPROTOOPT returned in *lpErrno*.

SO_DEBUG

Windows Sockets service providers are encouraged, but not required, to supply output debug information if the **SO_DEBUG** option is set by a Windows Sockets SPI client. The mechanism for generating the debug information and the format are beyond the scope of this specification.

SO_GROUP_PRIORITY

Reserved.

SO_KEEPALIVE

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

SO_LINGER

SO_LINGER controls the action taken when unsent data is queued on a socket and a **LPWSPCloseSocket** is performed. See **LPWSPCloseSocket** for a description of the way in which the **SO_LINGER** settings affect the semantics of **LPWSPCloseSocket**. The Windows Sockets SPI client sets the desired behavior by creating a **LINGER** structure, pointed to by the *optval* parameter, with the following elements.

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

To enable **SO_LINGER**, a Windows Sockets SPI client should set **l_onoff** to a nonzero value, set **l_linger** to zero or the desired time-out, in seconds, and call **LPWSPSetSockOpt**. To enable **SO_DONTLINGER**, that is, disable SO_LINGER, **l_onoff** should be set to zero and **LPWSPSetSockOpt** should be called. Be aware that enabling **SO_LINGER** with a nonzero time-out on a nonblocking socket is not recommended. For more information, see **LPWSPCloseSocket**.

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

SO_REUSEADDR

By default, a socket cannot be bound (for more information, see **LPWSPBind**) to a local address that is already in use. On occasion, however, it may be desirable to reuse an address in this way. Because every connection is uniquely identified by the combination of local and remote addresses, there is no problem with having two sockets bound to the same local address as long as the remote addresses are different. To inform the Windows Sockets provider that a **LPWSPBind** on a socket should be allowed to bind to a local address that is already in use by another socket, the Windows Sockets SPI client should set the **SO_REUSEADDR** socket option for the socket before issuing the **LPWSPBind**. Be aware that the option is interpreted only at the time of the **LPWSPBind**: it is therefore unnecessary, but harmless, to set the option on a socket that is not to be bound to an existing address, and setting or resetting the option after the **LPWSPBind** has no effect on this or any other socket.

SO_SNDBUF

When a Windows Sockets implementation supports the **SO_RCVBUF** and **SO_SNDBUF** options, a Windows Sockets SPI client can request different buffer sizes (larger or smaller). The call can succeed even though the service provider did not make available the entire amount requested. A Windows Sockets SPI client must call **LPWSPGetSockopt** with the same option to verify the buffer size actually provided.

PVD_CONFIG

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

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	ws2spi.h

See also

LPWSPBind

LPWSPEventSelect

LPWSPGetSockopt

LPWSPIoctl

LPWSPSocket

LPWSPSHUTDOWN callback function (ws2spi.h)

10/22/2022 • 2 minutes to read • Edit Online

The **LPWSPShutdown** function disables sends and/or receives on a socket.

Syntax

```
LPWSPSHUTDOWN Lpwspshutdown;

int Lpwspshutdown(
  [in]  SOCKET s,
  [in]  int how,
  [out] LPINT lpErrno
)
{...}
```

Parameters

`[in] s`

Descriptor identifying a socket.

`[in] how`

Flag that describes what types of operation will no longer be allowed.

`[out] lpErrno`

Pointer to the error code.

Return value

If no error occurs, **LPWSPShutdown** returns zero. Otherwise, a value of SOCKET_ERROR is returned, and a specific error code is available in *lpErrno*.

ERROR CODE	MEANING
WSAENETDOWN	The network subsystem has failed.
WSAEINVAL	The *how* is not valid, or is not consistent with the socket type. For example, SD_SEND is used with a UNI_RECV socket type.
WSAEINPROGRESS	Function is invoked when a callback is in progress.
WSAENOTCONN	Socket is not connected (connection-oriented sockets only).

WSAENOTSOCK	The descriptor is not a socket.

Remarks

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

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

If *how* is SD_SEND, subsequent sends on the socket are disallowed. For TCP sockets, a FIN will be sent. Setting *how* to SD_BOTH disables both sends and receives as described above.

Note that **LPWSPShutdown** does not close the socket, and resources attached to the socket will not be freed until **LPWSPCloseSocket** is invoked.

> **NOTE**
>
> The **LPWSPShutdown** function does not block regardless of the SO_LINGER setting on the socket. A Windows Sockets SPI client should not rely on being able to reuse a socket after it has been shut down. In particular, a Windows Sockets service provider is not required to support the use of **LPWSPConnect** on such a socket.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	ws2spi.h

See also

LPWSPConnect

LPWSPSocket

LPWSPSOCKET callback function (ws2spi.h)

10/22/2022 • 5 minutes to read • Edit Online

The **LPWSPSocket** function creates a socket. For info about the part played by **LPWSPSocket** in creating a shared socket, see Shared sockets and Shared sockets in the SPI.

Syntax

```
LPWSPSOCKET Lpwspsocket;

SOCKET Lpwspsocket(
  [in]  int af,
  [in]  int type,
  [in]  int protocol,
  [in]  LPWSAPROTOCOL_INFOW lpProtocolInfo,
  [in]  GROUP g,
        DWORD dwFlags,
  [out] LPINT lpErrno
)
{...}
```

Parameters

`[in] af`

Address family specification.

`[in] type`

Type specification for the new socket.

`[in] protocol`

Protocol to be used with the socket that is specific to the indicated address family.

`[in] lpProtocolInfo`

Pointer to a WSAProtocol_Info structure that defines the characteristics of the socket to be created.

`[in] g`

Reserved.

`dwFlags`

Socket attribute specification.

`[out] lpErrno`

Pointer to the error code.

Return value

If no error occurs, **LPWSPSocket** returns a descriptor referencing the new socket. Otherwise, a value of INVALID_SOCKET is returned, and a specific error code is available in *lpErrno*.

ERROR CODE	MEANING
WSAENETDOWN	The network subsystem has failed.
WSAEAFNOSUPPORT	The specified address family is not supported.
WSAEINPROGRESS	Blocking Windows Sockets call is in progress, or the service provider is still processing a callback function.
WSAEMFILE	No more socket descriptors are available.
WSAENOBUFS	No buffer space is available. The socket cannot be created.
WSAEPROTONOSUPPORT	The specified protocol is not supported.
WSAEPROTOTYPE	The specified protocol is the wrong type for this socket.
WSAESOCKTNOSUPPORT	The specified socket type is not supported in this address family.
WSAEINVAL	Parameter *g* specified is not valid.

Remarks

The **LPWSPSocket** function causes a socket descriptor and any related resources to be allocated. By default, the created socket will not have the overlapped attribute. Windows Sockets providers are encouraged to be realized as Windows installable file systems, and supply system file handles as socket descriptors. These providers must call WPUModifyIFSHandle prior to returning from this function. For nonfile-system Windows Sockets providers, WPUCreateSocketHandle must be used to acquire a unique socket descriptor from the Ws2_32.dll prior to returning from this function. See
Descriptor Allocation for more information.

The values for *af*, *type*, and *protocol* are those supplied by the application in the corresponding API functions socket or WSASocket. A service provider is free to ignore or pay attention to any or all of these values as is appropriate for the particular protocol. However, the provider must be willing to accept the value of zero for *af* and *type*, since the Ws2_32.dll considers these to be wildcard values. Also the value of manifest constant **FROM_PROTOCOL_INFO** must be accepted for any of *af*, *type*, and *protocol*. This value indicates that the Windows Sockets 2 application needs to use the corresponding values from the WSAProtocol_Info structure (**iAddressFamily**, **iSocketType**, **iProtocol**).

The *dwFlags* parameter can be used to specify the attributes of the socket by using the bitwise OR operator with any of the following flags.

FLAG	MEANING
WSA_FLAG_OVERLAPPED	This flag causes an overlapped socket to be created. Overlapped sockets can utilize LPWSPSend, LPWSPSendTo, LPWSPRecv, LPWSPRecvFrom, and LPWSPIoctl for overlapped I/O operations, which allow multiple operations to be initiated and in process simultaneously. All functions that allow overlapped operations also support nonoverlapped usage on an overlapped socket if the values for parameters related to overlapped operation are null.
WSA_FLAG_MULTIPOINT_C_ROOT	Indicates that the socket created will be a c_root in a multipoint session. Only allowed if a rooted control plane is indicated in the protocol's WSAProtocol_Info structure.
WSA_FLAG_MULTIPOINT_C_LEAF	Indicates that the socket created will be a c_leaf in a multicast session. Only allowed if XP1_SUPPORT_MULTIPOINT is indicated in the protocol's WSAProtocol_Info structure.
WSA_FLAG_MULTIPOINT_D_ROOT	Indicates that the socket created will be a d_root in a multipoint session. Only allowed if a rooted data plane is indicated in the protocol's WSAProtocol_Info structure.
WSA_FLAG_MULTIPOINT_D_LEAF	Indicates that the socket created will be a d_leaf in a multipoint session. Only allowed if XP1_SUPPORT_MULTIPOINT is indicated in the protocol's WSAProtocol_Info structure.

NOTE

For multipoint sockets, exactly one WSA_FLAG_MULTIPOINT_C_ROOT or WSA_FLAG_MULTIPOINT_C_LEAF must be specified, and exactly one of WSA_FLAG_MULTIPOINT_D_ROOT or WSA_FLAG_MULTIPOINT_D_LEAF must be specified. Refer to Protocol-Independent Multicast and Multipoint in the SPI for additional information.

Connection-oriented sockets such as SOCK_STREAM provide full-duplex connections, and must be in a connected state before any data can be sent or received on them. A connection to another socket is created with a LPWSPConnect call. Once connected, data can be transferred using LPWSPSend and LPWSPRecv calls. When a session has been completed, a LPWSPCloseSocket must be performed.

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

Connectionless, message-oriented sockets allow sending and receiving of datagrams to and from arbitrary peers using LPWSPSendTo and LPWSPRecvFrom. If such a socket is connected by using LPWSPConnect to a specific peer, datagrams can be sent to that peer using LPWSPSend and can be received from (only) this peer using LPWSPRecv.

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

A layered service provider supplies an implementation of this function, but it is also a client of this function if and when it calls **LPWSPSocket** of the next layer in the protocol chain. Some special considerations apply to this function's *lpProtocolInfo* parameter as it is propagated down through the layers of the protocol chain.

If the next layer in the protocol chain is another layer then when the next layer's **LPWSPSocket** is called, this layer must pass to the next layer a *lpProtocolInfo* that references the same unmodified WSAProtocol_Info structure with the same unmodified chain information. However, if the next layer is the base protocol (that is, the last element in the chain), this layer performs a substitution when calling the base provider's **LPWSPSocket**. In this case, the base provider's **WSAPROTOCOL_INFO** structure should be referenced by the *lpProtocolInfo* parameter.

One vital benefit of this policy is that base service providers do not have to be aware of protocol chains.

This same propagation policy applies when propagating a WSAProtocol_Info structure through a layered sequence of other functions such as LPWSPAddressToString, LPWSPDuplicateSocket, WSPStartup, or LPWSPStringToAddress.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	ws2spi.h

See also

- WPUCreateSocketHandle
- LPWSPAccept
- LPWSPBind
- LPWSPConnect
- LPWSPCloseSocket
- LPWSPGetSockName
- LPWSPGetSockOpt
- LPWSPIoctl
- LPWSPListen
- LPWSPRecv
- LPWSPRecvFrom
- LPWSPSend
- LPWSPSendTo
- LPWSPSetSockOpt
- LPWSPShutdown

LPWSPSTRINGTOADDRESS callback function (ws2spi.h)

10/22/2022 • 2 minutes to read • Edit Online

The **WSPStringToAddress** function converts a human-readable numeric string to a socket address structure (sockaddr) suitable to passing to Windows Sockets routines that take such a structure. Any missing components of the address are defaulted to a reasonable value, if possible. For example, a missing port number defaults to zero.

Syntax

```
LPWSPSTRINGTOADDRESS Lpwspstringtoaddress;

INT Lpwspstringtoaddress(
  [in]      LPWSTR AddressString,
  [in]      INT AddressFamily,
  [in]      LPWSAPROTOCOL_INFOW lpProtocolInfo,
  [out]     LPSOCKADDR lpAddress,
  [in, out] LPINT lpAddressLength,
  [out]     LPINT lpErrno
)
{...}
```

Parameters

`[in] AddressString`

Pointer to the zero-terminated, human-readable string to convert.

`[in] AddressFamily`

Address family to which the string belongs, or AF_UNSPEC if it is unknown.

`[in] lpProtocolInfo`

(required) Provider's WSAProtocol_Info structure.

`[out] lpAddress`

Buffer that is filled with a single sockaddr structure.

`[in, out] lpAddressLength`

Length of the Address buffer, in bytes. Returns the size of the resultant sockaddr structure. If the supplied buffer is not large enough, the function fails with a specific error of WSAEFAULT and this parameter is updated with the required size in bytes.

`[out] lpErrno`

Pointer to the error code.

Return value

If no error occurs, **WSPStringToAddress** returns zero. Otherwise, a value of SOCKET_ERROR is returned, and a

specific error code is available in *lpErrno*.

ERROR CODE	MEANING
WSAEFAULT	The specified address buffer is too small, pass in a larger buffer.
WSAEINVAL	Unable to translate the string into a sockaddr, or the provider was unable to support the indicated address family, or the specified *lpProtocolInfo* did not refer to a WSAProtocol_Info structure supported by the provider.

Remarks

A layered service provider supplies an implementation of this function, but it is also a client of this function if and when it calls **WSPStringToAddress** of the next layer in the protocol chain. Some special considerations apply to this function's *lpProtocolInfo* parameter as it is propagated down through the layers of the protocol chain.

If the next layer in the protocol chain is another layer, then when the next layer's **WSPStringToAddress** is called, this layer must pass to the next layer a *lpProtocolInfo* that references the same unmodified WSAProtocol_Info structure with the same unmodified chain information. However, if the next layer is the base protocol (that is, the last element in the chain), this layer performs a substitution when calling the base provider's **WSPStringToAddress**. In this case, the base provider's **WSAPROTOCOL_INFO** structure should be referenced by the *lpProtocolInfo* parameter.

One vital benefit of this policy is that base service providers do not have to be aware of protocol chains.

This same propagation policy applies when propagating a WSAProtocol_Info structure through a layered sequence of other functions such as LPWSPAddressToString, LPWSPDuplicateSocket, WSPStartup, or LPWSPSocket.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	ws2spi.h

See also

WSAProtocol_Info

LPWSPDuplicateSocket

LPWSPSocket

WSPStartup

sockaddr

NSP_ROUTINE structure (ws2spi.h)

10/22/2022 • 2 minutes to read • Edit Online

The **NSP_ROUTINE** structure contains information regarding the functions implemented by a namespace service provider version 1 (NSPv1) provider.

> **Note** The *Ws2spi.h* header file structure contains complete prototypes for all the NSPv1 function pointers.

Syntax

```
typedef struct _NSP_ROUTINE {
  DWORD                    cbSize;
  DWORD                    dwMajorVersion;
  DWORD                    dwMinorVersion;
  LPNSPCLEANUP             NSPCleanup;
  LPNSPLOOKUPSERVICEBEGIN  NSPLookupServiceBegin;
  LPNSPLOOKUPSERVICENEXT   NSPLookupServiceNext;
  LPNSPLOOKUPSERVICEEND    NSPLookupServiceEnd;
  LPNSPSETSERVICE          NSPSetService;
  LPNSPINSTALLSERVICECLASS NSPInstallServiceClass;
  LPNSPREMOVESERVICECLASS  NSPRemoveServiceClass;
  LPNSPGETSERVICECLASSINFO NSPGetServiceClassInfo;
  LPNSPIOCTL               NSPIoctl;
} NSP_ROUTINE, *LPNSP_ROUTINE;
```

Members

`cbSize`

Type: **DWORD**

The size, in bytes, of the structure. Note that the size of the **NSP_ROUTINE** structure changed on Windows XP and later.

`dwMajorVersion`

Type: **DWORD**

The major version of the service provider specification supported by this provider.

`dwMinorVersion`

Type: **DWORD**

The minor version of the service provider specification supported by this provider.

`NSPCleanup`

Type: **LPNSPCLEANUP**

A pointer to the NSPCleanup function implemented by the namespace provider. Every NSP function entry must point to a valid function. If the provider does not implement this function, the **NSPCleanup** function should return WSAEOPNOTSUPP.

`NSPLookupServiceBegin`

Type: **LPNSPLOOKUPSERVICEBEGIN**

A pointer to the NSPLookupServiceBegin function implemented by the namespace provider. Every NSP function entry must point to a valid function. If the provider does not implement this function, the **NSPLookupServiceBegin** function should return WSAEOPNOTSUPP.

`NSPLookupServiceNext`

Type: **LPNSPLOOKUPSERVICENEXT**

A pointer to the NSPLookupServiceNext function implemented by the namespace provider. Every NSP function entry must point to a valid function. If the provider does not implement this function, the **NSPLookupServiceNext** function should return WSAEOPNOTSUPP.

`NSPLookupServiceEnd`

Type: **LPNSPLOOKUPSERVICEEND**

A pointer to the NSPLookupServiceEnd function implemented by the namespace provider. Every NSP function entry must point to a valid function. If the provider does not implement this function, the **NSPLookupServiceEnd** function should return WSAEOPNOTSUPP.

`NSPSetService`

Type: **LPNSPSETSERVICE**

A pointer to the NSPSetService function implemented by the namespace provider. Every NSP function entry must point to a valid function. If the provider does not implement this function, the **NSPSetService** function should return WSAEOPNOTSUPP.

`NSPInstallServiceClass`

Type: **LPNSPINSTALLSERVICECLASS**

A pointer to the NSPInstallServiceClass function implemented by the namespace provider. Every NSP function entry must point to a valid function. If the provider does not implement this function, the **NSPInstallServiceClass** function should return WSAEOPNOTSUPP.

`NSPRemoveServiceClass`

Type: **LPNSPREMOVESERVICECLASS**

A pointer to the NSPRemoveServiceClass function implemented by the namespace provider. Every NSP function entry must point to a valid function. If the provider does not implement this function, the **NSPRemoveServiceClass** function should return WSAEOPNOTSUPP.

`NSPGetServiceClassInfo`

Type: **LPNSPGETSERVICECLASSINFO**

A pointer to the NSPGetServiceClassInfo function implemented by the namespace provider. Every NSP function entry must point to a valid function. If the provider does not implement this function, the **NSPGetServiceClassInfo** function should return WSAEOPNOTSUPP.

`NSPIoctl`

Type: **LPNSPIOCTL**

A pointer to the NSPIoctl function implemented by the namespace provider. Every NSP function entry must point to a valid function. If the provider does not implement this function, the **NSPIoctl** function should return

WSAEOPNOTSUPP.

> **Note** This structure member is only available on Windows XP and later.

Remarks

The size of the **NSP_ROUTINE** structure changed on Windows XP and later. The **cbSize** member should be used to determine which version of the **NSP_ROUTINE** structure is being used.

The version of the **NSP_ROUTINE** structure on Windows XP and later has the following new member added: **NSPIoctl**.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	ws2spi.h

See also

NSPCleanup

NSPGetServiceClassInfo

NSPInstallServiceClass

NSPIoctl

NSPLookupServiceBegin

NSPLookupServiceEnd

NSPLookupServiceNext

NSPRemoveServiceClass

NSPSetService

NSPStartup

NSPV2_ROUTINE

NSPStartup function (ws2spi.h)

10/22/2022 • 2 minutes to read • Edit Online

The **NSPStartup** function retrieves the dynamic information about a provider, such as the list of the DLL entry points.

This function is called by the client upon initialization. The **NSPStartup** and NSPCleanup functions must be called as pairs. All NSP functions must be called from within an **NSPStartup/NSPCleanup** pair. It is not required that WSC functions be called from within a **NSPStartup/NSPCleanup** pair.

Syntax

```
INT WSAAPI NSPStartup(
  [in]  LPGUID        lpProviderId,
  [out] LPNSP_ROUTINE lpnspRoutines
);
```

Parameters

`[in] lpProviderId`

The desired provider from which to return the entry points.

`[out] lpnspRoutines`

A pointer to an NSP_ROUTINE structure that points to provider entry points if the function call is successful.

Return value

The function should return **NO_ERROR** (zero) if the routine succeeds. It should return **SOCKET_ERROR** (–1) if the function fails and it must set the appropriate error code using WSASetLastError.

VALUE	DESCRIPTION
WSA_NOT_ENOUGH_MEMORY	There is not enough memory available to perform this operation.
WSAEINVAL	One or more parameters were invalid, or missing, for this provider.
WSAEINVALIDPROCTABLE	The procedure call table is invalid.
WSAEOPNOTSUPP	The operation is not supported. This error is returned if the namespace provider does not implement this function.
WSASYSNOTREADY	The NSPStartup function cannot operate at this time because the underlying system it uses to provide network services is currently unavailable.

Remarks

For more information, see the NSP_ROUTINE structure.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	ws2spi.h

See also

NSPCleanup

NSP_ROUTINE

WSASetLastError

NSPV2_ROUTINE structure (ws2spi.h)

10/22/2022 • 2 minutes to read • Edit Online

The **NSPV2_ROUTINE** structure contains information on the functions implemented by a namespace service provider version-2 (NSPv2) provider.

> **Note** The *Ws2spi.h* header file structure contains complete prototypes for all the NSPV2 function pointers.

Syntax

```
typedef struct _NSPV2_ROUTINE {
  DWORD                       cbSize;
  DWORD                       dwMajorVersion;
  DWORD                       dwMinorVersion;
  LPNSPV2STARTUP              NSPv2Startup;
  LPNSPV2CLEANUP              NSPv2Cleanup;
  LPNSPV2LOOKUPSERVICEBEGIN   NSPv2LookupServiceBegin;
  LPNSPV2LOOKUPSERVICENEXTEX  NSPv2LookupServiceNextEx;
  LPNSPV2LOOKUPSERVICEEND     NSPv2LookupServiceEnd;
  LPNSPV2SETSERVICEEX         NSPv2SetServiceEx;
  LPNSPV2CLIENTSESSIONRUNDOWN NSPv2ClientSessionRundown;
} NSPV2_ROUTINE, *PNSPV2_ROUTINE, *LPNSPV2_ROUTINE;
```

Members

`cbSize`

Type: **DWORD**

The size, in bytes, of the structure.

`dwMajorVersion`

Type: **DWORD**

The major version of the service provider specification supported by this provider.

`dwMinorVersion`

Type: **DWORD**

The minor version of the service provider specification supported by this provider.

`NSPv2Startup`

Type: ** LPNSPV2STARTUP**

A pointer to the NSPv2Startup function for this NSPv2 provider.

`NSPv2Cleanup`

Type: **LPNSPV2CLEANUP**

A pointer to the NSPv2Cleanup function for this NSPv2 provider.

`NSPv2LookupServiceBegin`

Type: **LPNSPV2LOOKUPSERVICEBEGIN**

A pointer to the NSPv2LookupServiceBegin function for this NSPv2 provider.

`NSPv2LookupServiceNextEx`

Type: **LPNSPV2LOOKUPSERVICENEXTEX**

A pointer to the NSPv2LookupServiceNextEx function for this NSPv2 provider.

`NSPv2LookupServiceEnd`

Type: **LPNSPV2LOOKUPSERVICEEND**

A pointer to the NSPv2LookupServiceEnd function for this NSPv2 provider.

`NSPv2SetServiceEx`

Type: **LPNSPV2SETSERVICEEX**

A pointer to the NSPv2SetServiceEx function for this NSPv2 provider.

`NSPv2ClientSessionRundown`

Type: **LPNSPV2CLIENTSESSIONRUNDOWN**

A pointer to the NSPv2ClientSessionRundown function for this NSPv2 provider.

Remarks

The **NSPV2_ROUTINE** structure is used as part of the namespace service provider version-2 (NSPv2) architecture available on Windows Vista and later.

On Windows Vista and Windows Server 2008, the **NSPV2_ROUTINE** structure can only be used for operations on NS_EMAIL namespace providers.

The WSAAdvertiseProvider function advertises an instance of a NSPv2 provider for clients to find. The **WSAAdvertiseProvider** caller passes a pointer to an **NSPV2_ROUTINE** structure in the *pNSPv2Routine* parameter with the NSPv2 entry points supported by the provider.

A NSPv2 provider is required to implement the following functions:

NSPv2LookupServiceBegin

- NSPv2LookupServiceNextEx

- NSPv2LookupServiceEnd

All other functions are optional, dependent on the requirements of the NSPv2 provider.

If a function isn't implemented, then calls to that function should be intercepted by a stub function that returns WSAEOPNOTSUPP. The NSPv2 function pointer to the unimplemented function in the **NSPV2_ROUTINE** structure should point be to the stub function.

In general, NSPv2 providers are implemented in processes other than the calling applications. NSPv2 providers are not activated as a result of client activity. Each provider hosting application decides when to make a specific provider available or unavailable by calling the WSAAdvertiseProvider and WSAUnadvertiseProvider functions. The client activity only results in attempts to contact the provider, when available (when the namespace provider is advertised).

A process can implement and advertise multiple providers at the same time. Windows Sockets will manage the namespace providers by dispatching calls to the correct one. It will also hide RPC interface details and translates cross-process calls into in-process calls. So that the NSPv2 provider has only to implement a table of entry point functions similar to the NSP_ROUTINE structure used by an NSPv1 provider. A NSPv2 provider does not have to worry about RPC specific requirements (data marshalling and serialization, for example).

The WSAUnadvertiseProvider function makes a specific namespace provider no longer available for clients.

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Header	ws2spi.h

See also

NSP_ROUTINE

NSPv2Cleanup

NSPv2ClientSessionRundown

NSPv2LookupServiceBegin

NSPv2LookupServiceEnd

NSPv2LookupServiceNextEx

NSPv2SetServiceEx

NSPv2Startup

WSAAdvertiseProvider

WSAProviderCompleteAsyncCall

WSAQUERYSET2

WSAUnadvertiseProvider

WPUCloseEvent function (ws2spi.h)

10/22/2022 • 2 minutes to read • Edit Online

The **WPUCloseEvent** function closes an open event object handle.

Syntax

```
BOOL WPUCloseEvent(
  [in]  WSAEVENT hEvent,
  [out] LPINT    lpErrno
);
```

Parameters

`[in] hEvent`

Handle to an open event object.

`[out] lpErrno`

Pointer to the error code.

Return value

If the function succeeds, the return value is **TRUE**. Otherwise, the return value is **FALSE** and a specific error code is available in *lpErrno*.

ERROR CODE	MEANING
WSA_INVALID_HANDLE	The *hEvent* parameter is not a valid event object handle.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	ws2spi.h

See also

WPUCreateEvent

WPUCloseSocketHandle function (ws2spi.h)

10/22/2022 • 2 minutes to read • Edit Online

The **WPUCloseSocketHandle** function closes an existing socket handle.

Syntax

```
int WPUCloseSocketHandle(
  [in]  SOCKET s,
  [out] LPINT  lpErrno
);
```

Parameters

`[in] s`

Handle to socket created with WPUCreateSocketHandle.

`[out] lpErrno`

Pointer to the error code.

Return value

If no error occurs, WPUCreateSocketHandle returns zero. Otherwise, it returns SOCKET_ERROR, and a specific error code is available in *lpErrno*.

ERROR CODE	MEANING
WSAENOTSOCK	The descriptor is not a socket created by WPUCreateSocketHandle.

Remarks

The **WPUCloseSocketHandle** function closes an existing socket handle created by WPUCreateSocketHandle. This function removes the socket from Ws2_32.dll's internal socket table. The owning service provider is responsible for releasing any resources associated with the socket.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows

Header	ws2spi.h

See also

WPUCreateSocketHandle

WPUCloseThread function (ws2spi.h)

10/22/2022 • 2 minutes to read • Edit Online

The **WPUCloseThread** function closes a thread opened with a call to WPUOpenCurrentThread.

Syntax

```
int WPUCloseThread(
  [in]  LPWSATHREADID lpThreadId,
  [out] LPINT         lpErrno
);
```

Parameters

`[in] lpThreadId`

Pointer to a WSATHREADID structure that identifies the thread context. This structure must have been initialized by a previous call to WPUOpenCurrentThread.

`[out] lpErrno`

Pointer to the error code.

Return value

If no error occurs, WPUOpenCurrentThread returns zero. Otherwise, it returns SOCKET_ERROR, and a specific error code is available in *lpErrno*.

ERROR CODE	MEANING
WSANOTINITIALISED	A successful WSPStartup call must occur before using this function.

Remarks

The **WPUCloseThread** function is used in a layered service provider to deallocate the resources that were initiated in a call by the WPUOpenCurrentThread function. The WSATHREADID structure in the *lpThreadId* is the thread to deallocate.

Every call to WPUOpenCurrentThread must have a call to **WPUCloseThread**. These two functions are used when the overlapped functions, such as LPWSPSend, are called in a lower layer of the service provider than the current thread.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	ws2spi.h

See also

WPUOpenCurrentThread

WSATHREADID

WPUCompleteOverlappedRequest function (ws2spi.h)

10/22/2022 • 4 minutes to read • Edit Online

The **WPUCompleteOverlappedRequest** function performs overlapped I/O completion notification for overlapped I/O operations.

Syntax

```
int WPUCompleteOverlappedRequest(
  [in]  SOCKET          s,
  [in]  LPWSAOVERLAPPED lpOverlapped,
  [in]  DWORD           dwError,
  [in]  DWORD           cbTransferred,
  [out] LPINT           lpErrno
);
```

Parameters

`[in] s`

The service provider socket created by WPUCreateSocketHandle.

`[in] lpOverlapped`

A pointer to a WSAOVERLAPPED structure associated with the overlapped I/O operation whose completion is to be notified.

`[in] dwError`

The completion status of the overlapped I/O operation whose completion is to be notified.

`[in] cbTransferred`

The number of bytes transferred to or from client buffers (the direction of the transfer depends on the send or receive nature of the overlapped I/O operation whose completion is to be notified).

`[out] lpErrno`

A pointer to the error code resulting from execution of this function.

Return value

If no error occurs, **WPUCompleteOverlappedRequest** returns zero and notifies completion of the overlapped I/O operation according to the mechanism selected by the client (signals an event found in the WSAOVERLAPPED structure referenced by *lpOverlapped* and/or queues a completion status report to the completion port associated with the socket if a completion port is associated). Otherwise, **WPUCompleteOverlappedRequest** returns SOCKET_ERROR, and a specific error code is available in *lpErrno*.

ERROR CODE	MEANING

WSAEINVAL	The socket passed in the *s* parameter is not a socket created by WPUCreateSocketHandle.

Remarks

The **WPUCompleteOverlappedRequest** function performs overlapped I/O completion notification for overlapped I/O operations where the client-specified completion mechanism is something other than user mode–asynchronous procedure call (APC). This function can only be used for socket handles created by WPUCreateSocketHandle.

> **Note** This function is different from other functions with the "WPU" prefix in that it is not accessed through the upcall table. Instead, it is exported directly by Ws2_32.dll. Service providers that need to call this function should link with WS2_32.lib or use appropriate operating system functions such as LoadLibrary and GetProcAddress to retrieve the function pointer.

The function **WPUCompleteOverlappedRequest** is used by service providers that do not implement Installable File System (IFS) functionality directly for the socket handles they expose. It performs completion notification for any overlapped I/O request for which the specified completion notification is other than a user-mode APC. **WPUCompleteOverlappedRequest** is supported only for the socket handles created by WPUCreateSocketHandle and not for sockets created by a service provider directly.

If the client selects a user-mode APC as the notification method, the service provider should use WPUQueueApc or another appropriate operating system function to perform the completion notification. If user-mode APC is not selected by the client, a service provider that does not implement IFS functionality directly cannot determine whether or not the client has associated a completion port with the socket handle. Thus, it cannot determine whether the completion notification method should be queuing a completion status record to a completion port or signaling an event found in the WSAOVERLAPPED structure. The Windows Socket 2 architecture keeps track of any completion port association with a socket created by WPUCreateSocketHandle and can make a correct decision between completion port-based notification or event-based notification.

When **WPUCompleteOverlappedRequest** queues a completion indication, it sets the **InternalHigh** member of the WSAOVERLAPPED structure to the count of bytes transferred. Then, it sets the **Internal** member to some OS-dependent value other than the special value WSS_OPERATION_IN_PROGRESS. There may be some slight delay after **WPUCompleteOverlappedRequest** returns before these values appear, since processing may occur asynchronously. However, the **InternalHigh** value (byte count) is guaranteed to be set by the time **Internal** is set.

WPUCompleteOverlappedRequest works as stated (performs the completion notification as requested by the client) whether or not the socket handle has been associated with a completion port.

Interaction with WSPGetOverlappedResult

The behavior of **WPUCompleteOverlappedRequest** places some constraints on how a service provider implements WSPGetOverlappedResult since only the **Offset** and **OffsetHigh** members of the WSAOVERLAPPED structure are exclusively controlled by the service provider, yet three values (byte count, flags, and error) must be retrieved from the structure by **WSPGetOverlappedResult**. A service provider may accomplish this any way it chooses as long as it interacts with the behavior of **WPUCompleteOverlappedRequest** properly. However, a typical implementation is as follows:

- At the start of overlapped processing, the service provider sets **Internal** to WSS_OPERATION_IN_PROGRESS.
- When the I/O operation has been completed, the provider sets **OffsetHigh** to the Windows Socket 2 error code resulting from the operation, sets **Offset** to the flags resulting from the I/O operation, and calls

WPUCompleteOverlappedRequest, passing the transfer byte count as one of the parameters. **WPUCompleteOverlappedRequest** eventually sets **InternalHigh** to the transfer byte count, then sets **Internal** to a value other than WSS_OPERATION_IN_PROGRESS.

- When WSPGetOverlappedResult is called, the service provider checks **Internal**. If it is WSS_OPERATION_IN_PROGRESS, the provider waits on the event handle in the **hEvent** member or returns an error, based on the setting of the FWAIT flag of **WSPGetOverlappedResult**. If not in progress, or after completion of waiting, the provider returns the values from **InternalHigh**, **OffsetHigh**, and **Offset** as the transfer count, operation result error code, and flags respectively.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	ws2spi.h

See also

WPUCreateSocketHandle

WPUQueueApc

WSAOVERLAPPED

WSPGetOverlappedResult

WPUCreateEvent function (ws2spi.h)

10/22/2022 • 2 minutes to read • Edit Online

The **WPUCreateEvent** function creates a new event object.

Syntax

```
WSAEVENT WPUCreateEvent(
  [out] LPINT lpErrno
);
```

Parameters

`[out] lpErrno`

Pointer to the error code.

Return value

If no error occurs, **WPUCreateEvent** function returns the handle of the event object.

Otherwise, the return value is WSA_INVALID_EVENT and a specific error code is available in *lpErrno*.

ERROR CODE	MEANING
WSA_NOT_ENOUGH_MEMORY	There is not enough free memory available to create the event object.

Remarks

The event object created by this function is manually reset with an initial state of nonsignaled. If a Windows service provider requires auto reset events, it can call the Windows CreateEvent function directly.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	ws2spi.h

See also

WPUCloseEvent

WPUCreateSocketHandle function (ws2spi.h)

10/22/2022 • 3 minutes to read • Edit Online

The **WPUCreateSocketHandle** function creates a new socket handle.

Syntax

```
SOCKET WPUCreateSocketHandle(
  [in]  DWORD     dwCatalogEntryId,
  [in]  DWORD_PTR dwContext,
  [out] LPINT     lpErrno
);
```

Parameters

`[in] dwCatalogEntryId`

Descriptor identifying the calling service provider.

`[in] dwContext`

Context value to associate with the new socket handle.

`[out] lpErrno`

Pointer to the error code.

Return value

If no error occurs, **WPUCreateSocketHandle** returns the new socket handle. Otherwise, it returns INVALID_SOCKET, and a specific error code is available in *lpErrno*.

ERROR CODE	MEANING
WSAENOBUFS	There are not enough buffers available to create the new socket handle.

Remarks

The **WPUCreateSocketHandle** function creates a new socket handle for the specified provider. The handles created by **WPUCreateSocketHandle** are indistinguishable from true file system handles. This is significant in two respects. First, the Windows Socket 2 architecture takes care of redirecting the file system functions ReadFile and WriteFile to this service provider's LPWSPRecv and LPWSPSend functions, respectively. Second, in operating systems that support completion ports, the Windows Sockets 2 architecture supports associating a completion port with the socket handle and using it to report overlapped I/O completion.

Note The mechanism for redirecting ReadFile and WriteFile necessarily involves a user-to-kernel transition to get to the redirector, followed by a kernel-to-user transition to get to [LPWSPRecv](nc-ws2spi-lpwsprecv.md) or [LPWSPSend](nc-ws2spi-lpwspsend.md). On return, these transitions are retraced in reverse. This can be a significant performance penalty. Any service provider that uses **WPUCreateSocketHandle** to create its socket handles should not set XP1_IFS_HANDLES in its WSAProtocol_Info structure. Clients should take the absence of XP1_IFS_HANDLES as guidance to avoid the use of **ReadFile** and **WriteFile**.

Note There is no exceptional performance penalty for using the completion port mechanism with socket handles created with **WPUCreateSocketHandle**. A service provider should use WPUCompleteOverlappedRequest to announce completion of overlapped I/O operations that may involve a completion port. Clients may freely use operating system functions to create, associate, and use a completion port for completion notification (for example, CreateIoCompletionPort, GetQueuedCompletionStatus, see the relevant operating system documentation for details). Note that completion ports are not integrated with the other asynchronous notification mechanisms offered by Windows Sockets 2. That is, a client can do a multiple-wait that includes multiple events and completion callbacks, but there is no predefined way for the multiple-wait to include completion ports.

Layered Service Provider Considerations

This procedure is of particular interest to Layered Service Providers. A layered service provider may use this procedure, instead of WPUModifyIFSHandle to create the socket handles it exposes to its client. The advantage of using this procedure is that all I/O requests involving the socket can be guaranteed to go through this service provider. This is true even if the client assumes that the sockets are file system handles and calls the file system functions ReadFile and WriteFile (although it would pay a performance penalty for this assumption).

The guarantee that all I/O goes through this layer is a requirement for layers that need to process the I/O stream either before or after the actual I/O operation. Creating socket handles using **WPUCreateSocketHandle** and specifying an appropriate service provider interface procedure dispatch table at the time of WSPStartup ensures that the layer has the chance to get involved in starting each I/O operation. When the client requests overlapped I/O operations, this service provider layer will usually have to arrange to get into the path of I/O completion notification as well.

To see why this is true, consider what happens if the client associates a completion port with the socket handle for the purpose of overlapped I/O completion notification. The port is associated with the socket handle exposed by this layer, not the next layer's socket handle. There is no way for this layer to determine if a completion port has been associated or what the port is. When this layer calls the next layer's I/O operation, it uses the next layer's socket handle. The next layer's socket handle will not have the same completion port association. The client's expected completion-port notification will not happen without some extra help.

The usual way a layered service provider takes care of this is to substitute a different overlapped I/O structure and different overlapped I/O parameters when invoking an I/O operation in the next layer. The substitute overlapped I/O structure references the client's stored overlapped structure and parameters. The invocation of the next layer sets up a callback notification. When the callback notification occurs, this layer performs any post-processing desired, retrieves the overlapped I/O information it stored on behalf of the client, discards the substitute structures, and forwards an appropriate completion notification to the client.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]

Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	ws2spi.h

See also

WPUCloseSocketHandle

WPUCompleteOverlappedRequest

WPUModifyIFSHandle

WPUQuerySocketHandleContext

LPWSPRecv

LPWSPSend

WSPStartup

WPUFDIsSet function (ws2spi.h)

10/22/2022 • 2 minutes to read • Edit Online

The **WPUFDIsSet** function checks the membership of the specified socket handle.

Syntax

```
int WPUFDIsSet(
  [in] SOCKET s,
  [in] fd_set *fdset
);
```

Parameters

`[in] s`

Descriptor identifying the socket.

`[in] fdset`

Set to check for the membership of socket *s*.

Return value

If no error occurs, a value of nonzero is returned denoting that socket *s* is a member of the *set* parameter. Otherwise, the return value is zero.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	ws2spi.h

See also

LPWSPSelect

WPUGetProviderPath function (ws2spi.h)

10/22/2022 • 2 minutes to read • Edit Online

The **WPUGetProviderPath** function retrieves the DLL path for the specified provider.

Syntax

```
int WPUGetProviderPath(
  [in]      LPGUID lpProviderId,
  [out]     WCHAR  *lpszProviderDllPath,
  [in, out] LPINT  lpProviderDllPathLen,
  [out]     LPINT  lpErrno
);
```

Parameters

`[in] lpProviderId`

Locally unique identifier of the provider. This must be a value obtained by using WSCEnumProtocols.

`[out] lpszProviderDllPath`

Pointer to a buffer containing a string that identifies the provider DLL's path. This path is a null-terminated string and any embedded environment strings (such as %SystemRoot%) have not been expanded.

`[in, out] lpProviderDllPathLen`

Size of the buffer pointed to by *lpszProviderDllPath*, in characters.

`[out] lpErrno`

Pointer to the error code.

Return value

If no error occurs, **WPUGetProviderPath** returns zero. Otherwise, it returns SOCKET_ERROR, and a specific error code is available in *lpErrno*.

ERROR CODE	MEANING
WSAEINVAL	The *lpProviderId* parameter does not specify a valid provider.
WSAEFAULT	Either *lpszProviderDllPath* or *lpErrno* is not in a valid part of the user address space, or *lpProviderDllPathLen* is too small.

Remarks

The **WPUGetProviderPath** function retrieves the DLL path for the specified provider. The DLL path is null-

terminated and may contain embedded environment strings (such as %SystemRoot%). Thus, the string should be expanded prior to being used with LoadLibrary. For more information, see **LoadLibrary**.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	ws2spi.h

See also

WSCEnumProtocols

WSCInstallProvider

WPUModifyIFSHandle function (ws2spi.h)

10/22/2022 • 2 minutes to read • Edit Online

The **WPUModifyIFSHandle** function receives a (possibly) modified IFS handle from Ws2_32.dll.

Syntax

```
SOCKET WPUModifyIFSHandle(
  [in]  DWORD  dwCatalogEntryId,
  [in]  SOCKET ProposedHandle,
  [out] LPINT  lpErrno
);
```

Parameters

`[in] dwCatalogEntryId`

Descriptor identifying the calling service provider.

`[in] ProposedHandle`

IFS handle allocated by the provider.

`[out] lpErrno`

Pointer to the error code.

Return value

If no error occurs, **WPUModifyIFSHandle** returns the modified socket handle. Otherwise, it returns INVALID_SOCKET, and a specific error code is available in *lpErrno*.

ERROR CODE	MEANING
WSAEINVAL	The proposed handle is invalid.

Remarks

The **WPUModifyIFSHandle** handle allows the WS2_32.dll to streamline its internal operations by returning a possibly modified version of the supplied IFS handle. The returned handle is guaranteed to be indistinguishable from the proposed handle as far as the operating system is concerned. IFS providers must call this function before returning any newly created socket descriptor to a service provider. The service provider will only use the modified handle for any subsequent socket operations. This routine is only used by IFS providers whose socket descriptors are real IFS handles.

Layered Service Provider Considerations

This procedure may also be used by a layered provider that is layered on top of a base IFS provider and wants to expose the base provider's socket handles as its own instead of creating them with a WPUCreateSocketHandle call. A layered provider that wishes to "pass through" the IFS socket handles it receives from the next layer in the chain can call **WPUModifyIFSHandle**, passing its own catalog entry ID as *dwCatalogEntryId*. This informs the Windows Sockets DLL that this layer, and not the next layer, should be the target for SPI calls involving that socket handle.

There are several limitations a layered provider should observe if it takes this approach.

- The provider should expose base provider entry points for LPWSPSend and LPWSPRecv in the procedure dispatch table it returns at the time of WSPStartup to make sure the Windows Sockets SPI client's access to these functions is as efficient as possible.
- The provider cannot rely on its LPWSPSend and LPWSPRecv functions being invoked for all I/O, particularly in the case of the I/O system functions ReadFile and WriteFile. These functions would bypass the layered provider and invoke the base IFS provider's implementation directly even if the layered provider puts its own entry points for these functions into the procedure dispatch table.
- The provider cannot rely on any ability to post-process overlapped I/O using LPWSPSend, LPWSPSendTo, LPWSPRecv, LPWSPRecvFrom, or LPWSPIoctl. Post-processing notification may happen through completion ports and bypass the layered provider entirely. A layered provider has no way to determine that a completion port was used or determine what port it is. The layered provider has no way to insert itself into the notification sequence.
- The provider should pass through all overlapped I/O requests directly to the base provider using the original overlapped parameters (for example, the WSAOVERLAPPED structure and completion routine pointer). The provider should expose the base provider entry point for WSPGetOverlappedResult. Since some overlapped I/O requests can bypass the layered provider completely, the layered provider cannot reliably mark **WSAOVERLAPPED** structures to determine which ones it can report results for, and which ones would have to be passed through to the underlying provider's **WSPGetOverlappedResult**. This effectively means that LPWSPIoctl has to be a pass-through operation to the underlying provider.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	ws2spi.h

See also

WPUCreateSocketHandle

WSAOVERLAPPED

WSPGetOverlappedResult

LPWSPIoctl

LPWSPRecv

LPWSPSend

WPUOpenCurrentThread function (ws2spi.h)

10/22/2022 • 2 minutes to read • Edit Online

The **WPUOpenCurrentThread** function opens a handle to the current thread that can be used with overlapped functions in a layered service provider. This is intended to be used by layered service providers that wish to initiate overlapped I/O from nonapplication threads.

Syntax

```
int WPUOpenCurrentThread(
  [out] LPWSATHREADID lpThreadId,
  [out] LPINT         lpErrno
);
```

Parameters

`[out] lpThreadId`

Pointer to a WSATHREADID structure that can then be passed to an overlapped function.

`[out] lpErrno`

Pointer to the error code.

Return value

If no error occurs, **WPUOpenCurrentThread** returns the zero. Otherwise, it returns SOCKET_ERROR, and a specific error code is available in *lpErrno*.

ERROR CODE	MEANING
WSANOTINITIALISED	A successful WSPStartup call must occur before using this function.

Remarks

The **WPUOpenCurrentThread** function provides a pointer to a WSATHREADID structure that can then be passed to an overlapped function such as LPWSPSend or LPWSPRecv. Layered service providers using a private thread in one of the upper layers will use **WPUOpenCurrentThread** to pass a **WSATHREADID** pointer to the lower layer that is administering overlapped functions.

Overlapped functions such as LPWSPSend and LPWSPRecv can then be used in the same way as a regular service provider.

Every call to **WPUOpenCurrentThread** must have a corresponding call to WPUCloseThread.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	ws2spi.h

See also

WPUCloseThread

LPWSPRecv

LPWSPSend

WPUPostMessage function (ws2spi.h)

10/22/2022 • 2 minutes to read • Edit Online

The **WPUPostMessage** function performs the standard Windows PostMessage function in a way that maintains backward compatibility with older versions of WSOCK32.dll.

Syntax

```
BOOL WPUPostMessage(
  [in] HWND   hWnd,
  [in] UINT   Msg,
  [in] WPARAM wParam,
  [in] LPARAM lParam
);
```

Parameters

`[in] hWnd`

Handle to the window that will receive the message.

`[in] Msg`

Message that will be posted.

`[in] wParam`

First parameter containing additional message-specific information.

`[in] lParam`

Second parameter containing additional message-specific information.

Return value

If no error occurs, **WPUPostMessage** returns the **TRUE** value. Otherwise, the **FALSE** value is returned.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	ws2spi.h

WPUQueryBlockingCallback function (ws2spi.h)

10/22/2022 • 2 minutes to read • Edit Online

The **WPUQueryBlockingCallback** function returns a pointer to a callback function the service provider should invoke periodically while servicing blocking operations.

Syntax

```
int WPUQueryBlockingCallback(
  [in]  DWORD              dwCatalogEntryId,
  [out] LPBLOCKINGCALLBACK *lplpfnCallback,
  [out] PDWORD_PTR         lpdwContext,
  [out] LPINT              lpErrno
);
```

Parameters

`[in] dwCatalogEntryId`

Descriptor identifying the calling service provider.

`[out] lplpfnCallback`

Pointer that receives the blocking callback function.

`[out] lpdwContext`

Pointer that receives a context value that the service provider must pass into the blocking callback.

`[out] lpErrno`

Pointer to the error code.

Return value

If no error occurs, **WPUQueryBlockingCallback** returns zero and stores a pointer to a blocking callback function in *lpfnCallback* and an associated context value in *lpdwContext*. Otherwise, it returns SOCKET_ERROR, and a specific error code is available in *lpErrno*.

ERROR CODE	MEANING
WSAEFAULT	The *lpfnCallback* or the *lpdwContext* parameter is not a valid part of the process address space.
WSAEINVAL	The *dwCatalogEntryId* parameter is invalid.

Remarks

The **WPUQueryBlockingCallback** function returns a pointer to a callback function in *lpfnCallback* to be invoked periodically during blocking operations. This function also returns a context value in *lpdwContext* to be passed into the blocking callback.

In Windows, this function can return null in *lpfnCallback*, indicating that no user defined–blocking hook is installed. In this case, the service provider should use the native Windows synchronization objects to implement blocking.

LPBLOCKINGCALLBACK is defined as follows:

```
typedef BOOL ( CALLBACK FAR * LPBLOCKINGCALLBACK )( DWORD dwContext );
```

The blocking callback will return **TRUE** if the service provider is to continue waiting for the blocking operation to complete. It will return **FALSE** if the blocking operation has been canceled with the WSPCancelBlockingCall.

Any missing components of the address will default to a reasonable value if possible. For example, a missing port number will default to zero.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	ws2spi.h

See also

WSPCancelBlockingCall

WPUQuerySocketHandleContext function (ws2spi.h)

10/22/2022 • 2 minutes to read • Edit Online

The **WPUQuerySocketHandleContext** function queries the context value associated with the specified socket handle.

Syntax

```
int WPUQuerySocketHandleContext(
  [in]  SOCKET     s,
  [out] PDWORD_PTR lpContext,
  [out] LPINT      lpErrno
);
```

Parameters

`[in] s`

Description that identifies the socket whose context is to be queried.

`[out] lpContext`

Pointer that will receive the context value.

`[out] lpErrno`

Pointer to the error code.

Return value

If no error occurs, **WPUQuerySocketHandleContext** returns zero and stores the current context value in *lpContext*. Otherwise, it returns SOCKET_ERROR, and a specific error code is available in *lpErrno*.

ERROR CODE	MEANING
WSAENOTSOCK	The descriptor is not a socket created by WPUCreateSocketHandle.

Remarks

The **WPUQuerySocketHandleContext** function queries the current context value associated with the specified socket handle. Service providers typically use this function to retrieve a pointer to provider-specific data associated with the socket. For example, a service provider can use the socket context to store a pointer to a structure containing the socket's state, local and remote transport addresses, and event objects for signaling network events.

Only non-IFS providers use this function, because IFS providers are not able to supply a context value.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	ws2spi.h

See also

WPUCreateSocketHandle

WPUQueueApc function (ws2spi.h)

10/22/2022 • 2 minutes to read • Edit Online

The **WPUQueueApc** function queues a user mode–asynchronous procedure call (APC) to the specified thread in order to facilitate invocation of overlapped I/O completion routines.

Syntax

```
int WPUQueueApc(
  [in]  LPWSATHREADID lpThreadId,
  [in]  LPWSAUSERAPC  lpfnUserApc,
  [in]  DWORD_PTR     dwContext,
  [out] LPINT         lpErrno
);
```

Parameters

`[in] lpThreadId`

Pointer to a WSATHREADID structure that identifies the thread context. A pointer to this structure is supplied to the service provider by the Ws2_32.dll as an input parameter to an overlapped operation. The provider should store the **WSATHREADID** structure locally and provide a pointer to this local store. The local copy of **WSATHREADID** is no longer needed once **WPUQueueApc** returns.

`[in] lpfnUserApc`

Pointer to the APC function to be called.

`[in] dwContext`

32-bit context value that is subsequently supplied as an input parameter to the APC function.

`[out] lpErrno`

Pointer to the error code.

Return value

If no error occurs, **WPUQueueApc** returns zero and queues the completion routine for the specified thread. Otherwise, it returns SOCKET_ERROR, and a specific error code is available in *lpErrno*.

ERROR CODE	MEANING
WSAEFAULT	The *dwThreadId* parameter does not specify a valid thread.

Remarks

This function queues an APC function against the specified thread. Under Windows, this will be done using a

user mode–asynchronous procedure call (APC). The APC will only execute when the specified thread is blocked in an alertable wait and a callback will be made directly. This call is safe for use within an interrupt context.

LPWSAUSERAPC is defined as follows:

```
typedef void ( CALLBACK FAR * LPWSAUSERAPC )( DWORD dwContext );
```

Because the APC mechanism supports only a single context value, *lpfnUserApc* itself cannot be the client specified–completion routine, which involves more parameters. The service provider must instead supply a pointer to its own APC function that uses the supplied *dwContext* 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 *dwContext* value and its own intermediate procedure as the target procedure *lpfnUserApc*. - 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.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	ws2spi.h

See also

WSATHREADID

LPWSPIoctl

LPWSPRecv

LPWSPRecvFrom

LPWSPSend

LPWSPSendTo

WPUResetEvent function (ws2spi.h)

10/22/2022 • 2 minutes to read • Edit Online

The **WPUResetEvent** function resets the state of the specified event object to nonsignaled. This call is safe for use within interrupt context.

Syntax

```
BOOL WPUResetEvent(
  [in]  WSAEVENT hEvent,
  [out] LPINT    lpErrno
);
```

Parameters

`[in] hEvent`

Handle that identifies an open event object.

`[out] lpErrno`

Pointer to the error code.

Return value

If no error occurs, the **WPUResetEvent** function returns the value **TRUE**. Otherwise, **FALSE** is returned, and a specific error code is available in *lpErrno*.

ERROR CODE	MEANING
WSA_INVALID_HANDLE	The *hEvent* parameter is not a valid event object handle.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	ws2spi.h

See also

WPUCloseEvent

WPUCreateEvent

WPUSetEvent

WPUSetEvent function (ws2spi.h)

10/22/2022 • 2 minutes to read • Edit Online

The **WPUSetEvent** function sets the state of the specified event object to signaled. This call is safe for use within interrupt context.

Syntax

```
BOOL WPUSetEvent(
  [in]  WSAEVENT hEvent,
  [out] LPINT    lpErrno
);
```

Parameters

`[in] hEvent`

Handle that identifies an open event object.

`[out] lpErrno`

Pointer to the error code.

Return value

If no error occurs, the **WPUSetEvent** function returns the value **TRUE**. Otherwise, **FALSE** is returned, and a specific error code is available in *lpErrno*.

ERROR CODE	MEANING
ERROR_INVALID_HANDLE	The *hEvent* parameter is not a valid event object handle.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	ws2spi.h

See also

WPUCloseEvent

WPUCreateEvent

WPUResetEvent

WSAAdvertiseProvider function (ws2spi.h)

10/22/2022 • 2 minutes to read • Edit Online

The **WSAAdvertiseProvider** function makes a specific namespace version-2 provider available for all eligible clients.

Syntax

```
INT WSAAPI WSAAdvertiseProvider(
  [in] const GUID             *puuidProviderId,
  [in] const LPCNSPV2_ROUTINE pNSPv2Routine
);
```

Parameters

`[in] puuidProviderId`

A pointer to the provider ID of the namespace provider to be advertised.

`[in] pNSPv2Routine`

A pointer to a **NSPV2_ROUTINE** structure with the namespace service provider version-2 entry points supported by the provider.

Return value

If no error occurs, WSAProviderCompleteAsyncCall returns zero.

If the function fails, the return value is SOCKET_ERROR. To get extended error information, call WSAGetLastError, which returns one of the following extended error values.

ERROR CODE	MEANING
WSA_NOT_ENOUGH_MEMORY	There was insufficient memory to perform the operation.
WSAEFAULT	An internal error occurred.
WSAEINVAL	A parameter was not valid. This error is returned if the *puuidProviderId* or *pNSPv2Routine* parameters were **NULL**. This error is also returned if the **NSPv2LookupServiceBegin**, **NSPv2LookupServiceNextEx**, or **NSPv2LookupServiceEnd** members of the **NSPV2_ROUTINE** structure pointed to by the *pNSPv2Routine* parameter are **NULL**. A namespace version-2 provider must at least support name resolution which this minimum set of functions.

WSAEINVALIDPROVIDER	The namespace provider could not be found for the specified *puuidProviderId* parameter.
WSANOTINITIALISED	The *Ws2_32.dll* has not been initialized. The application must first call WSAStartup before calling any Windows Sockets functions.

Remarks

The **WSAAdvertiseProvider** function is used as part of the namespace service provider version-2 (NSPv2) architecture available on Windows Vista and later.

On Windows Vista and Windows Server 2008, the **WSAAdvertiseProvider** function can only be used for operations on NS_EMAIL namespace providers.

The **WSAAdvertiseProvider** function advertises an instance of a NSPv2 provider for clients to find. If the instance to be advertised is an instance of an application-type provider (a namespace provider where the **dwProvideType** member of the NAPI_PROVIDER_INSTALLATION_BLOB structure is **ProviderType_Application**), the advertised provider instance will be visible to all the client processes running under the same user and in the same session as the caller of **WSAAdvertiseProvider**.

In general, NSPv2 providers are implemented in processes other than the calling applications. NSPv2 providers are not activated as a result of client activity. Each provider hosting application decides when to make a specific provider available or unavailable by calling the **WSAAdvertiseProvider** and WSAUnadvertiseProvider functions. The client activity only results in attempts to contact the provider, when available (when the namespace provider is advertised).

The **WSAAdvertiseProvider** function is called by any application that wants to make a specific provider available for all eligible clients (currently all the applications running with the same credentials as the hosting application, and in the same user session).

A process can implement and advertise multiple providers at the same time. Windows Sockets will manage the namespace providers by dispatching calls to the correct one. It will also hide RPC interface details and translates cross-process calls into in-process calls. So that the NSPv2 provider has only to implement a table of entry point functions similar to the NSP_ROUTINE structure used by an NSPv1 provider. A NSPv2 provider does not have to worry about RPC specific requirements (data marshalling and serialization, for example).

The **WSAAdvertiseProvider** caller passes a pointer to an NSPV2_ROUTINE structure in the *pNSPv2Routine* parameter with the NSPv2 entry points supported by the provider.

The WSAUnadvertiseProvider function makes a specific namespace provider no longer available for clients.

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows
Header	ws2spi.h

Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

NAPI_PROVIDER_INSTALLATION_BLOB

NAPI_PROVIDER_TYPE

NSPV2_ROUTINE

WSAEnumNameSpaceProviders

WSAEnumNameSpaceProvidersEx

WSAProviderCompleteAsyncCall

WSASetService

WSAUnadvertiseProvider

WSCEnumNameSpaceProvidersEx32

WSAProviderCompleteAsyncCall function (ws2spi.h)

10/22/2022 • 2 minutes to read • Edit Online

The **WSAProviderCompleteAsyncCall** function notifies a client when an asynchronous call to a namespace version-2 provider is completed.

Syntax

```
INT WSAAPI WSAProviderCompleteAsyncCall(
  HANDLE hAsyncCall,
  INT    iRetCode
);
```

Parameters

`hAsyncCall`

The handle passed to the asynchronous call being completed. This handle is passed by the client to the namespace version-2 provider in the asynchronous function call.

`iRetCode`

The return code for the asynchronous call to the namespace version-2 provider.

Return value

If no error occurs, **WSAProviderCompleteAsyncCall** returns zero.

If the function fails, the return value is SOCKET_ERROR. To get extended error information, call WSAGetLastError, which returns one of the following extended error values.

ERROR CODE	MEANING
WSA_NOT_ENOUGH_MEMORY	There was insufficient memory to perform the operation.
WSAEFAULT	An internal error occurred.
WSAEINVAL	A parameter was not valid. This error is returned if the *hAsyncCall* parameter was **NULL**.
WSANOTINITIALISED	The *Ws2_32.dll* has not been initialized. The application must first call WSAStartup before calling any Windows Sockets functions.

Remarks

The **WSAProviderCompleteAsyncCall** function is used as part of the namespace service provider version-2 (NSPv2) architecture available on Windows Vista and later.

On Windows Vista and Windows Server 2008, the WSAUnadvertiseProvider function can only be used for operations on NS_EMAIL namespace providers. Asynchronous calls to NSPv2 providers are not supported on Windows Vista and Windows Server 2008. So the **WSAProviderCompleteAsyncCall** is not currently applicable. This function is planned for use in later versions of Windows when asynchronous calls to namespace providers are supported.

In general, NSPv2 providers are implemented in processes other than the calling applications. NSPv2 providers are not activated as result of client activity. Each provider hosting application decides when to make a specific provider available or unavailable by calling the WSAAdvertiseProvider and WSAUnadvertiseProvider functions. The client activity only results in attempts to contact the provider, when available (when the namespace provider is advertised).

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows
Header	ws2spi.h
Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

NSPV2_ROUTINE

WSAAdvertiseProvider

WSAGetLastError

WSAUnadvertiseProvider

WSATHREADID structure (ws2spi.h)

10/22/2022 • 2 minutes to read • Edit Online

The **WSATHREADID** structure enables a provider to identify a thread on which asynchronous procedure calls (APCs) can be queued using the **WPUQueueApc** function.

Syntax

```
typedef struct _WSATHREADID {
  HANDLE    ThreadHandle;
  DWORD_PTR Reserved;
} WSATHREADID, *LPWSATHREADID;
```

Members

`ThreadHandle`

Handle to the thread ID.

`Reserved`

Reserved.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	ws2spi.h

See also

WPUQueueApc

LPWSPIoctl

LPWSPRecv

LPWSPSend

WSAUnadvertiseProvider function (ws2spi.h)

10/22/2022 • 2 minutes to read • Edit Online

The **WSAUnadvertiseProvider** function makes a specific namespace version-2 provider no longer available for clients.

Syntax

```
INT WSAAPI WSAUnadvertiseProvider(
  [in] const GUID *puuidProviderId
);
```

Parameters

`[in] puuidProviderId`

A pointer to the provider ID of the namespace provider.

Return value

If no error occurs, **WSAUnadvertiseProvider** returns zero. Otherwise, it returns **SOCKET_ERROR**, and a specific error code is available by calling WSAGetLastError.

ERROR CODE	MEANING
WSAEINVAL	A parameter was not valid. This error is returned if the *puuidProviderId* parameter was **NULL**.

Remarks

The **WSAUnadvertiseProvider** function is used as part of the namespace service provider version-2 (NSPv2) architecture available on Windows Vista and later.

On Windows Vista and Windows Server 2008, the **WSAUnadvertiseProvider** function can only be used for operations on NS_EMAIL namespace providers.

In general, NSPv2 providers are implemented in processes other than the calling applications. NSPv2 providers are not activated as result of client activity. Each provider hosting application decides when to make a specific provider available or unavailable by calling the WSAAdvertiseProvider and **WSAUnadvertiseProvider** functions. The client activity only results in attempts to contact the provider, when available (when the namespace provider is advertised).

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]

Target Platform	Windows
Header	ws2spi.h
Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

NSPV2_ROUTINE

WSAAdvertiseProvider

WSAEnumNameSpaceProviders

WSAEnumNameSpaceProvidersEx

WSASetService

WSC_PROVIDER_AUDIT_INFO structure (ws2spi.h)

10/22/2022 • 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 **WSC_PROVIDER_AUDIT_INFO** structure contains audit information for a layered service provider (LSP) entry in Windows Sockets 2.

Syntax

```
typedef struct _WSC_PROVIDER_AUDIT_INFO {
  DWORD RecordSize;
  PVOID Reserved;
} WSC_PROVIDER_AUDIT_INFO;
```

Members

`RecordSize`

The size, in bytes of this audit information record which includes this member.

`Reserved`

A reserved member for the audit information record.

Remarks

The **WSC_PROVIDER_AUDIT_INFO** structure is not currently used.

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Header	ws2spi.h

See also

WSCGetProviderInfo

WSCGetProviderInfo32

WSCSetProviderInfo

WSCSetProviderInfo32

WSC_PROVIDER_INFO_TYPE

WSC_PROVIDER_INFO_TYPE enumeration (ws2spi.h)

10/22/2022 • 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 Windows Sockets **WSC_PROVIDER_INFO_TYPE**enumeration type is used to specify the information class of a layered service protocol (LSP) in Windows Sockets 2.

Syntax

```
typedef enum _WSC_PROVIDER_INFO_TYPE {
  ProviderInfoLspCategories,
  ProviderInfoAudit
} WSC_PROVIDER_INFO_TYPE;
```

Constants

`ProviderInfoLspCategories`	The LSP category information for a protocol entry in a layered protocol. The information class should point to a DWORD value containing the appropriate LSP category flags implemented by LSP.
`ProviderInfoAudit`	The LSP class information for audit information for the LSP entry. The information class should point to a WSC_PROVIDER_AUDIT_INFO structure containing an audit record for the LSP.

Remarks

The WSC_PROVIDER_AUDIT_INFO structure is not currently used.

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Header	ws2spi.h

See also

WSCGetProviderInfo

WSCGetProviderInfo32

WSCSetProviderInfo

WSCSetProviderInfo32

WSC_PROVIDER_AUDIT_INFO

WSCDeinstallProvider function (ws2spi.h)

10/22/2022 • 2 minutes to read • Edit Online

The **WSCDeinstallProvider** function removes the specified transport provider from the system configuration database.

Syntax

```
int WSCDeinstallProvider(
  [in]  LPGUID lpProviderId,
  [out] LPINT  lpErrno
);
```

Parameters

`[in] lpProviderId`

A pointer to a globally unique identifier (GUID) for the provider. This value is stored within each WSAProtocol_Info structure.

`[out] lpErrno`

A pointer to the error code if the function fails.

Return value

If no error occurs, **WSCDeinstallProvider** returns zero. Otherwise, it returns **SOCKET_ERROR**, and a specific error code is available in *lpErrno*.

ERROR CODE	MEANING
WSAEINVAL	The *lpProviderId* parameter does not specify a valid provider.
WSAEFAULT	The *lpErrno* parameter is not in a valid part of the user address space.
WSANO_RECOVERY	A nonrecoverable error occurred. This error is returned under several conditions including the following: the user lacks the administrative privileges required to write to the Windows Sockets registry, or a failure occurred when opening a catalog entry.
WSA_NOT_ENOUGH_MEMORY	Insufficient memory was available. This error is returned when there is insufficient memory to allocate a new catalog entry.

Remarks

The **WSCDeinstallProvider** function removes the common Windows Sockets 2 configuration information for the specified provider. After this routine completes successfully, the configuration information stored in the

registry will be changed. However, any Ws2_32.dll instances currently in memory will not be able to recognize this change.

On success, **WSCDeinstallProvider** will attempt to alert all interested applications that have registered for notification of the change by calling WSAProviderConfigChange.

The **WSCDeinstallProvider** function can only be called by a user logged on as a member of the Administrators group. If **WSCDeinstallProvider** is called by a user that is not a member of the Administrators group, the function call will fail and **WSANO_RECOVERY** is returned in the *lpErrno* parameter.

For computers running Windows Vista or Windows Server 2008, this function can also fail because of user account control (UAC). If an application that contains this function 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 on Windows Vista or Windows Server 2008 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.

The caller of this function must remove any additional files or service provider–specific configuration information that is needed to completely uninstall the service provider.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	ws2spi.h
Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

WSAProviderConfigChange

WSCEnumProtocols

WSCInstallProvider

WSCDeinstallProvider32 function (ws2spi.h)

10/22/2022 • 2 minutes to read • Edit Online

The **WSCDeinstallProvider32** function removes the specified 32-bit transport provider from the system configuration database.

> **Note** This call enables a 64-bit process to manipulate the 32-bit Winsock catalog because WSCDeinstallProvider, on 64-bit computers, only manipulates the native 64-bit Windows Sockets catalog.

Syntax

```
int WSCDeinstallProvider32(
  [in]  LPGUID lpProviderId,
  [out] LPINT  lpErrno
);
```

Parameters

`[in] lpProviderId`

A pointer to a globally unique identifier (GUID) for the provider. This value is stored within each WSAProtocol_Info structure.

`[out] lpErrno`

A pointer to the error code if the function fails.

Return value

If no error occurs, **WSCDeinstallProvider32** returns zero. Otherwise, it returns **SOCKET_ERROR**, and a specific error code is available in *lpErrno*.

ERROR CODE	MEANING
WSAEINVAL	The *lpProviderId* parameter does not specify a valid provider.
WSAEFAULT	The *lpErrno* parameter is not in a valid part of the user address space.
WSANO_RECOVERY	A nonrecoverable error occurred. This error is returned under several conditions including the following: the user lacks the administrative privileges required to write to the Windows Sockets registry, or a failure occurred when opening a catalog entry.

WSA_NOT_ENOUGH_MEMORY	Insufficient memory was available. This error is returned when there is insufficient memory to allocate a new catalog entry.

Remarks

WSCDeinstallProvider32 is a strictly 32-bit version of WSCDeinstallProvider. On a 64-bit computer, all calls not specifically 32-bit (for example, all functions that do not end in "32") operate on the native 64-bit catalog. Processes that execute on a 64-bit computer must use the specific 32-bit function calls to operate on a strictly 32-bit catalog and preserve compatibility. The definitions and semantics of the specific 32-bit calls are the same as their native counterparts.

The **WSCDeinstallProvider32** function removes the common Windows Sockets 2 configuration information for the specified 32-bit provider. After this routine completes successfully, the configuration information stored in the registry will be changed. However, any Ws2_32.dll instances currently in memory will not be able to recognize this change.

On success, **WSCDeinstallProvider32** will attempt to alert all interested applications that have registered for notification of the change by calling WSAProviderConfigChange.

The **WSCDeinstallProvider32** function can only be called by a user logged on as a member of the Administrators group. If **WSCDeinstallProvider32** is called by a user that is not a member of the Administrators group, the function call will fail and **WSANO_RECOVERY** is returned in the *lpErrno* parameter.

For computers running Windows Vista or Windows Server 2008, this function can also fail because of user account control (UAC). If an application that contains this function 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 on Windows Vista or Windows Server 2008 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.

The caller of this function must remove any additional files or service provider–specific configuration information that is needed to completely uninstall the service provider.

Requirements

Minimum supported client	Windows Vista, Windows XP Professional x64 Edition [desktop apps only]
Minimum supported server	Windows Server 2008, Windows Server 2003 x64 Edition [desktop apps only]
Target Platform	Windows
Header	ws2spi.h
Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

WSAProviderConfigChange

WSCDeinstallProvider

WSCEnumProtocols32

WSCInstallProvider64_32

WSCEnableNSProvider function (ws2spi.h)

10/22/2022 • 2 minutes to read • Edit Online

The **WSCEnableNSProvider** function changes the state of a given namespace provider. It is intended to give the end-user the ability to change the state of the namespace providers.

Syntax

```
INT WSCEnableNSProvider(
  [in] LPGUID lpProviderId,
  [in] BOOL   fEnable
);
```

Parameters

`[in] lpProviderId`

A pointer to a globally unique identifier (GUID) for the namespace provider.

`[in] fEnable`

A Boolean value that, if **TRUE**, the provider is set to the active state. If **FALSE**, the provider is disabled and will not be available for query operations or service registration.

Return value

If no error occurs, the **WSCEnableNSProvider** function returns **NO_ERROR** (zero). Otherwise, it returns **SOCKET_ERROR** if the function fails, and you must retrieve the appropriate error code using the WSAGetLastError function.

ERROR CODE	MEANING
WSAEFAULT	The *lpProviderId* parameter points to memory that is not in a valid part of the user address space.
WSAEINVAL	The specified namespace provider identifier is invalid.
WSASYSCALLFAILURE	A system call that should never fail has failed.
WSA_NOT_ENOUGH_MEMORY	Insufficient memory was available. This error is returned when there is insufficient memory to allocate a new catalog entry.

Remarks

The **WSCEnableNSProvider** function is intended to be used to change the state of the namespace providers. An independent software vendor (ISV) should not normally de-activate another ISV namespace provider in order to activate its own. The choice should be left to the user.

The **WSCEnableNSProvider** function does not affect applications that are already running. Newly installed namespace providers will not be visible to applications nor will the changes in a namespace provider's activation state be visible. Applications launched after the call to **WSCEnableNSProvider** will see the changes.

The **WSCEnableNSProvider** function can only be called by a user logged on as a member of the Administrators group. If **WSCEnableNSProvider** is called by a user that is not a member of the Administrators group, the function call will fail.

For computers running Windows Vista or Windows Server 2008, this function can also fail because of user account control (UAC). If an application that contains this function 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 on Windows Vista or Windows Server 2008 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.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	ws2spi.h
Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

WSCEnumProtocols

WSCInstallNameSpace

WSCUnInstallNameSpace

WSCWriteNameSpaceOrder

WSCEnableNSProvider32 function (ws2spi.h)

10/22/2022 • 2 minutes to read • Edit Online

The **WSCEnableNSProvider32** function enables or disables a specified 32-bit namespace provider. It is intended to give the end-user the ability to change the state of the namespace providers.

> **Note** This call is a strictly 32-bit version of WSCEnableNSProvider for use on 64-bit computers. It is provided to allow 64-bit processes to access the 32-bit catalogs.

Syntax

```
INT WSCEnableNSProvider32(
  [in] LPGUID lpProviderId,
  [in] BOOL   fEnable
);
```

Parameters

`[in] lpProviderId`

A pointer to a globally unique identifier (GUID) for the namespace provider.

`[in] fEnable`

A Boolean value that, if **TRUE**, the namespace provider is set to the active state. If **FALSE**, the namespace provider is disabled and will not be available for query operations or service registration.

Return value

If no error occurs, the **WSCEnableNSProvider32** function returns **NO_ERROR** (zero). Otherwise, it returns **SOCKET_ERROR** if the function fails, and you must retrieve the appropriate error code using the WSAGetLastError function.

ERROR CODE	MEANING
WSAEFAULT	The *lpProviderId* parameter points to memory that is not in a valid part of the user address space.
WSAEINVAL	The specified namespace provider identifier is invalid.
WSASYSCALLFAILURE	A system call that should never fail has failed.
WSA_NOT_ENOUGH_MEMORY	Insufficient memory was available. This error is returned when there is insufficient memory to allocate a new catalog entry.

Remarks

The **WSCEnableNSProvider32** function is intended to be used to change the state of the namespace providers. An independent software vendor (ISV) should not normally de-activate another ISV's namespace provider in order to activate its own. The choice should be left to the user.

WSCEnableNSProvider32 is a strictly 32-bit version of WSCEnableNSProvider. On a 64-bit computer, all calls not specifically 32-bit (for example, all functions that do not end in "32") operate on the native 64-bit catalog. Processes that execute on a 64-bit computer must use the specific 32-bit function calls to operate on a strictly 32-bit catalog and preserve compatibility. The definitions and semantics of the specific 32-bit calls are the same as their native counterparts.

The namespace configuration functions do not affect applications that are already running. Newly installed namespace providers will not be visible to applications nor will the changes in a namespace provider's activation state. Applications launched after the call to **WSCEnableNSProvider32** will see the changes.

The **WSCEnableNSProvider32** function can only be called by a user logged on as a member of the Administrators group. If **WSCEnableNSProvider32** is called by a user that is not a member of the Administrators group, the function call will fail.

For computers running on Windows Vista or Windows Server 2008, this function can also fail because of user account control (UAC). If an application that contains this function 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 on Windows Vista or Windows Server 2008 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.

Requirements

Minimum supported client	Windows Vista, Windows XP Professional x64 Edition [desktop apps only]
Minimum supported server	Windows Server 2008, Windows Server 2003 x64 Edition [desktop apps only]
Target Platform	Windows
Header	ws2spi.h
Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

WSCEnableNSProvider

WSCEnumProtocols32

WSCInstallNameSpace32

WSCUnInstallNameSpace32

WSCWriteNameSpaceOrder32

WSCEnumNameSpaceProviders32 function (ws2spi.h)

10/22/2022 • 2 minutes to read • Edit Online

The **WSCEnumNameSpaceProviders32** function returns information on available 32-bit namespace providers.

> **Note** This call is a strictly 32-bit version of WSAEnumNameSpaceProviders for use on 64-bit platforms. It is provided to allow 64-bit processes to access the 32-bit catalogs.

Syntax

```
INT WSAAPI WSCEnumNameSpaceProviders32(
  [in, out] LPDWORD              lpdwBufferLength,
  [out]     LPWSANAMESPACE_INFOW lpnspBuffer
);
```

Parameters

`[in, out] lpdwBufferLength`

On input, the number of bytes contained in the buffer pointed to by *lpnspBuffer*. On output (if the function fails, and the error is WSAEFAULT), the minimum number of bytes to allocate for the *lpnspBuffer* buffer to allow it to retrieve all the requested information. The buffer passed to **WSCEnumNameSpaceProviders32** must be sufficient to hold all of the namespace information.

`[out] lpnspBuffer`

A buffer that is filled with WSANAMESPACE_INFOW structures. The returned structures are located consecutively at the head of the buffer. Variable sized information referenced by pointers in the structures point to locations within the buffer located between the end of the fixed sized structures and the end of the buffer. The number of structures filled in is the return value of **WSCEnumNameSpaceProviders32**.

Return value

The **WSCEnumNameSpaceProviders32** function returns the number of WSANAMESPACE_INFOW structures copied into *lpnspBuffer*. Otherwise, the value SOCKET_ERROR is returned, and a specific error number can be retrieved by calling WSAGetLastError.

ERROR CODE	MEANING
WSAEFAULT	The *lpnspBuffer* parameter was a **NULL** pointer or the buffer length, *lpdwBufferLength*, was too small to receive all the relevant WSANAMESPACE_INFOW structures and associated information. When this error is returned, the buffer length required is returned in the *lpdwBufferLength* parameter.

WSANOTINITIALISED	The WS2_32.DLL has not been initialized. The application must first call WSAStartup before calling any Windows Sockets functions.
WSA_NOT_ENOUGH_MEMORY	There was insufficient memory to perform the operation.

Remarks

WSCEnumNameSpaceProviders32 is a strictly 32-bit version of WSAEnumNameSpaceProviders. On a 64-bit computer, all calls not specifically 32-bit (for example, all functions that do not end in "32") operate on the native 64-bit catalog. Processes that execute on a 64-bit computer must use the specific 32-bit function calls to operate on a strictly 32-bit catalog and preserve compatibility. The definitions and semantics of the specific 32-bit calls are the same as their native counterparts.

The 32-bit SPI function is equivalent to the native API function (WSAEnumNameSpaceProviders) because there is no concept of a "hidden" namespace provider.

The **WSCEnumNameSpaceProviders32** function is a Unicode only function and returns WSANAMESPACE_INFOEXW structures.

Requirements

Minimum supported client	Windows Vista, Windows XP Professional x64 Edition [desktop apps only]
Minimum supported server	Windows Server 2008, Windows Server 2003 x64 Edition [desktop apps only]
Target Platform	Windows
Header	ws2spi.h
Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

WSAEnumNameSpaceProviders

WSANAMESPACE_INFOW

WSCInstallNameSpace32

WSCInstallNameSpaceEx32

WSCEnumNameSpaceProvidersEx32 function (ws2spi.h)

10/22/2022 • 2 minutes to read • Edit Online

The **WSCEnumNameSpaceProvidersEx32** function retrieves information on available 32-bit namespace providers.

Syntax

```
INT WSAAPI WSCEnumNameSpaceProvidersEx32(
  [in, out] LPDWORD                lpdwBufferLength,
  [out]     LPWSANAMESPACE_INFOEXW lpnspBuffer
);
```

Parameters

`[in, out] lpdwBufferLength`

On input, the number of bytes contained in the buffer pointed to by *lpnspBuffer*. On output (if the function fails, and the error is WSAEFAULT), the minimum number of bytes to allocate for the *lpnspBuffer* buffer to allow it to retrieve all the requested information. The buffer passed to **WSCEnumNameSpaceProvidersEx32** must be sufficient to hold all of the namespace information.

`[out] lpnspBuffer`

A buffer that is filled with WSANAMESPACE_INFOEXW structures. The returned structures are located consecutively at the head of the buffer. Variable sized information referenced by pointers in the structures point to locations within the buffer located between the end of the fixed sized structures and the end of the buffer. The number of structures filled in is the return value of **WSCEnumNameSpaceProvidersEx32**.

Return value

The **WSCEnumNameSpaceProvidersEx32** function returns the number of WSANAMESPACE_INFOEXW structures copied into *lpnspBuffer*. Otherwise, the value SOCKET_ERROR is returned, and a specific error number can be retrieved by calling WSAGetLastError.

ERROR CODE	MEANING
WSAEFAULT	The buffer length was too small to receive all the relevant WSANAMESPACE_INFOEXW structures and associated information or the *lpnspBuffer* parameter was a **NULL** pointer. When this error is returned, the buffer length required is returned in the *lpdwBufferLength* parameter.
WSANOTINITIALISED	The WS2_32.DLL has not been initialized. The application must first call WSAStartup before calling any Windows Sockets functions.
WSA_NOT_ENOUGH_MEMORY	There was insufficient memory to perform the operation.

Remarks

WSCEnumNameSpaceProvidersEx32 is a strictly 32-bit version of WSAEnumNameSpaceProvidersEx. On a 64-bit computer, all calls not specifically 32-bit (for example, all functions that do not end in "32") operate on the native 64-bit catalog. Processes that execute on a 64-bit computer must use the specific 32-bit function calls to operate on a strictly 32-bit catalog and preserve compatibility. The definitions and semantics of the specific 32-bit calls are the same as their native counterparts.

Currently, the only namespace included with Windows that uses information in the **ProviderSpecific** member of the WSANAMESPACE_INFOEXW structure are namespace providers for the NS_EMAIL namespace. The format of the **ProviderSpecific** member for an NS_EMAIL namespace provider is a NAPI_PROVIDER_INSTALLATION_BLOB structure.

The 32-bit SPI function is equivalent to the native API function (WSAEnumNameSpaceProvidersEx) because there is no concept of a "hidden" namespace provider.

The provider-specific data blob associated with namespace entry passed in the *lpProviderInfo* parameter to the WSCInstallNameSpaceEx32 function can be queried using **WSCEnumNameSpaceProvidersEx32** function.

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows
Header	ws2spi.h
Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

NAPI_PROVIDER_INSTALLATION_BLOB

WSAEnumNameSpaceProvidersEx

WSANAMESPACE_INFOEXW

WSCEnumNameSpaceProviders32

WSCInstallNameSpaceEx32

WSCEnumProtocols function (ws2spi.h)

10/22/2022 • 5 minutes to read • Edit Online

The **WSCEnumProtocols** function retrieves information about available transport protocols.

Syntax

```
int WSCEnumProtocols(
  [in]      LPINT               lpiProtocols,
  [out]     LPWSAPROTOCOL_INFOW lpProtocolBuffer,
  [in, out] LPDWORD             lpdwBufferLength,
  [out]     LPINT               lpErrno
);
```

Parameters

`[in] lpiProtocols`

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

`[out] lpProtocolBuffer`

A pointer to a buffer that is filled with WSAPROTOCOL_INFOW structures.

`[in, out] lpdwBufferLength`

On input, size of the *lpProtocolBuffer* buffer passed to **WSCEnumProtocols**, in bytes. On output, the minimum buffer size, in bytes, that can be passed to **WSCEnumProtocols** to retrieve all the requested information.

`[out] lpErrno`

A pointer to the error code.

Return value

If no error occurs, **WSCEnumProtocols** returns the number of protocols to be reported on. Otherwise, a value of SOCKET_ERROR is returned and a specific error code is available in *lpErrno*.

ERROR CODE	MEANING
WSAEFAULT	One of more of the arguments is not in a valid part of the user address space.
WSAEINVAL	Indicates that one of the specified parameters was invalid.
WSAENOBUFS	Buffer length was too small to receive all the relevant WSAProtocol_Info structures and associated information. Pass in a buffer at least as large as the value returned in *lpdwBufferLength*.

Remarks

The **WSCEnumProtocols** function is used to discover information about the collection of transport protocols installed on the local computer. This function differs from its API counterpart (WSAEnumProtocols) in that WSAPROTOCOL_INFOW structures for all installed protocols are returned. This includes protocols that the service provider has set the **PFL_HIDDEN** flag in the **dwProviderFlags** member of the **WSAPROTOCOL_INFOW** structure to indicate to the Ws2_32.dll that this protocol should not be returned in the result buffer generated by **WSAEnumProtocols** function. In addition, the **WSCEnumProtocols** also returns data for **WSAPROTOCOL_INFOW** structures that have a chain length of zero (a dummy LSP provider). The **WSAEnumProtocols** only returns information on base protocols and protocol chains that lack the **PFL_HIDDEN** flag and don't have a protocol chain length of zero.

> **Note** Layered Service Providers are deprecated. Starting with Windows 8 and Windows Server 2012, use Windows Filtering Platform.

The *lpiProtocols* parameter can be used as a filter to constrain the amount of information provided. Typically, a null pointer is supplied so the function will return information on all available transport protocols.

A WSAPROTOCOL_INFOW structure is provided in the buffer pointed to by *lpProtocolBuffer* for each requested protocol. If the supplied buffer is not large enough (as indicated by the input value of *lpdwBufferLength*), the value pointed to by *lpdwBufferLength* will be updated to indicate the required buffer size. The Windows Sockets SPI client should then obtain a large enough buffer and call this function again. The **WSCEnumProtocols** function cannot enumerate over multiple calls; the passed-in buffer must be large enough to hold all expected entries in order for the function to succeed. This reduces the complexity of the function and should not pose a problem because the number of protocols loaded on a local computer is typically small.

The order in which the WSAPROTOCOL_INFOW structures appear in the buffer coincides with the order in which the protocol entries were registered by the service provider with the WS2_32.dll, or with any subsequent reordering that may have occurred through the Windows Sockets applet supplied for establishing default transport providers.

Examples

The following example demonstrates the use of the **WSCEnumProtocols** function to retrieve an array of WSAPROTOCOL_INFOW structures for protocols installed on the local computer.

```
#ifndef UNICODE
#define UNICODE 1
#endif

#include <winsock2.h>
#include <ws2tcpip.h>
#include <ws2spi.h>
#include <objbase.h>
#include <stdio.h>

// Link with ws2_32.lib and ole32.lib
#pragma comment (lib, "Ws2_32.lib")
#pragma comment (lib, "ole32.lib")

#define MALLOC(x) HeapAlloc(GetProcessHeap(), 0, (x))
#define FREE(x) HeapFree(GetProcessHeap(), 0, (x))
// Note: could also use malloc() and free()

int wmain()
{

    //-----------------------------------------
    // Declare and initialize variables
```

```
    WSADATA wsaData;
    int iResult = 0;

    INT iNuminfo = 0;

    int i;

    // Allocate a 16K buffer to retrieve all the protocol providers
    DWORD dwBufferLen = 16384;
    LPWSAPROTOCOL_INFOW lpProtocolInfo = NULL;
    int iErrno = 0;

    // variables needed for converting provider GUID to a string
    int iRet = 0;
    WCHAR GuidString[40] = { 0 };

    // Initialize Winsock
    iResult = WSAStartup(MAKEWORD(2, 2), &wsaData);
    if (iResult != 0) {
        wprintf(L"WSAStartup failed: %d\n", iResult);
        return 1;
    }

    lpProtocolInfo = (LPWSAPROTOCOL_INFOW) MALLOC(dwBufferLen);
    if (lpProtocolInfo == NULL) {
        wprintf(L"Memory allocation for providers buffer failed\n");
        WSACleanup();
        return 1;
    }

    iNuminfo = WSCEnumProtocols(NULL, lpProtocolInfo, &dwBufferLen, &iErrno);
    if (iNuminfo == SOCKET_ERROR) {
        if (iErrno != WSAENOBUFS) {
            wprintf(L"WSCEnumProtocols failed with error: %d\n", iErrno);
            if (lpProtocolInfo) {
                FREE(lpProtocolInfo);
                lpProtocolInfo = NULL;
            }
            WSACleanup();
            return 1;
        } else {
            wprintf(L"WSCEnumProtocols failed with error: WSAENOBUFS (%d)\n",
                    iErrno);
            wprintf(L"  Increasing buffer size to %d\n\n", dwBufferLen);
            if (lpProtocolInfo) {
                FREE(lpProtocolInfo);
                lpProtocolInfo = NULL;
            }
            lpProtocolInfo = (LPWSAPROTOCOL_INFOW) MALLOC(dwBufferLen);
            if (lpProtocolInfo == NULL) {
                wprintf(L"Memory allocation increase for buffer failed\n");
                WSACleanup();
                return 1;
            }
            iNuminfo =
                WSCEnumProtocols(NULL, lpProtocolInfo, &dwBufferLen, &iErrno);
            if (iNuminfo == SOCKET_ERROR) {
                wprintf(L"WSCEnumProtocols failed with error: %d\n", iErrno);
                if (lpProtocolInfo) {
                    FREE(lpProtocolInfo);
                    lpProtocolInfo = NULL;
                }
                WSACleanup();
                return 1;
            }

        }
    }
```

```
    wprintf(L"WSCEnumProtocols succeeded with protocol count = %d\n\n",
            iNuminfo);
    for (i = 0; i < iNuminfo; i++) {
        wprintf(L"Winsock Catalog Provider Entry #%d\n", i);
        wprintf
            (L"-----------------------------------------------------------\n");
        wprintf(L"Entry type:\t\t\t ");
        if (lpProtocolInfo[i].ProtocolChain.ChainLen == 1)
            wprintf(L"Base Service Provider\n");
        else
            wprintf(L"Layered Chain Entry\n");

        wprintf(L"Protocol:\t\t\t %ws\n", lpProtocolInfo[i].szProtocol);

        iRet =
            StringFromGUID2(lpProtocolInfo[i].ProviderId,
                            (LPOLESTR) & GuidString, 39);
        if (iRet == 0)
            wprintf(L"StringFromGUID2 failed\n");
        else
            wprintf(L"Provider ID:\t\t\t %ws\n", GuidString);

        wprintf(L"Catalog Entry ID:\t\t %u\n",
                lpProtocolInfo[i].dwCatalogEntryId);

        wprintf(L"Version:\t\t\t %d\n", lpProtocolInfo[i].iVersion);

        wprintf(L"Address Family:\t\t\t %d\n",
                lpProtocolInfo[i].iAddressFamily);
        wprintf(L"Max Socket Address Length:\t %d\n",
                lpProtocolInfo[i].iMaxSockAddr);
        wprintf(L"Min Socket Address Length:\t %d\n",
                lpProtocolInfo[i].iMinSockAddr);

        wprintf(L"Socket Type:\t\t\t %d\n", lpProtocolInfo[i].iSocketType);
        wprintf(L"Socket Protocol:\t\t %d\n", lpProtocolInfo[i].iProtocol);
        wprintf(L"Socket Protocol Max Offset:\t %d\n",
                lpProtocolInfo[i].iProtocolMaxOffset);

        wprintf(L"Network Byte Order:\t\t %d\n",
                lpProtocolInfo[i].iNetworkByteOrder);
        wprintf(L"Security Scheme:\t\t %d\n",
                lpProtocolInfo[i].iSecurityScheme);
        wprintf(L"Max Message Size:\t\t %u\n", lpProtocolInfo[i].dwMessageSize);

        wprintf(L"ServiceFlags1:\t\t\t 0x%x\n",
                lpProtocolInfo[i].dwServiceFlags1);
        wprintf(L"ServiceFlags2:\t\t\t 0x%x\n",
                lpProtocolInfo[i].dwServiceFlags2);
        wprintf(L"ServiceFlags3:\t\t\t 0x%x\n",
                lpProtocolInfo[i].dwServiceFlags3);
        wprintf(L"ServiceFlags4:\t\t\t 0x%x\n",
                lpProtocolInfo[i].dwServiceFlags4);
        wprintf(L"ProviderFlags:\t\t\t 0x%x\n",
                lpProtocolInfo[i].dwProviderFlags);

        wprintf(L"Protocol Chain length:\t\t %d\n",
                lpProtocolInfo[i].ProtocolChain.ChainLen);

        wprintf(L"\n");
    }

    if (lpProtocolInfo) {
        FREE(lpProtocolInfo);
        lpProtocolInfo = NULL;
    }
    WSACleanup();

    return 0;
```

```
}
```

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	ws2spi.h
Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

WSAEnumProtocols

WSAPROTOCOL_INFOW

WSCEnumProtocols32

WSCEnumProtocols32 function (ws2spi.h)

10/22/2022 • 5 minutes to read • Edit Online

The **WSCEnumProtocols32** function retrieves information about available transport protocols.

> **Note** This call is a strictly 32-bit version of WSCEnumProtocols for use on 64-bit platforms. It is provided to allow 64-bit processes to access the 32-bit catalogs.

Syntax

```
int WSCEnumProtocols32(
  [in]      LPINT               lpiProtocols,
  [out]     LPWSAPROTOCOL_INFOW lpProtocolBuffer,
  [in, out] LPDWORD             lpdwBufferLength,
  [out]     LPINT               lpErrno
);
```

Parameters

`[in] lpiProtocols`

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

`[out] lpProtocolBuffer`

Buffer that is filled with WSAPROTOCOL_INFOW structures.

`[in, out] lpdwBufferLength`

On input, size of the *lpProtocolBuffer* buffer passed to WSCEnumProtocols, in bytes. On output, the minimum buffer size, in bytes, that can be passed to **WSCEnumProtocols** to retrieve all the requested information.

`[out] lpErrno`

Pointer to the error code.

Return value

If no error occurs, **WSCEnumProtocols32** returns the number of protocols to be reported on. Otherwise, a value of SOCKET_ERROR is returned and a specific error code is available in *lpErrno*.

ERROR CODE	MEANING
WSAEFAULT	One of more of the arguments is not in a valid part of the user address space.
WSAEINVAL	Indicates that one of the specified parameters was invalid.

WSAENOBUFS	Buffer length was too small to receive all the relevant WSAProtocol_Info structures and associated information. Pass in a buffer at least as large as the value returned in *lpdwBufferLength*.

Remarks

WSCEnumProtocols32 is a strictly 32-bit version of WSCEnumProtocols. On a 64-bit computer, all calls not specifically 32-bit (for example, all functions that do not end in "32") operate on the native 64-bit catalog. Processes that execute on a 64-bit computer must use the specific 32-bit function calls to operate on a strictly 32-bit catalog and preserve compatibility. The definitions and semantics of the specific 32-bit calls are the same as their native counterparts.

This function is used to discover information about the collection of transport protocols installed on the local computer. This function differs from its API counterpart (WSAEnumProtocols) in that WSAPROTOCOL_INFOW structures for all installed protocols are returned. This includes protocols that the service provider has set the **PFL_HIDDEN** flag in the **dwProviderFlags** member of the **WSAPROTOCOL_INFOW** structure to indicate to the Ws2_32.dll that this protocol should not be returned in the result buffer generated by **WSAEnumProtocols** function. In addition, the **WSCEnumProtocols32** also returns data for **WSAPROTOCOL_INFOW** structures that have a chain length of zero (a dummy LSP provider). The **WSAEnumProtocols** only returns information on base protocols and protocol chains that lack the **PFL_HIDDEN** flag and don't have a protocol chain length of zero.

> **Note** Layered Service Providers are deprecated. Starting with Windows 8 and Windows Server 2012, use Windows Filtering Platform.

The *lpiProtocols* parameter can be used as a filter to constrain the amount of information provided. Typically, a NULL pointer is supplied so the function will return information on all available transport protocols.

A WSAPROTOCOL_INFOW structure is provided in the buffer pointed to by *lpProtocolBuffer* for each requested protocol. If the supplied buffer is not large enough (as indicated by the input value of *lpdwBufferLength*), the value pointed to by *lpdwBufferLength* will be updated to indicate the required buffer size. The Windows Sockets SPI client should then obtain a large enough buffer and call this function again. The **WSCEnumProtocols32** function cannot enumerate over multiple calls; the passed-in buffer must be large enough to hold all expected entries in order for the function to succeed. This reduces the complexity of the function and should not pose a problem because the number of protocols loaded on a computer is typically small.

The order in which the WSAPROTOCOL_INFOW structures appear in the buffer coincides with the order in which the protocol entries were registered by the service provider with the WS2_32.dll, or with any subsequent reordering that may have occurred through the Windows Sockets applet supplied for establishing default transport providers.

Examples

The following example demonstrates the use of the **WSCEnumProtocols32** function for use on 64-bit platforms to retrieve an array of WSAPROTOCOL_INFOW structures for protocols installed on the local computer in the 32-bit catalog.

```
#ifndef UNICODE
#define UNICODE 1
#endif

#include <winsock2.h>
#include <ws2tcpip.h>
#include <ws2spi.h>
#include <objbase.h>
```

```
#include <objbase.h>
#include <stdio.h>

// Link with ws2_32.lib and ole32.lib
#pragma comment (lib, "Ws2_32.lib")
#pragma comment (lib, "ole32.lib")

#define MALLOC(x) HeapAlloc(GetProcessHeap(), 0, (x))
#define FREE(x) HeapFree(GetProcessHeap(), 0, (x))
// Note: could also use malloc() and free()

int wmain()
{

    //-----------------------------------------
    // Declare and initialize variables
    WSADATA wsaData;
    int iResult = 0;

    INT iNuminfo = 0;

    int i;

    // Allocate a 16K buffer to retrieve all the protocol providers
    DWORD dwBufferLen = 16384;
    LPWSAPROTOCOL_INFOW lpProtocolInfo = NULL;
    int iErrno = 0;

    // variables needed for converting provider GUID to a string
    int iRet = 0;
    WCHAR GuidString[40] = { 0 };

    // Initialize Winsock
    iResult = WSAStartup(MAKEWORD(2, 2), &wsaData);
    if (iResult != 0) {
        wprintf(L"WSAStartup failed: %d\n", iResult);
        return 1;
    }

    lpProtocolInfo = (LPWSAPROTOCOL_INFOW) MALLOC(dwBufferLen);
    if (lpProtocolInfo == NULL) {
        wprintf(L"Memory allocation for providers buffer failed\n");
        WSACleanup();
        return 1;
    }

    iNuminfo = WSCEnumProtocols32(NULL, lpProtocolInfo, &dwBufferLen, &iErrno);
    if (iNuminfo == SOCKET_ERROR) {
        if (iErrno != WSAENOBUFS) {
            wprintf(L"WSCEnumProtocols32 failed with error: %d\n", iErrno);
            if (lpProtocolInfo) {
                FREE(lpProtocolInfo);
                lpProtocolInfo = NULL;
            }
            WSACleanup();
            return 1;
        } else {
            wprintf(L"WSCEnumProtocols32 failed with error: WSAENOBUFS (%d)\n",
                    iErrno);
            wprintf(L"  Increasing buffer size to %d\n\n", dwBufferLen);
            if (lpProtocolInfo) {
                FREE(lpProtocolInfo);
                lpProtocolInfo = NULL;
            }
            lpProtocolInfo = (LPWSAPROTOCOL_INFOW) MALLOC(dwBufferLen);
            if (lpProtocolInfo == NULL) {
                wprintf(L"Memory allocation increase for buffer failed\n");
                WSACleanup();
                return 1;
            }
```

```
        }
        iNuminfo =
            WSCEnumProtocols32(NULL, lpProtocolInfo, &dwBufferLen, &iErrno);
        if (iNuminfo == SOCKET_ERROR) {
            wprintf(L"WSCEnumProtocols32 failed with error: %d\n", iErrno);
            if (lpProtocolInfo) {
                FREE(lpProtocolInfo);
                lpProtocolInfo = NULL;
            }
            WSACleanup();
            return 1;
        }

    }
}

wprintf(L"WSCEnumProtocols32 succeeded with protocol count = %d\n\n",
        iNuminfo);
for (i = 0; i < iNuminfo; i++) {
    wprintf(L"Winsock Catalog Provider Entry #%d\n", i);
    wprintf
        (L"----------------------------------------------------------\n");
    wprintf(L"Entry type:\t\t\t ");
    if (lpProtocolInfo[i].ProtocolChain.ChainLen = 1)
        wprintf(L"Base Service Provider\n");
    else
        wprintf(L"Layered Chain Entry\n");

    wprintf(L"Protocol:\t\t\t %ws\n", lpProtocolInfo[i].szProtocol);

    iRet =
        StringFromGUID2(lpProtocolInfo[i].ProviderId,
                        (LPOLESTR) & GuidString, 39);
    if (iRet == 0)
        wprintf(L"StringFromGUID2 failed\n");
    else
        wprintf(L"Provider ID:\t\t\t %ws\n", GuidString);

    wprintf(L"Catalog Entry ID:\t\t %u\n",
            lpProtocolInfo[i].dwCatalogEntryId);

    wprintf(L"Version:\t\t\t %d\n", lpProtocolInfo[i].iVersion);

    wprintf(L"Address Family:\t\t\t %d\n",
            lpProtocolInfo[i].iAddressFamily);
    wprintf(L"Max Socket Address Length:\t %d\n",
            lpProtocolInfo[i].iMaxSockAddr);
    wprintf(L"Min Socket Address Length:\t %d\n",
            lpProtocolInfo[i].iMinSockAddr);

    wprintf(L"Socket Type:\t\t\t %d\n", lpProtocolInfo[i].iSocketType);
    wprintf(L"Socket Protocol:\t\t %d\n", lpProtocolInfo[i].iProtocol);
    wprintf(L"Socket Protocol Max Offset:\t %d\n",
            lpProtocolInfo[i].iProtocolMaxOffset);

    wprintf(L"Network Byte Order:\t\t %d\n",
            lpProtocolInfo[i].iNetworkByteOrder);
    wprintf(L"Security Scheme:\t\t %d\n",
            lpProtocolInfo[i].iSecurityScheme);
    wprintf(L"Max Message Size:\t\t %u\n", lpProtocolInfo[i].dwMessageSize);

    wprintf(L"ServiceFlags1:\t\t\t 0x%x\n",
            lpProtocolInfo[i].dwServiceFlags1);
    wprintf(L"ServiceFlags2:\t\t\t 0x%x\n",
            lpProtocolInfo[i].dwServiceFlags2);
    wprintf(L"ServiceFlags3:\t\t\t 0x%x\n",
            lpProtocolInfo[i].dwServiceFlags3);
    wprintf(L"ServiceFlags4:\t\t\t 0x%x\n",
            lpProtocolInfo[i].dwServiceFlags4);
```

```
        wprintf(L"ProviderFlags:\t\t\t 0x%x\n",
                lpProtocolInfo[i].dwProviderFlags);

        wprintf(L"Protocol Chain length:\t\t %d\n",
                lpProtocolInfo[i].ProtocolChain.ChainLen);

        wprintf(L"\n");
    }

    if (lpProtocolInfo) {
        FREE(lpProtocolInfo);
        lpProtocolInfo = NULL;
    }
    WSACleanup();

    return 0;
}
```

Requirements

Minimum supported client	Windows Vista, Windows XP Professional x64 Edition [desktop apps only]
Minimum supported server	Windows Server 2008, Windows Server 2003 x64 Edition [desktop apps only]
Target Platform	Windows
Header	ws2spi.h
Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

WSAEnumProtocols

WSAPROTOCOL_INFOW

WSCEnumProtocols

WSCGetApplicationCategory function (ws2spi.h)

10/22/2022 • 6 minutes to read • Edit Online

Note Layered Service Providers are deprecated. Starting with Windows 8 and Windows Server 2012, use Windows Filtering Platform.

The **WSCGetApplicationCategory** function retrieves the layered service provider (LSP) categories associated with an application.

Syntax

```
int WSCGetApplicationCategory(
  [in]  LPCWSTR Path,
  [in]  DWORD   PathLength,
  [in]  LPCWSTR Extra,
  [in]  DWORD   ExtraLength,
  [out] DWORD   *pPermittedLspCategories,
  [out] LPINT   lpErrno
);
```

Parameters

`[in] Path`

A pointer to a Unicode string that contains the load path to the executable image for the application. This string observes the usual rules for path resolution and can contain embedded environment strings (such as *%SystemRoot%*).

`[in] PathLength`

The length, in characters, of the *Path* parameter. This length does not include the terminating **NULL**.

`[in] Extra`

A pointer to a Unicode string which represents the command line arguments used when starting the application specified in the *Path* parameter. The *Extra* parameter is used to distinguish between multiple, distinct instances of an application when launched with a consistent command line. This is to support different application categorizations for different instances of Svchost.exe or Rundll32.exe. If only the *Path* parameter is required and no command line arguments are needed to further distinguish between instances of an application, then the *Extra* parameter should be set to **NULL**.

`[in] ExtraLength`

The length, in characters, of the *Extra* parameter. This length does not include the terminating **NULL**.

`[out] pPermittedLspCategories`

A pointer to a DWORD value of permitted LSP categories which are permitted for all instances of this application. The application is identified by the combination of the values of the *Path* and *Extra* parameters.

`[out] lpErrno`

A pointer to the error code if the function fails.

Return value

If no error occurs, **WSCGetApplicationCategory** returns **ERROR_SUCCESS** (zero). Otherwise, it returns **SOCKET_ERROR**, and a specific error code is returned in the *lpErrno* parameter.

ERROR CODE	MEANING
WSAEFAULT	One or more of the arguments is not in a valid part of the user address space.
WSAEINVAL	One or more of the arguments are invalid.
WSASERVICE_NOT_FOUND	The service could not be found based on the *Path* and *Extra* parameters. The error can also be returned if the application you are querying does not exist in the registry. In this case, the error indicates that the application is not currently categorized.
WSANO_RECOVERY	A nonrecoverable error occurred. This error is returned under several conditions including the following: the user lacks the administrative privileges required to access the Winsock registry, or a failure occurred when opening a Winsock catalog entry or an application ID entry.

Remarks

WSCGetApplicationCategory is used to retrieve the LSP category flags associated with an application instance. Applications can determine which LSP behaviors are acceptable within the application's context. Therefore, by specifying permitted LSP categories, an application can permit only those layered service providers which implement acceptable behaviors to be loaded.

The *Extra* parameter is required when the command line is used to distinguish between different instances of an application or service hosted within the same executable. Each instance can have different application categorization needs. Svchost.exe and Rundll32.exe are two examples where the command line is required to differentiate between different process instances. For SvcHost.exe, the **-k <svcinstance>** switch defines the process instance.

For services, using the Service Name is not sufficient, since the Winsock Catalog is global to a given process, and a process may host several services.

Window sockets determine an application's identity and retrieves the permitted LSP categories during the first call to WSAStartup. This will be the set of permitted LSP categories for the duration of the application instance. Subsequent changes to the permitted LSP categories for a given application identity will not be picked up until the next instance of the application. The permitted LSP categories are not mutable during the lifetime of the application instance.

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 refers to 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.

During LSP initialization, the LSP must provide pointers to a number of Winsock SPI functions. These functions will be called during normal processing by the layer directly above the LSP (either another LSP or Ws2_32.DLL).

An LSP that implements an installable file system (IFS) can selectively choose to provide pointers to functions which are implemented by itself, or pass back the pointers provided by the layer directly below the LSP. Non-IFS LSPs, because they provide their own handles, must implement all of the Winsock SPI functions. This is because each SPI will require the LSP to map all of the socket handles it created to the socket handle of the lower provider (either another LSP or the base protocol).

However, all LSPs perform their specific work by doing extra processing on only a subset of the Winsock SPI functions.

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, 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
LSP_CRYPTO_COMPRESS	The LSP is a cryptography or data compression provider.
LSP_FIREWALL	The LSP is a firewall provider.
LSP_LOCAL_CACHE	The LSP is a local cache provider.
LSP_INBOUND_MODIFY	The LSP modifies inbound data.
LSP_INSPECTOR	The LSP inspects or filters data.
LSP_OUTBOUND_MODIFY	The LSP modifies outbound data.
LSP_PROXY	The LSP acts as a proxy and redirects packets.
LSP_REDIRECTOR	The LSP is a network redirector.
LSP_SYSTEM	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).

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows
Header	ws2spi.h
Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

Categorizing Layered Service Providers and Applications

WSAStartup

WSCGetProviderInfo

WSCGetProviderInfo32

WSCSetApplicationCategory

WSCSetProviderInfo

WSCSetProviderInfo32

WSCGetProviderInfo function (ws2spi.h)

10/22/2022 • 5 minutes to read • Edit Online

Note Layered Service Providers are deprecated. Starting with Windows 8 and Windows Server 2012, use Windows Filtering Platform.

The **WSCGetProviderInfo** function retrieves the data associated with an information class for a layered service provider (LSP).

Syntax

```
int WSCGetProviderInfo(
  [in]      LPGUID                 lpProviderId,
  [in]      WSC_PROVIDER_INFO_TYPE InfoType,
  [out]     PBYTE                  Info,
  [in, out] size_t                 *InfoSize,
  [in]      DWORD                  Flags,
  [out]     LPINT                  lpErrno
);
```

Parameters

`[in] lpProviderId`

A pointer to a globally unique identifier (GUID) for the provider.

`[in] InfoType`

The information class that is requested for this LSP protocol entry.

`[out] Info`

A pointer to a buffer to receive the information class data for the requested LSP protocol entry. If this parameter is **NULL**, then **WSCGetProviderInfo** returns failure and the size required for this buffer is returned in the *InfoSize* parameter.

`[in, out] InfoSize`

The size, in bytes, of the buffer pointed to by the *Info* parameter. If the Info parameter is **NULL**, then **WSCGetProviderInfo** returns failure and the *InfoSize* parameter will receive the size of the required buffer.

`[in] Flags`

The flags used to modify the behavior of the **WSCGetProviderInfo** function call.

`[out] lpErrno`

A pointer to the error code if the function fails.

Return value

If no error occurs, **WSCGetProviderInfo** returns **ERROR_SUCCESS** (zero). Otherwise, it returns

SOCKET_ERROR, and a specific error code is returned in the *lpErrno* parameter.

ERROR CODE	MEANING
ERROR_CALL_NOT_IMPLEMENTED	The call is not implemented. This error is returned if **ProviderInfoAudit** is specified in the *InfoType* parameter.
WSAEFAULT	One or more of the arguments is not in a valid part of the user address space.
WSAEINVAL	One or more of the arguments are invalid.
WSAEINVALIDPROVIDER	The protocol entry could not be found for the specified *lpProviderId*.
WSANO_RECOVERY	A nonrecoverable error occurred. This error is returned under several conditions including the following: the user lacks the administrative privileges required to access the Winsock registry, or a failure occurred when opening a Winsock catalog entry.
WSA_NOT_ENOUGH_MEMORY	Insufficient memory was available. This error is returned when there is insufficient memory to allocate a new catalog entry.

Remarks

WSCGetProviderInfo is used to retrieve information class data for a layered service provider. When the *InfoType* parameter is set to **ProviderInfoLspCategories**, on success **WSCGetProviderInfo** returns with the *Info* parameter set with appropriate LSP category flags implemented by the LSP.

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 refers to 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. A protocol chain would then be defined as one or more layered protocols strung together and anchored by a base protocol. A base protocol has the **ChainLen** member of the WSAProtocol_Info structure set to **BASE_PROTOCOL** which is defined to be 1. A layered protocol has the **ChainLen** member of the **WSAPROTOCOL_INFO** structure set to **LAYERED_PROTOCOL** which is defined to be zero. A protocol chain has the **ChainLen** member of the **WSAPROTOCOL_INFO** structure set to greater than 1.

During LSP initialization, the LSP must provide pointers to a number of Winsock SPI functions. These functions will be called during normal processing by the layer directly above the LSP (either another LSP or Ws2_32.DLL).

An LSP that implements an installable file system (IFS) can selectively choose to provide pointers to functions which are implemented by itself, or pass back the pointers provided by the layer directly below the LSP. Non-IFS LSPs, because they provide their own handles, must implement all of the Winsock SPI functions. This is because each SPI will require the LSP to map all of the socket handles it created to the socket handle of the lower provider (either another LSP or the base protocol).

However, all LSPs perform their specific work by doing extra processing on only a subset of the Winsock SPI functions.

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, 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 into which an LSP can be classified.

LSP CATEGORY	DESCRIPTION
LSP_CRYPTO_COMPRESS	The LSP is a cryptography or data compression provider.
LSP_FIREWALL	The LSP is a firewall provider.
LSP_LOCAL_CACHE	The LSP is a local cache provider.
LSP_INBOUND_MODIFY	The LSP modifies inbound data.
LSP_INSPECTOR	The LSP inspects or filters data.
LSP_OUTBOUND_MODIFY	The LSP modifies outbound data.
LSP_PROXY	The LSP acts as a proxy and redirects packets.
LSP_REDIRECTOR	The LSP is a network redirector.
LSP_SYSTEM	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).

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]

Target Platform	Windows
Header	ws2spi.h
Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

Categorizing Layered Service Providers and Applications

WSAProtocol_Info

WSCGetApplicationCategory

WSCSetApplicationCategory

WSCSetProviderInfo

WSC_PROVIDER_INFO_TYPE

WSCGetProviderInfo32 function (ws2spi.h)

10/22/2022 • 5 minutes to read • Edit Online

Note Layered Service Providers are deprecated. Starting with Windows 8 and Windows Server 2012, use Windows Filtering Platform.

The **WSCGetProviderInfo32** function retrieves the data associated with an information class for a 32-bit layered service provider (LSP).

Note This call is a strictly 32-bit version of WSCGetProviderInfo for use on 64-bit platforms. It is provided to allow 64-bit processes to access the 32-bit catalogs.

Syntax

```
int WSCGetProviderInfo32(
  [in]      LPGUID                 lpProviderId,
  [in]      WSC_PROVIDER_INFO_TYPE InfoType,
  [out]     PBYTE                  Info,
  [in, out] size_t                 *InfoSize,
  [in]      DWORD                  Flags,
  [out]     LPINT                  lpErrno
);
```

Parameters

`[in] lpProviderId`

A pointer to a globally unique identifier (GUID) for the provider.

`[in] InfoType`

The information class that is requested for this LSP protocol entry.

`[out] Info`

A pointer to a buffer to receive the information class data for the requested LSP protocol entry. If this parameter is **NULL**, then **WSCGetProviderInfo32** returns failure and the size required for this buffer is returned in the *InfoSize* parameter.

`[in, out] InfoSize`

The size, in bytes, of the buffer pointed to by the *Info* parameter. If the Info parameter is **NULL**, then **WSCGetProviderInfo32** returns failure and the *InfoSize* parameter will receive the size of the required buffer.

`[in] Flags`

The flags used to modify the behavior of the **WSCGetProviderInfo32** function call.

`[out] lpErrno`

A pointer to the error code if the function fails.

Return value

If no error occurs, **WSCGetProviderInfo32** returns **ERROR_SUCCESS** (zero). Otherwise, it returns **SOCKET_ERROR**, and a specific error code is returned in the *lpErrno* parameter.

ERROR CODE	MEANING
ERROR_CALL_NOT_IMPLEMENTED	The call is not implemented. This error is returned if **ProviderInfoAudit** is specified in the *InfoType* parameter.
WSAEFAULT	One or more of the arguments is not in a valid part of the user address space.
WSAEINVAL	One or more of the arguments are invalid.
WSAEINVALIDPROVIDER	The protocol entry could not be found for the specified *lpProviderId*.
WSANO_RECOVERY	A nonrecoverable error occurred. This error is returned under several conditions including the following: the user lacks the administrative privileges required to access the Winsock registry, or a failure occurred when opening a Winsock catalog entry.
WSA_NOT_ENOUGH_MEMORY	Insufficient memory was available. This error is returned when there is insufficient memory to allocate a new catalog entry.

Remarks

WSCGetProviderInfo32 is a strictly 32-bit version of WSCGetProviderInfo. On a 64-bit computer, all calls not specifically 32-bit (for example, all functions that do not end in "32") operate on the native 64-bit catalog. Processes that execute on a 64-bit computer must use the specific 32-bit function calls to operate on a strictly 32-bit catalog and preserve compatibility. The definitions and semantics of the specific 32-bit calls are the same as their native counterparts.

WSCGetProviderInfo32 is used to retrieve information class data for a protocol entry on a 32-bit layered service provider. When the *InfoType* parameter is set to **ProviderInfoLspCategories**, on success **WSCGetProviderInfo32** returns with the *Info* parameter set with appropriate LSP category flags implemented by 32-bit LSP.

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 refers to 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. A protocol chain would then be defined as one or more layered protocols strung together and anchored by a base protocol. A base protocol has the **ChainLen** member of the WSAProtocol_Info structure set to **BASE_PROTOCOL** which is defined to be 1. A layered

protocol has the **ChainLen** member of the **WSAPROTOCOL_INFO** structure set to **LAYERED_PROTOCOL** which is defined to be zero. A protocol chain has the **ChainLen** member of the **WSAPROTOCOL_INFO** structure set to greater than 1.

During LSP initialization, the LSP must provide pointers to a number of Winsock SPI functions. These functions will be called during normal processing by the layer directly above the LSP (either another LSP or Ws2_32.DLL).

An LSP that implements an installable file system (IFS) can selectively choose to provide pointers to functions which are implemented by itself, or pass back the pointers provided by the layer directly below the LSP. Non-IFS LSPs, because they provide their own handles, must implement all of the Winsock SPI functions. This is because each SPI will require the LSP to map all of the socket handles it created to the socket handle of the lower provider (either another LSP or the base protocol).

However, all LSPs perform their specific work by doing extra processing on only a subset of the Winsock SPI functions.

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, 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 into which an LSP can be classified.

LSP CATEGORY	DESCRIPTION
LSP_CRYPTO_COMPRESS	The LSP is a cryptography or data compression provider.
LSP_FIREWALL	The LSP is a firewall provider.
LSP_LOCAL_CACHE	The LSP is a local cache provider.
LSP_INBOUND_MODIFY	The LSP modifies inbound data.
LSP_INSPECTOR	The LSP inspects or filters data.
LSP_OUTBOUND_MODIFY	The LSP modifies outbound data.
LSP_PROXY	The LSP acts as a proxy and redirects packets.
LSP_REDIRECTOR	The LSP is a network redirector.
LSP_SYSTEM	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).

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows
Header	ws2spi.h
Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

Categorizing Layered Service Providers and Applications

WSAProtocol_Info

WSCGetApplicationCategory

WSCGetProviderInfo

WSCSetApplicationCategory

WSCSetProviderInfo32

WSC_PROVIDER_INFO_TYPE

WSCGetProviderPath function (ws2spi.h)

10/22/2022 • 2 minutes to read • Edit Online

The **WSCGetProviderPath** function retrieves the DLL path for the specified provider.

Syntax

```
int WSCGetProviderPath(
  [in]      LPGUID lpProviderId,
  [out]     WCHAR  *lpszProviderDllPath,
  [in, out] LPINT  lpProviderDllPathLen,
  [out]     LPINT  lpErrno
);
```

Parameters

`[in] lpProviderId`

A pointer to a globally unique identifier (GUID) for the provider. This value is obtained by using WSCEnumProtocols.

`[out] lpszProviderDllPath`

A pointer to a buffer into which the provider DLL's path string is returned. The path is a null-terminated string and any embedded environment strings, such as %SystemRoot%, have not been expanded.

`[in, out] lpProviderDllPathLen`

The size, in characters, of the buffer pointed to by the *lpszProviderDllPath* parameter.

`[out] lpErrno`

A pointer to the error code if the function fails.

Return value

If no error occurs, **WSCGetProviderPath** returns zero. Otherwise, it returns SOCKET_ERROR. The specific error code is available in *lpErrno*.

ERROR CODE	MEANING
WSAEINVAL	The *lpProviderId* parameter does not specify a valid provider.
WSAEFAULT	The *lpszProviderDllPath* or *lpErrno* parameter is not in a valid part of the user address space, or *lpProviderDllPathLen* is too small.

Remarks

The **WSCGetProviderPath** function retrieves the DLL path for the specified provider. The DLL path can contain embedded environment strings, such as %SystemRoot%, and thus should be expanded prior to being used with

the Windows LoadLibrary function. For more information, see **LoadLibrary**.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	ws2spi.h
Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

WSCEnumProtocols

WSCInstallProvider

WSCGetProviderPath32 function (ws2spi.h)

10/22/2022 • 2 minutes to read • Edit Online

The **WSCGetProviderPath32** function retrieves the DLL path for the specified 32-bit provider.

> **Note** This call is a strictly 32-bit version of WSCGetProviderPath for use on 64-bit platforms. It is provided to allow 64-bit processes to access the 32-bit catalogs.

Syntax

```
int WSCGetProviderPath32(
  [in]      LPGUID lpProviderId,
  [out]     WCHAR  *lpszProviderDllPath,
  [in, out] LPINT  lpProviderDllPathLen,
  [out]     LPINT  lpErrno
);
```

Parameters

`[in] lpProviderId`

Locally unique identifier of the provider. This value is obtained by using WSCEnumProtocols32.

`[out] lpszProviderDllPath`

Pointer to a buffer into which the provider DLL's path string is returned. The path is a null-terminated string and any embedded environment strings, such as %SystemRoot%, have not been expanded.

`[in, out] lpProviderDllPathLen`

Size of the buffer pointed to by the *lpszProviderDllPath* parameter, in characters.

`[out] lpErrno`

Pointer to the error code.

Return value

If no error occurs, **WSCGetProviderPath32** returns zero. Otherwise, it returns SOCKET_ERROR. The specific error code is available in *lpErrno*.

ERROR CODE	MEANING
WSAEINVAL	The *lpProviderId* parameter does not specify a valid provider.
WSAEFAULT	The *lpszProviderDllPath* or *lpErrno* parameter is not in a valid part of the user address space, or *lpProviderDllPathLen* is too small.

Remarks

WSCGetProviderPath32 is a strictly 32-bit version of WSCGetProviderPath. On a 64-bit computer, all calls not specifically 32-bit (for example, all functions that do not end in "32") operate on the native 64-bit catalog. Processes that execute on a 64-bit computer must use the specific 32-bit function calls to operate on a strictly 32-bit catalog and preserve compatibility. The definitions and semantics of the specific 32-bit calls are the same as their native counterparts.

The **WSCGetProviderPath32** function retrieves the DLL path for the specified provider. The DLL path can contain embedded environment strings, such as %SystemRoot%, and thus should be expanded prior to being used with the Windows LoadLibrary function. For more information, see **LoadLibrary**.

Requirements

Minimum supported client	Windows Vista, Windows XP Professional x64 Edition [desktop apps only]
Minimum supported server	Windows Server 2008, Windows Server 2003 x64 Edition [desktop apps only]
Target Platform	Windows
Header	ws2spi.h
Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

WSCEnumProtocols32

WSCInstallProvider64_32

WSCInstallNameSpace function (ws2spi.h)

10/22/2022 • 2 minutes to read • Edit Online

The **WSCInstallNameSpace** function installs a namespace provider. For providers that are able to support multiple namespaces, this function must be called for each namespace supported, and a unique provider identifier must be supplied each time.

Syntax

```
INT WSCInstallNameSpace(
  [in] LPWSTR lpszIdentifier,
  [in] LPWSTR lpszPathName,
  [in] DWORD  dwNameSpace,
  [in] DWORD  dwVersion,
  [in] LPGUID lpProviderId
);
```

Parameters

`[in] lpszIdentifier`

A pointer to a string that identifies the provider associated with the globally unique identifier (GUID) passed in the *lpProviderId* parameter.

`[in] lpszPathName`

A pointer to a Unicode string that contains the load path to the provider DLL. This string observes the usual rules for path resolution and can contain embedded environment strings (such as *%SystemRoot%*). Such environment strings are expanded when the Ws2_32.dll must subsequently load the provider DLL on behalf of an application. After any embedded environment strings are expanded, the Ws2_32.dll passes the resulting string to the LoadLibrary function which loads the provider into memory. For more information, see **LoadLibrary**.

`[in] dwNameSpace`

The namespace supported by this provider.

`[in] dwVersion`

The version number of the provider.

`[in] lpProviderId`

A pointer to a GUID for the provider. This GUID should be generated by Uuidgen.exe.

Return value

If no error occurs, the **WSCInstallNameSpace** function returns **NO_ERROR** (zero). Otherwise, it returns **SOCKET_ERROR** if the function fails, and you must retrieve the appropriate error code using the WSAGetLastError function.

ERROR CODE	MEANING

WSAEACCES	The calling routine does not have sufficient privileges to install a namespace.
WSAEINVAL	One or more of the arguments are invalid.
WSANO_RECOVERY	A nonrecoverable error occurred. This error is returned under several conditions including the following: the provider is already installed, the user lacks the administrative privileges required to write to the Winsock registry, or a failure occurred when creating or installing a catalog entry.
WSASYSCALLFAILURE	A system call that should never fail has failed.
WSA_NOT_ENOUGH_MEMORY	Insufficient memory was available. This error is returned when there is insufficient memory to allocate a new catalog entry.

Remarks

The namespace–configuration functions do not affect applications that are already running. Newly installed namespace providers will not be visible to applications nor will the changes in a namespace provider's activation state. Applications launched after the call to **WSCInstallNameSpace** will see the changes.

The **WSCInstallNameSpace** function can only be called by a user logged on as a member of the Administrators group. If **WSCInstallNameSpace** is called by a user that is not a member of the Administrators group, the function call will fail. For computers running on Windows Vista or Windows Server 2008, this function can also fail because of user account control (UAC). If an application that contains this function 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 on Windows Vista or Windows Server 2008 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.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	ws2spi.h
Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

WSCDeinstallProvider

WSCEnumProtocols

WSCInstallNameSpace32

WSCInstallNameSpaceEx

WSCUnInstallNameSpace

WSCInstallNameSpace32 function (ws2spi.h)

10/22/2022 • 3 minutes to read • Edit Online

The **WSCInstallNameSpace32** function installs a specified 32-bit namespace provider. For providers that are able to support multiple namespaces, this function must be called for each namespace supported, and a unique provider identifier must be supplied each time.

> **Note** This call is a strictly 32-bit version of WSCInstallNameSpace for use on 64-bit platforms. It is provided to allow 64-bit processes to access the 32-bit catalogs.

Syntax

```
INT WSCInstallNameSpace32(
  [in] LPWSTR lpszIdentifier,
  [in] LPWSTR lpszPathName,
  [in] DWORD  dwNameSpace,
  [in] DWORD  dwVersion,
  [in] LPGUID lpProviderId
);
```

Parameters

`[in] lpszIdentifier`

A pointer to a string that identifies the provider associated with the globally unique identifier (GUID) passed in the *lpProviderId* parameter.

`[in] lpszPathName`

A pointer to a string that contains the path to the provider's DLL image. The string observes the usual rules for path resolution: this path can contain embedded environment strings (such as %SystemRoot%). Such environment strings are expanded whenever the WS2_32.DLL must subsequently load the provider DLL on behalf of an application. After any embedded environment strings are expanded, the Ws2_32.dll passes the resulting string into the LoadLibrary function to load the provider into memory. For more information, see **LoadLibrary**.

`[in] dwNameSpace`

A descriptor that specifies the namespace supported by this provider.

`[in] dwVersion`

A descriptor that specifies the version number of the provider.

`[in] lpProviderId`

A unique identifier for this provider. This GUID should be generated by Uuidgen.exe.

Return value

If no error occurs, the **WSCInstallNameSpace32** function returns NO_ERROR (zero). Otherwise, it returns

SOCKET_ERROR if the function fails, and you must retrieve the appropriate error code using the WSAGetLastError function.

ERROR CODE	MEANING
WSAEACCES	The calling routine does not have sufficient privileges to install a namespace.
WSAEINVAL	One or more of the arguments are invalid.
WSANO_RECOVERY	A nonrecoverable error occurred. This error is returned under several conditions including the following: the provider is already installed, the user lacks the administrative privileges required to write to the Winsock registry, or a failure occurred when creating or installing a catalog entry.
WSASYSCALLFAILURE	A system call that should never fail has failed.
WSA_NOT_ENOUGH_MEMORY	Insufficient memory was available. This error is returned when there is insufficient memory to allocate a new catalog entry.

Remarks

WSCInstallNameSpace32 is a strictly 32-bit version of WSCInstallNameSpace. On a 64-bit computer, all calls not specifically 32-bit (for example, all functions that do not end in "32") operate on the native 64-bit catalog. Processes that execute on a 64-bit computer must use the specific 32-bit function calls to operate on a strictly 32-bit catalog and preserve compatibility. The definitions and semantics of the specific 32-bit calls are the same as their native counterparts.

The namespace configuration functions do not affect applications that are already running. Newly installed namespace providers will not be visible to applications nor will the changes in a namespace provider's activation state. Applications launched after the call to **WSCInstallNameSpace32** will recognize the changes.

The **WSCInstallNameSpace32** function can only be called by a user logged on as a member of the Administrators group. If **WSCInstallNameSpace32** is called by a user that is not a member of the Administrators group, the function call will fail. For computers running Windows Vista or Windows Server 2008, this function can also fail because of user account control (UAC). If an application that contains this function 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 on Windows Vista or Windows Server 2008 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.

Requirements

Minimum supported client	Windows Vista, Windows XP Professional x64 Edition [desktop apps only]

Minimum supported server	Windows Server 2008, Windows Server 2003 x64 Edition [desktop apps only]
Target Platform	Windows
Header	ws2spi.h
Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

WSCDeinstallProvider32

WSCEnumProtocols32

WSCInstallNameSpace

WSCInstallNameSpaceEx32

WSCUnInstallNameSpace32

WSCInstallNameSpaceEx function (ws2spi.h)

10/22/2022 • 3 minutes to read • Edit Online

The **WSCInstallNameSpaceEx** function installs a namespace provider. For providers that are able to support multiple namespaces, this function must be called for each namespace supported, and a unique provider identifier must be supplied each time.

Syntax

```
INT WSCInstallNameSpaceEx(
  [in] LPWSTR lpszIdentifier,
  [in] LPWSTR lpszPathName,
  [in] DWORD  dwNameSpace,
  [in] DWORD  dwVersion,
  [in] LPGUID lpProviderId,
  [in] LPBLOB lpProviderSpecific
);
```

Parameters

`[in] lpszIdentifier`

A pointer to a string that identifies the provider associated with the globally unique identifier (GUID) passed in the *lpProviderId* parameter.

`[in] lpszPathName`

A pointer to a Unicode string that contains the load path to the provider DLL. This string observes the usual rules for path resolution and can contain embedded environment strings (such as *%SystemRoot%*). Such environment strings are expanded when the Ws2_32.dll must subsequently load the provider DLL on behalf of an application. After any embedded environment strings are expanded, the Ws2_32.dll passes the resulting string to the LoadLibrary function which loads the provider into memory. For more information, see **LoadLibrary**.

`[in] dwNameSpace`

The namespace supported by this provider.

`[in] dwVersion`

The version number of the provider.

`[in] lpProviderId`

A pointer to a GUID for the provider. This GUID should be generated by Uuidgen.exe.

`[in] lpProviderSpecific`

A provider-specific data blob associated with namespace entry.

Return value

If no error occurs, the **WSCInstallNameSpaceEx** function returns **NO_ERROR** (zero). Otherwise, it returns **SOCKET_ERROR** if the function fails, and you must retrieve the appropriate error code using the

WSAGetLastError function.

ERROR CODE	MEANING
WSAEACCES	The calling routine does not have sufficient privileges to install a namespace.
WSAEINVAL	One or more of the arguments are invalid.
WSANO_RECOVERY	A nonrecoverable error occurred. This error is returned under several conditions including the following: the provider is already installed, the user lacks the administrative privileges required to write to the Winsock registry, or a failure occurred when creating or installing a catalog entry.
WSASYSCALLFAILURE	A system call that should never fail has failed.
WSA_NOT_ENOUGH_MEMORY	Insufficient memory was available. This error is returned when there is insufficient memory to allocate a new catalog entry.

Remarks

The namespace–configuration functions do not affect applications that are already running. Newly installed name-space providers will not be visible to applications nor will the changes in a name-space provider's activation state. Applications launched after the call to **WSCInstallNameSpaceEx** will see the changes.

The provider-specific data blob associated with namespace entry passed in the *lpProviderInfo* parameter can be queried using the WSAEnumNameSpaceProvidersEx function.

Currently, the only namespace provider included with Windows that uses the *lpProviderInfo* parameter is the NS_EMAIL provider. The format of the buffer pointed to by the *lpProviderInfo* parameter for an NS_EMAIL namespace provider is a NAPI_PROVIDER_INSTALLATION_BLOB structure.

The **WSCInstallNameSpaceEx** function can only be called by a user logged on as a member of the Administrators group. If **WSCInstallNameSpaceEx** is called by a user that is not a member of the Administrators group, the function call will fail. For computers running on Windows Vista or Windows Server 2008, this function can also fail because of user account control (UAC). If an application that contains this function 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 on Windows Vista or Windows Server 2008 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.

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]

Target Platform	Windows
Header	ws2spi.h
Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

NAPI_PROVIDER_INSTALLATION_BLOB

WSAEnumNameSpaceProviders

WSAEnumNameSpaceProvidersEx

WSCInstallNameSpace

WSCInstallNameSpaceEx32

WSCUnInstallNameSpace

WSCInstallNameSpaceEx32 function (ws2spi.h)

10/22/2022 • 3 minutes to read • Edit Online

The **WSCInstallNameSpaceEx32** function installs a specified 32-bit namespace provider. For providers that are able to support multiple names spaces, this function must be called for each namespace supported, and a unique provider identifier must be supplied each time.

> **Note** This call is a strictly 32-bit version of WSCInstallNameSpaceEx32 for use on 64-bit platforms. It is provided to allow 64-bit processes to access the 32-bit catalogs.

Syntax

```
INT WSCInstallNameSpaceEx32(
  [in] LPWSTR lpszIdentifier,
  [in] LPWSTR lpszPathName,
  [in] DWORD  dwNameSpace,
  [in] DWORD  dwVersion,
  [in] LPGUID lpProviderId,
  [in] LPBLOB lpProviderSpecific
);
```

Parameters

`[in] lpszIdentifier`

A pointer to a string that identifies the provider associated with the globally unique identifier (GUID) passed in the *lpProviderId* parameter.

`[in] lpszPathName`

A pointer to a Unicode string that contains the load path to the provider DLL. This string observes the usual rules for path resolution and can contain embedded environment strings (such as *%SystemRoot%*). Such environment strings are expanded when the Ws2_32.dll must subsequently load the provider DLL on behalf of an application. After any embedded environment strings are expanded, the Ws2_32.dll passes the resulting string to the LoadLibrary function which loads the provider into memory. For more information, see **LoadLibrary**.

`[in] dwNameSpace`

The namespace supported by this provider.

`[in] dwVersion`

The version number of the provider.

`[in] lpProviderId`

A pointer to a GUID for the provider. This GUID should be generated by Uuidgen.exe.

`[in] lpProviderSpecific`

A provider-specific data blob associated with namespace entry.

Return value

If no error occurs, the **WSCInstallNameSpaceEx32** function returns **NO_ERROR** (zero). Otherwise, it returns **SOCKET_ERROR** if the function fails, and you must retrieve the appropriate error code using the WSAGetLastError function.

ERROR CODE	MEANING
WSAEACCES	The calling routine does not have sufficient privileges to install a namespace.
WSAEINVAL	One or more of the arguments are invalid.
WSANO_RECOVERY	A nonrecoverable error occurred. This error is returned under several conditions including the following: the provider is already installed, the user lacks the administrative privileges required to write to the Winsock registry, or a failure occurred when creating or installing a catalog entry.
WSASYSCALLFAILURE	A system call that should never fail has failed.
WSA_NOT_ENOUGH_MEMORY	Insufficient memory was available. This error is returned when there is insufficient memory to allocate a new catalog entry.

Remarks

WSCInstallNameSpaceEx32 is a strictly 32-bit version of WSCInstallNameSpaceEx. On a 64-bit computer, all calls not specifically 32-bit (for example, all functions that do not end in "32") operate on the native 64-bit catalog. Processes that execute on a 64-bit computer must use the specific 32-bit function calls to operate on a strictly 32-bit catalog and preserve compatibility. The definitions and semantics of the specific 32-bit calls are the same as their native counterparts.

The namespace–configuration functions do not affect applications that are already running. Newly installed name-space providers will not be visible to applications nor will the changes in a name-space provider's activation state. Applications launched after the call to **WSCInstallNameSpaceEx32** will see the changes.

The provider-specific data blob associated with namespace entry passed in the *lpProviderInfo* parameter can be queried using WSCEnumNameSpaceProvidersEx32 function.

Currently, the only namespace provider included with Windows that uses the *lpProviderInfo* parameter is the NS_EMAIL provider. The format of the buffer pointed to by the *lpProviderInfo* parameter for an NS_EMAIL namespace provider is a NAPI_PROVIDER_INSTALLATION_BLOB structure.

The **WSCInstallNameSpaceEx32** function can only be called by a user logged on as a member of the Administrators group. If **WSCInstallNameSpaceEx32** is called by a user that is not a member of the Administrators group, the function call will fail. For computers running on Windows Vista or Windows Server 2008, this function can also fail because of user account control (UAC). If an application that contains this function 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 on Windows Vista or Windows

Server 2008 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.

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows
Header	ws2spi.h
Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

NAPI_PROVIDER_INSTALLATION_BLOB

WSCEnumNameSpaceProviders32

WSCEnumNameSpaceProvidersEx32

WSCInstallNameSpace32

WSCInstallNameSpaceEx

WSCUnInstallNameSpace32

WSCInstallProvider function (ws2spi.h)

10/22/2022 • 4 minutes to read • Edit Online

Note Layered Service Providers are deprecated. Starting with Windows 8 and Windows Server 2012, use Windows Filtering Platform.

The **WSCInstallProvider** function installs the specified transport provider into the system configuration database.

Syntax

```
int WSCInstallProvider(
  [in]  LPGUID                    lpProviderId,
  [in]  const WCHAR               *lpszProviderDllPath,
  [in]  const LPWSAPROTOCOL_INFOW lpProtocolInfoList,
  [in]  DWORD                     dwNumberOfEntries,
  [out] LPINT                     lpErrno
);
```

Parameters

`[in] lpProviderId`

A pointer to a globally unique identifier (GUID) for the provider.

`[in] lpszProviderDllPath`

A pointer to a Unicode string that contains the load path to the provider DLL. This string observes the usual rules for path resolution and can contain embedded environment strings (such as *%SystemRoot%*). Such environment strings are expanded when the Ws2_32.dll must subsequently load the provider DLL on behalf of an application. After any embedded environment strings are expanded, the Ws2_32.dll passes the resulting string to the LoadLibrary function which loads the provider into memory. For more information, see **LoadLibrary**.

`[in] lpProtocolInfoList`

A pointer to an array of WSAProtocol_Info structures. Each structure defines a protocol, address family, and socket type supported by the provider.

`[in] dwNumberOfEntries`

The number of entries in the *lpProtocolInfoList* array.

`[out] lpErrno`

A pointer to the error code if the function fails.

Return value

If **WSCInstallProvider** succeeds, it returns zero. Otherwise, it returns **SOCKET_ERROR**, and a specific error code is returned in the *lpErrno* parameter.

ERROR CODE	MEANING
WSAEFAULT	One or more of the arguments is not in a valid part of the user address space.
WSAEINVAL	One or more of the arguments are invalid.
WSAENOBUFS	Memory cannot be allocated for buffers.
WSANO_RECOVERY	A nonrecoverable error occurred. This error is returned under several conditions including the following: the provider is already installed, the user lacks the administrative privileges required to write to the Winsock registry, or a failure occurred when creating or installing a catalog entry.
WSASYSCALLFAILURE	A system call that should never fail has failed.
WSA_NOT_ENOUGH_MEMORY	Insufficient memory was available. This error is returned when there is insufficient memory to allocate a new catalog entry.

Remarks

WSCInstallProvider is used to install a single transport service provider. This routine creates the necessary common Windows Sockets 2 configuration information for the specified provider. It is applicable to base protocols, layered protocols, and protocol chains. If a layered service provider is being installed, then WSCInstallProviderAndChains should be used. **WSCInstallProviderAndChains** can install a layered protocol and one or more protocol chains with a single function call. To accomplish the same work using **WSCInstallProvider** would require multiple function calls.

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 would be a security layer that adds a 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 capable of performing data communications with a remote endpoint. 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. A base protocol has the **ChainLen** member of the WSAProtocol_Info structure set to **BASE_PROTOCOL** which is defined to be 1. A layered protocol has the **ChainLen** member of the **WSAPROTOCOL_INFO** structure set to **LAYERED_PROTOCOL** which is defined to be zero. A protocol chain has the **ChainLen** member of the **WSAPROTOCOL_INFO** structure set to greater than 1.

The *lpProtocolInfoList* parameter contains a list of protocol entries to install. Callers of **WSCInstallProvider** are responsible for setting up the proper protocol entries. The *lpProtocolInfoList* parameter must not be **NULL**.

Upon successful completion of this call, any subsequent calls to WSAEnumProtocols or WSCEnumProtocols will return the newly-created protocol entries. Be aware that in Windows environments, only instances of Ws_32.dll created by calling WSAStartup after the successful completion of **WSCInstallProvider** will include the new entries when **WSAEnumProtocols** and **WSCEnumProtocols** returns.

> **Note** The WSAEnumProtocols function does not enumerate a layered protocol entry while WSCEnumProtocols does.

On success, **WSCInstallProvider** will attempt to alert all interested applications that have registered for notification of the change by calling WSAProviderConfigChange.

The **WSCInstallProvider** function can only be called by a user logged on as a member of the Administrators group. If **WSCInstallProvider** is called by a user that is not a member of the Administrators group, the function call will fail and WSANO_RECOVERY is returned in the *lpErrno* parameter. For computers running Windows Vista or Windows Server 2008, this function can also fail because of user account control (UAC). If an application that contains this function 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 on Windows Vista or Windows Server 2008 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.

Any file installation or service provider-specific configuration must be performed by the caller.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	ws2spi.h
Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

WSAEnumProtocols

WSAProtocol_Info

WSAProviderConfigChange

WSAStartup

WSCDeinstallProvider

WSCEnumProtocols

WSCInstallProvider64_32 function (ws2spi.h)

10/22/2022 • 5 minutes to read • Edit Online

[**WSCInstallProvider64_32** is no longer available for use as of Windows Vista. Instead, use WSCInstallProvider or WSCInstallProviderAndChains.]

The **WSCInstallProvider64_32** function installs the specified transport service provider into the 32-bit and 64-bit system configuration databases on a 64-bit computer.

Syntax

```
int WSCInstallProvider64_32(
  [in]  LPGUID                    lpProviderId,
  [in]  const WCHAR               *lpszProviderDllPath,
  [in]  const LPWSAPROTOCOL_INFOW lpProtocolInfoList,
  [in]  DWORD                     dwNumberOfEntries,
  [out] LPINT                     lpErrno
);
```

Parameters

`[in] lpProviderId`

A pointer to a globally unique identifier (GUID) for the provider.

`[in] lpszProviderDllPath`

A pointer to a Unicode string that contains the load path to the provider 64-bit DLL. This string observes the usual rules for path resolution and can contain embedded environment strings (such as %SystemRoot%). Such environment strings are expanded when the Ws2_32.dll must subsequently load the provider DLL on behalf of an application. After any embedded environment strings are expanded, the Ws2_32.dll passes the resulting string to the LoadLibrary function which loads the provider into memory. For more information, see **LoadLibrary**.

`[in] lpProtocolInfoList`

A pointer to an array of WSAProtocol_Info structures. Each structure defines a protocol, address family, and socket type supported by the provider.

`[in] dwNumberOfEntries`

The number of entries in the *lpProtocolInfoList* array.

`[out] lpErrno`

A pointer to the error code if the function fails.

Return value

If **WSCInstallProvider64_32** succeeds, it returns zero. Otherwise, it returns **SOCKET_ERROR**, and a specific error code is returned in the *lpErrno* parameter.

ERROR CODE	MEANING

WSAEFAULT	One or more of the arguments is not in a valid part of the user address space.
WSAEINVAL	One or more of the arguments are invalid.
WSAENOBUFS	Memory could not be allocated for buffers.
WSANO_RECOVERY	A nonrecoverable error occurred. This error is returned under several conditions including the following: the provider is already installed, the user lacks the administrative privileges required to write to the Winsock registry, or a failure occurred when creating or installing a catalog entry.
WSASYSCALLFAILURE	A system call that should never fail has failed.
WSA_NOT_ENOUGH_MEMORY	Insufficient memory was available. This error is returned when there is insufficient memory to allocate a new catalog entry.

Remarks

WSCInstallProvider64_32 is a basic version of the WSCInstallProviderAndChains64_32 function that only installs a single transport service provider. **WSCInstallProvider64_32** can be used to install a base protocol, a layered protocol, or a protocol chain. If a layered service provider is being installed, then **WSCInstallProviderAndChains64_32** should be used because this function allows a layered protocol and one or more protocol chains to be installed with a single function call. To accomplish the same work using **WSCInstallProvider64_32** would require multiple function calls to install each service provider component.

Windows Sockets (Winsock) 2 accommodates the notion of a layered protocol. A layered protocol is one that implements only higher level communications functions while relying on an underlying transport stack for the actual exchange of data with a remote endpoint. An example of a layered protocol would be a security layer that adds a protocol to the connection establishment process 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. 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. A base protocol has the **ChainLen** member of the WSAProtocol_Info structure set to **BASE_PROTOCOL** which is defined to be 1. A layered protocol has the **ChainLen** member of the **WSAPROTOCOL_INFO** structure set to **LAYERED_PROTOCOL** which is defined to be zero. A protocol chain has the **ChainLen** member of the **WSAPROTOCOL_INFO** structure set to greater than 1.

WSCInstallProvider64_32 is the 64-bit version of WSCInstallProvider that installs the provider into both the 32-bit and 64-bit catalogs on 64-bit platforms. That is, on 64-bit platforms, two Winsock catalogs are maintained, and both 32-bit and 64-bit processes are able to load the transport provider installed with this function. On 64-bit platforms, **WSCInstallProvider** installs only to the 64-bit Winsock catalog.

On a 64-bit computer, all calls not specifically designed for 32-bit (for example, all functions that do not end in "32") operate on the native 64-bit catalog. Processes that execute on a 64-bit computer must use **WSCInstallProvider64_32** to operate on both the 32-bit catalog as well as the 64-bit catalog, preserving

compatibility. The definitions and semantics of the specific 32-bit calls are the same as their native counterparts.

This routine creates the necessary common Winsock 2 configuration information for the specified provider. It is applicable to base protocols, layered protocols, and protocol chains.

The *lpProtocolInfoList* parameter contains a list of protocol entries to install. Callers of **WSCInstallProvider64_32** are responsible for setting up the proper protocol entries. The *lpProtocolInfoList* parameter must not be **NULL**.

After this routine completes successfully, the protocol information provided in *lpProtocolInfoList* will be returned by WSAEnumProtocols, WSCEnumProtocols, or WSCEnumProtocols32. Be aware that in Windows, only instances of the Ws2_32.dll created by calling WSAStartup after a successful completion of this function will include the new entries in **WSAEnumProtocols**, **WSCEnumProtocols**, and **WSCEnumProtocols32**.

> **Note** The WSAEnumProtocols function does not enumerate a layered protocol entry while WSCEnumProtocols and WSCEnumProtocols32 do.

On success, **WSCInstallProvider64_32** will attempt to alert all interested applications that have registered for notification of the change by calling WSAProviderConfigChange.

The **WSCInstallProvider64_32** function can only be called by a user logged on as a member of the Administrators group. If **WSCInstallProvider64_32** is called by a user that is not a member of the Administrators group, the function call will fail and WSANO_RECOVERY is returned in the *lpErrno* parameter. For computers running Windows Vista or Windows Server 2008, this function can also fail because of user account control (UAC). If an application that contains this function 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 on Windows Vista or Windows Server 2008 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.

Any file installation or service provider-specific configuration must be performed by the calling application.

If the WSCInstallProvider or WSCInstallProviderAndChains function is used, the function must be called once to install the provider in the 32-bit catalog and once to install the provider in the 64-bit catalog on a 64-bit platform.

Requirements

Minimum supported client	Windows XP Professional x64 Edition [desktop apps only]
Minimum supported server	Windows Server 2003 x64 Edition [desktop apps only]
Target Platform	Windows
Header	ws2spi.h
Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

LoadLibrary

Transport Configuration and Installation

Transport Service Providers

WSAEnumProtocols

WSAProviderConfigChange

WSAStartup

WSCDeinstallProvider32

WSCEnumProtocols

WSCEnumProtocols32

WSCInstallProvider

WSCInstallProviderAndChains

WSCInstallProviderAndChains64_32

WSCInstallProviderAndChains function (ws2spi.h)

10/22/2022 • 6 minutes to read • Edit Online

Note Layered Service Providers are deprecated. Starting with Windows 8 and Windows Server 2012, use Windows Filtering Platform.

The **WSCInstallProviderAndChains** function installs the specified 32-bit transport provider as well as its specific protocol chains into the Winsock 2 system configuration database on a 32-bit computer. This function ensures that the protocol chains are ordered at the beginning of the transport provider configuration information, ensuring that a separate call to WSCWriteProviderOrder is not necessary.

Syntax

```
int WSCInstallProviderAndChains(
  [in]            LPGUID              lpProviderId,
  [in]            const LPWSTR        lpszProviderDllPath,
  [in]            const LPWSTR        lpszLspName,
  [in]            DWORD               dwServiceFlags,
  [in]            LPWSAPROTOCOL_INFOW lpProtocolInfoList,
  [in]            DWORD               dwNumberOfEntries,
  [out, optional] LPDWORD             lpdwCatalogEntryId,
  [out]           LPINT               lpErrno
);
```

Parameters

`[in] lpProviderId`

A pointer to a provider-specific, globally unique identifier (GUID).

`[in] lpszProviderDllPath`

A pointer to a Unicode string containing the load path to the provider's DLL. This string observes the usual rules for path resolution and can contain embedded environment strings (%SystemRoot%, for example). Such environment strings are expanded whenever *Ws2_32.dll* subsequently loads the provider DLL on behalf of an application. After any embedded environment strings are expanded, *Ws2_32.dll* passes the resulting string into the LoadLibrary function to load the provider into memory. For more information, see **LoadLibrary**.

`[in] lpszLspName`

A pointer to a Unicode string that contains the name of the socket provider.

`[in] dwServiceFlags`

The service flags for the type of "dummy" catalog entry to be created.

A dummy entry is a WSAProtocol_Info structure with the **ChainLen** member set to 0. The actual LSP catalog entry will reference the ID of this dummy entry in its **ProtocolChain** member.

The possible flags that can be set for this parameter are as follows:

VALUE	MEANING
XP1_IFS_HANDLES	The catalog entry is for an Installable File System (IFS) LSP, which returns IFS-specific socket handles. These handles are returned directly to the calling application. An IFS LSP cannot intercept the completion of Winsock calls, and does not have to have all Winsock functions implemented or available on it.

`[in] lpProtocolInfoList`

A pointer to an array of WSAProtocol_Info structures. Each structure defines a protocol, address family, and socket type supported by the provider. The members of the **WSAPROTOCOL_INFO** structure that are examined are **iProtocol**, **iAddressFamily**, and **iSocketType**.

`[in] dwNumberOfEntries`

The number of entries in the *lpProtocolInfoList* array.

`[out, optional] lpdwCatalogEntryId`

Receives a pointer to the newly-installed "dummy" entry for the transport provider in the Winsock 2 system configuration database. This ID is used to install the catalog entries for the LSP.

`[out] lpErrno`

A pointer that receives an error code generated by the call if the function fails.

Return value

If **WSCInstallProviderAndChains** succeeds, it returns zero. Otherwise, it returns **SOCKET_ERROR**, and a specific error code is returned in the *lpErrno* parameter.

ERROR CODE	MEANING
WSAEFAULT	One or more of the arguments is not in a valid part of the user address space.
WSAEINVAL	One or more of the arguments are invalid. This error is returned for the following conditions: the *lpProviderId* parameter is **NULL**, the *lpszProviderDllPath* parameter is invalid or the path length is too large (**MAX_PATH** was exceeded), the *lpszLspName* parameter is invalid or the name length is too large (**WSAPROTOCOL_LEN** is exceeded), the *lpProtocolInfoList* is set to a non-**NULL** and the *dwNumberOfEntries* parameter is zero, a duplicate provider ID or the layered service provider name already exist in the catalog, or a match cannot be found for the specified protocol, address family, and socket type.
WSAEINPROGRESS	A provider installation is already in progress.
WSAEINVALIDPROCTABLE	The provider is missing required functionality.

WSAENOBUFS	Memory cannot be allocated for buffers.
WSANO_RECOVERY	A nonrecoverable error occurred. This error is returned under several conditions including the following: the provider is already installed, the *lpProtocolInfoList* parameter was **NULL** and there was no base provider found, the maximum protocol chain length (**MAX_PROTOCOL_CHAIN**) was reached, the user lacks the administrative privileges required to write to the Winsock registry, or a failure occurred when creating or installing a catalog entry.
WSASYSCALLFAILURE	A system call that should never fail has failed.

Remarks

WSCInstallProviderAndChains is an enhanced version of the basic WSCInstallProvider function used to install a single transport service provider. If a layered service provider is being installed, then **WSCInstallProviderAndChains** should be used. **WSCInstallProviderAndChains** can install a layered protocol and one or more protocol chains with a single function call. To accomplish the same work using **WSCInstallProvider** would require multiple function calls.

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 would be a security layer that adds a 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 capable of performing data communications with a remote endpoint. 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. A base protocol has the **ChainLen** member of the WSAProtocol_Info structure set to **BASE_PROTOCOL** which is defined to be 1. A layered protocol has the **ChainLen** member of the **WSAPROTOCOL_INFO** structure set to **LAYERED_PROTOCOL** which is defined to be zero. A protocol chain has the **ChainLen** member of the **WSAPROTOCOL_INFO** structure set to greater than 1.

If *lpProtocolInfoList* is set to **NULL**, this function creates protocol chains where the provider is layered over the base protocol for each unique protocol type as defined by the address family, socket type, and protocol. This eliminates the creation of any inaccessible duplicate provider entries.

If *lpProtocolInfoList* is set to a non-**NULL** value, this function creates protocol chains by obtaining the top-most entry in the configuration information that matches the address family, socket type, and protocol from each element in the provided array. Again, only the address family, socket type, and protocol are considered; all other members and duplicates are ignored.

Upon successful completion of this call, any subsequent calls to WSAEnumProtocols or WSCEnumProtocols will return the newly-created protocol chain entries. Be aware that in Windows environments, only instances of *Ws_32.dll* created by calling WSAStartup after the successful completion of **WSCInstallProviderAndChains** will include the new entries when **WSAEnumProtocols** and **WSCEnumProtocols** returns.

> **Note** The WSAEnumProtocols function does not enumerate a layered protocol entry while WSCEnumProtocols does.

On success, **WSCInstallProviderAndChains** will attempt to alert all interested applications that have registered for notification of the change by calling WSAProviderConfigChange.

The **WSCInstallProviderAndChains** function can only be called by a user logged on as a member of the Administrators group. If **WSCInstallProviderAndChains** is called by a user that is not a member of the Administrators group, the function call will fail and WSANO_RECOVERY is returned in the *lpErrno* parameter. For computers running Windows Vista or Windows Server 2008, this function can also fail because of user account control (UAC). If an application that contains this function 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 on Windows Vista or Windows Server 2008 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.

Any file installation or provider-specific configuration must be performed by the calling application.

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows
Header	ws2spi.h
Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

LoadLibrary

Transport Configuration and Installation

Transport Service Providers

WSAEnumProtocols

WSAProviderConfigChange

WSAStartup

WSCEnumProtocols

WSCInstallProvider

WSCWriteProviderOrder

WSCInstallProviderAndChains64_32 function (ws2spi.h)

10/22/2022 • 9 minutes to read • Edit Online

The **WSCInstallProviderAndChains64_32** function installs the specified transport provider and its specific protocol chains into both the 32-bit and 64-bit Winsock 2 system configuration databases on a 64-bit computer. This function ensures that the protocol chains are ordered at the beginning of the transport provider configuration information, making a separate call to WSCWriteProviderOrder unnecessary.

Syntax

```
int WSCInstallProviderAndChains64_32(
  [in]            LPGUID              lpProviderId,
  [in]            const LPWSTR        lpszProviderDllPath,
  [in]            const LPWSTR        lpszProviderDllPath32,
  [in]            const LPWSTR        lpszLspName,
  [in]            DWORD               dwServiceFlags,
  [in]            LPWSAPROTOCOL_INFOW lpProtocolInfoList,
  [in]            DWORD               dwNumberOfEntries,
  [out, optional] LPDWORD             lpdwCatalogEntryId,
  [out]           LPINT               lpErrno
);
```

Parameters

`[in] lpProviderId`

A pointer to a globally unique identifier (GUID) for the provider.

`[in] lpszProviderDllPath`

A pointer to a Unicode string that contains the load path to the provider 64-bit DLL. This string observes the usual rules for path resolution and can contain embedded environment strings (such as *%SystemRoot%*). Such environment strings are expanded when the *Ws2_32.dll* must subsequently load the provider DLL on behalf of an application. After any embedded environment strings are expanded, the *Ws2_32.dll* passes the resulting string to the LoadLibrary function which loads the provider into memory. For more information, see **LoadLibrary**.

> **Note** If *lpszProviderDllPath32* is set to **NULL** then *lpszProviderDllPath* must be set to *%windir%\system32\<dllname>*. If not, the call fails and returns the WSAEINVAL error code.

`[in] lpszProviderDllPath32`

A pointer to a Unicode string that contains the fully qualified path to the provider 32-bit DLL. This string observes the usual rules for path resolution and can contain embedded environment strings (such as *%SystemRoot%*). Such environment strings are expanded when the *Ws2_32.dll* subsequently loads the provider DLL on behalf of an application. After any embedded environment strings are expanded, the *Ws2_32.dll* passes the resulting string into the LoadLibrary function to load the provider into memory. For more information, see **LoadLibrary**.

> **Note** If this parameter is set to **NULL**, the 64-bit provider must exist in the *%windir%\system32* folder and the 32-bit provider must reside in the *%windir%\syswow64* folder.

`[in] lpszLspName`

A pointer to a Unicode string that contains the name of the layered service provider (LSP).

`[in] dwServiceFlags`

The service flags for the type of layered protocol catalog entry to be created. A layered protocol entry is a WSAProtocol_Info structure with the **ChainLen** member set to 0. The actual catalog entry for the LSP will reference the ID of this layered protocol entry in its **ProtocolChain** member.

VALUE	MEANING
XP1_IFS_HANDLES	The catalog entry is for an Installable File System (IFS) LSP, which returns IFS-specific socket handles. An IFS LSP cannot intercept the completion of Winsock calls, and does not have to implement all of the Winsock functions.

`[in] lpProtocolInfoList`

A pointer to an array of WSAProtocol_Info structures. Each structure defines a protocol, address family, and socket type supported by the provider. The members of the **WSAPROTOCOL_INFO** structure that are examined are **iProtocol**, **iAddressFamily**, and **iSocketType**.

`[in] dwNumberOfEntries`

The number of entries in the *lpProtocolInfoList* array.

`[out, optional] lpdwCatalogEntryId`

A pointer to the newly installed layered protocol entry for the transport provider in the Winsock 2 system configuration database. This was the ID used to build the protocol chain entries installed in the catalog for the LSP.

`[out] lpErrno`

A pointer to the error code generated by the call if the function fails.

Return value

If **WSCInstallProviderAndChains64_32** succeeds, it returns zero. Otherwise, it returns **SOCKET_ERROR**, and a specific error code is returned in the *lpErrno* parameter.

ERROR CODE	MEANING
WSAEFAULT	One or more of the arguments is not in a valid part of the user address space.

WSAEINVAL	One or more of the arguments are invalid. This error is returned for the following conditions: the *lpProviderId* parameter is **NULL**, the *lpszProviderDllPath* parameter is invalid or the path length is too large (**MAX_PATH** was exceeded), the *lpszLspName* parameter is invalid or the name length is too large (**WSAPROTOCOL_LEN** is exceeded), the *lpProtocolInfoList* is set to a non-**NULL** and the *dwNumberOfEntries* parameter is zero, a duplicate provider ID or the layered service provider name already exist in the catalog, or a match cannot be found for the specified protocol, address family, and socket type.
WSAEINVALIDPROCTABLE	The provider is missing required functionality. A non-IFS provider must implement all of the Winsock 2 extension functions (AcceptEx, ConnectEx, DisconnectEx, TransmitFile, TransmitPackets, and LPFN_WSARECVMSG (WSARecvMsg)).
WSAEINPROGRESS	A provider installation is already in progress.
WSAENOBUFS	Memory cannot be allocated for buffers.
WSANO_RECOVERY	A nonrecoverable error occurred. This error is returned under several conditions including the following: the provider is already installed, the *lpProtocolInfoList* parameter was **NULL** and there was no base provider found, the maximum protocol chain length (**MAX_PROTOCOL_CHAIN**) was reached, the user lacks the administrative privileges required to write to the Winsock registry, or a failure occurred when creating or installing a catalog entry.
WSASYSCALLFAILURE	A system call that should never fail has failed.

Remarks

WSCInstallProviderAndChains64_32 is an enhanced version of the basic WSCInstallProvider64_32 function used to install a single transport service provider. If a layered service provider is being installed, then **WSCInstallProviderAndChains64_32** should be used. **WSCInstallProviderAndChains64_32** can install a layered protocol and one or more protocol chains with a single function call. To accomplish the same work using **WSCInstallProvider64_32** would require multiple function calls.

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 would be a security layer that adds a 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 capable of performing data communications with a remote endpoint. 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. A base protocol has the **ChainLen** member of the WSAProtocol_Info structure set to **BASE_PROTOCOL** which is defined to be 1. A layered protocol has the **ChainLen** member of the **WSAPROTOCOL_INFO** structure set to **LAYERED_PROTOCOL** which is defined to be zero. A protocol chain has the **ChainLen** member of the **WSAPROTOCOL_INFO** structure set to greater

than 1.

WSCInstallProviderAndChains64_32 is the 64-bit version of WSCInstallProviderAndChains. It installs the provider into both the 32-bit and 64-bit catalogs on 64-bit platforms. This means that on 64-bit platforms, two Winsock catalogs are maintained, and that both 32-bit and 64-bit processes are able to load the LSP installed with this function.

On a 64-bit computer, all calls not specifically designed for 32-bit (for example, all functions that do not end in "32") operate on the native 64-bit catalog. Processes that execute on a 64-bit computer must use **WSCInstallProviderAndChains64_32** to operate on both the 32-bit catalog as well as the 64-bit catalog, preserving compatibility. The definitions and semantics of the specific 32-bit calls are the same as their native counterparts.

> **Note** On 64-bit platforms, there is no **WSCInstallProviderAndChains** function to install only in the 64-bit catalog. On 64-bit platforms, there must be both a 32-bit and 64-bit version of the LSP installed. For providers that must install only in the 64-bit Winsock catalog, the WSCInstallProvider function can be used.

If *lpProtocolInfoList* is set to **NULL**, this function creates protocol chains where the provider is layered over the base protocol for each unique protocol type as defined by the address family, socket type, and protocol. This eliminates the creation of any inaccessible duplicate provider entries.

If *lpProtocolInfoList* is set to a non-**NULL** value, this function creates protocol chains by obtaining the top-most entry in the configuration information that matches the *{address family, socket type, protocol}* tuple from each element in the provided array. Again, only the *{address family, socket type, protocol}* tuple is considered; all other members and duplicates are ignored.

Upon successful completion of this call, any subsequent calls to WSAEnumProtocols, WSCEnumProtocols, or WSCEnumProtocols32 will return the newly created protocol chain entries. Be aware that in Windows environments, only instances of *Ws_32.dll* created by calling WSAStartup after the successful completion of **WSCInstallProviderAndChains64_32** will include the new entries when **WSAEnumProtocols**, **WSCEnumProtocols**, and **WSCEnumProtocols32** returns.

> **Note** The WSAEnumProtocols function does not enumerate a layered protocol entry while WSCEnumProtocols and WSCEnumProtocols32 do.

On success, **WSCInstallProviderAndChains64_32** will attempt to alert all interested applications that have registered for notification of the change by calling WSAProviderConfigChange.

The **WSCInstallProviderAndChains64_32** function can only be called by a user logged on as a member of the Administrators group. If **WSCInstallProviderAndChains64_32** is called by a user that is not a member of the Administrators group, the function call will fail and WSANO_RECOVERY is returned in the *lpErrno* parameter. For computers running Windows Vista or Windows Server 2008, this function can also fail because of user account control (UAC). If an application that contains this function 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 on Windows Vista or Windows Server 2008 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.

Any file installation or provider-specific configuration must be performed by the calling application.

IFS and Non-IFS Providers

An IFS provider is one that returns native operating system handles. Typically these handles are associated with kernel mode protocol drivers. For example, the base TCP/IPv4, UDP/IPv4, TCP/IPv6, and UDP/IPv6 are IFS providers as these entries correspond to a kernel mode component. IFS handles can be used in **ReadFile**, **WriteFile**, and **CancelIo** function calls. A layered service provider that is an IFS provider simply returns the socket handle created from the lower provider (which must also be an IFS provider) directly to the calling application. An IFS LSP cannot intercept completion notifications for Winsock calls.

A non-IFS provider is one that creates an intermediate handle with WPUCreateSocketHandle and returns this handle to the caller. This allows a non-IFS LSP to intercept send and receive completion events before the calling applications to allow for post processing (for example, decrypting a received chunk of data). Non-IFS socket handles can be used in calls to **ReadFile** and **WriteFile**, but cannot be used with **CancelIo**. The only guaranteed method of canceling an operation on a non-IFS handle is by closing the socket with closesocket.

Paths for 32-bit and 64-bit Provider DLLs

lpszProviderDllPath represents the fully qualified path to the 64-bit version of the provider DLL. This parameter can contain embedded environment strings (such as *%SystemRoot%*).

lpszProviderDllPath32 represents the fully qualified path to the 32-bit version of the provider DLL. This parameter can contain embedded environment strings (such as *%SystemRoot%*).

If *lpszProviderDllPath32* is **NULL**, then *lpszProviderDllPath* is the path for both 32 and 64 bit providers. When a 32-bit process on a 64-bit computer is running (for example, when a Winsock application loads the 32-bit version of an LSP), it attempts to load the 32-bit provider specified in *lpszProviderDllPath*. If *lpszProviderDllPath32* is **NULL**, then the *lpszProviderDllPath* parameter must be set to *%windir%\system32<dllname>*. If this is not the case, the call fails with WSAEINVAL. If the path in *lpszProviderDllPath* is *%windir%\system32<dllname>* when *lpszProviderDllPath32* is **NULL**, the call will be redirected (using the file system redirector) to the directory returned by **GetSystemWow64Directory** where the 32-bit LSP must reside. For Windows XP 64-bit edition, Windows Server 2003 and Windows Vista, this directory is *%windir%\syswow64*.

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows
Header	ws2spi.h
Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

LoadLibrary

Transport Configuration and Installation

Transport Service Providers

WPUCreateSocketHandle

WSAStartup

WSCEnumProtocols

WSCEnumProtocols32

WSCInstallProvider64_32

WSCInstallProviderAndChains

WSCWriteProviderOrder

WSCInstallQOSTemplate function (ws2spi.h)

10/22/2022 • 3 minutes to read • Edit Online

[This function is not supported in Windows Vista and subsequent versions of the operating system.]

The **WSCInstallQOSTemplate** function installs the specified QoS template in the system configuration database.

Syntax

```
int WSCInstallQOSTemplate(
  [in] const LPGUID Guid,
  [in] LPWSABUF     QosName,
  [in] LPQOS        Qos
);
```

Parameters

`[in] Guid`

The globally unique identifier (GUID) for the quality of service (QoS) provider.

`[in] QosName`

A pointer to a WSABUF structure that contains the QoS name of the template to install.

`[in] Qos`

A pointer to a QOS structure that specifies the quality of service flow specifications and any provider-specific information for the QoS template.

Return value

If **WSCInstallQOSTemplate** function succeeds, the return value is zero. Otherwise, it returns one of the following error codes.

ERROR CODE	MEANING
WSAEFAULT	One or more of the arguments is not in a valid part of the user address space.
WSAEINVAL	One or more of the arguments are invalid. This error is returned if the QoS provider specified in the *Guid* parameter is invalid or the QoS template name specified in the *QosName* parameter is invalid. This error is also returned if the contents of the template structure specified in the *Qos* parameter is invalid or incomplete.
WSAENOBUFS	Memory cannot be allocated for buffers.

WSANO_RECOVERY	A nonrecoverable error occurred. This error is returned under several conditions including the following: the provider is already installed, the user lacks the administrative privileges required to write to the Winsock registry, or a failure occurred when creating or installing a catalog entry.
WSASYSCALLFAILURE	A system call that should never fail has failed.
WSA_NOT_ENOUGH_MEMORY	Insufficient memory was available. This error is returned when there is insufficient memory to allocate a new catalog entry.

Remarks

The **WSCInstallQOSTemplate** function is not supported on Windows Vista and later. If this function is called on Windows Vista, and error is returned.

The **WSCInstallQOSTemplate** function installs a QoS template, based on a QoS name. The caller of the **WSCInstallQOSTemplate** function must have appropriate administrative rights for the call to succeed.

The QOS structure that contains the QoS settings can later be retrieved by calling the WSPGetQOSByName function and passing in the associated QoS name.

The **WSCInstallQOSTemplate** function installs a named QoS template that contains the QOS structure specified in the *Qos* parameter. If a QoS template already exists with the QoS name specified in the *Qosname* parameter, the settings specified in the *Qos* parameter replace the settings of the existing template.

If the *Guid* parameter is set to **NULL**, the installed QOS template applies to all service providers. If the *Guid* parameter is not **NULL**, then the installed QoS template applies only to the provider indicated by the *Guid* parameter.

QoS template settings are stored in nonvolatile storage, so subsequent calls to the WSAGetQOSByName function with the same QoS name specified in the *lpQOSName* parameter, return the same **QOS** structure passed to the **WSCInstallQOSTemplate** function .

Windows Sockets 2 includes a base set of QoS templates. You can override and replace any of these QoS templates or change an existing QoS template by simply installing a new template with the existing name. You do not need to delete an existing template before replacing or modifying it. You cannot delete the base set of QoS-named templates included in Windows Sockets 2. However, you can delete templates added subsequently, perhaps by other service providers.

The *Qos* parameter points to a **QOS** structure that can include a buffer that contains provider-specific settings in the **ProviderSpecific** member of the **QOS** structure. Any provider-specific settings are stored with the basic **QOS** structure structure and are returned in subsequent calls to the **WSAGetQOSByName** function.

The **ProviderSpecific** member of the **QOS** structure can be set even if the *Guid* parameter is set to **NULL** to install a global QoS template for all service providers. Note that this practice may lead a service provider to ignore the **ProviderSpecific** member of the **QOS** structure if the service provider does not recognize its contents. The recommended use of **WSCInstallQOSTemplate** function is to include provider-specific settings in the **ProviderSpecific** member of the **QOS** structure only if the named template is being installed on a particular service provider (the *Guid* parameter is not **NULL**).

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	ws2spi.h

See also

QOS

WSABUF

WSPGetQOSByName

WSCRemoveQOSTemplate function (ws2spi.h)

10/22/2022 • 2 minutes to read • Edit Online

[This function is not supported in Windows Vista and subsequent versions of the operating system.]

The **WSCRemoveQOSTemplate** function removes the specified QoS template from the system configuration database.

Syntax

```
int WSCRemoveQOSTemplate(
  [in] const LPGUID Guid,
  [in] LPWSABUF     QosName
);
```

Parameters

`[in] Guid`

The globally unique identifier (GUID) for the quality of service (QoS) provider.

`[in] QosName`

A pointer to a WSABUF structure that contains the QoS name of the template to remove.

Return value

If **WSCRemoveQOSTemplate** function succeeds, the return value is zero. Otherwise, it returns one of the following error codes.

ERROR CODE	MEANING
WSAEFAULT	One or more of the arguments is not in a valid part of the user address space.
WSAEINVAL	One or more of the arguments are invalid. This error is returned if the if QoS provider specified in the *Guid* parameter is invalid or the QoS template name specified in the *QosName* parameter is invalid.
WSAENOBUFS	Memory cannot be allocated for buffers.
WSANO_RECOVERY	A nonrecoverable error occurred. This error is returned under several conditions including the following: the provider is already installed, the user lacks the administrative privileges required to write to the Winsock registry, or a failure occurred when creating or installing a catalog entry.
WSASYSCALLFAILURE	A system call that should never fail has failed.

WSA_NOT_ENOUGH_MEMORY	Insufficient memory was available. This error is returned when there is insufficient memory to allocate a new catalog entry.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	ws2spi.h

See also

WSCInstallQOSTemplate

WSCSetApplicationCategory function (ws2spi.h)

10/22/2022 • 7 minutes to read • Edit Online

Note Layered Service Providers are deprecated. Starting with Windows 8 and Windows Server 2012, use Windows Filtering Platform.

The **WSCSetApplicationCategory** function sets the permitted layered service provider (LSP) categories associated with an application.

Syntax

```
int WSCSetApplicationCategory(
  [in]  LPCWSTR Path,
  [in]  DWORD   PathLength,
  [in]  LPCWSTR Extra,
  [in]  DWORD   ExtraLength,
  [in]  DWORD   PermittedLspCategories,
  [out] DWORD   *pPrevPermLspCat,
  [out] LPINT   lpErrno
);
```

Parameters

`[in] Path`

A pointer to a Unicode string that contains the load path to the executable image for the application. This string observes the usual rules for path resolution and can contain embedded environment strings (such as *%SystemRoot%*).

`[in] PathLength`

The length, in characters, of the *Path* parameter. This length does not include the terminating **NULL**.

`[in] Extra`

A pointer to a Unicode string which represents the command line arguments used when starting the application specified in the *Path* parameter. The *Extra* parameter is used to distinguish between multiple, distinct instances of an application when launched with a consistent command line. This is to support different application categorizations for different instances of Svchost.exe or Rundll32.exe. If only the *Path* parameter is required and no command line arguments are needed to further distinguish between instances of an application, then the *Extra* parameter should be set to **NULL**.

`[in] ExtraLength`

The length, in characters, of the *Extra* parameter. This length does not include the terminating **NULL**.

`[in] PermittedLspCategories`

A DWORD value of the LSP categories which are permitted for all instances of this application. The application is identified by the combination of the values of the *Path* and *Extra* parameters.

`[out] pPrevPermLspCat`

A pointer to receive the previous set of permitted LSP categories which were permitted for all instances of this application. This parameter is optional can be **NULL**.

`[out] lpErrno`

A pointer to the error code if the function fails.

Return value

If no error occurs, **WSCSetApplicationCategory** returns **ERROR_SUCCESS** (zero). Otherwise, it returns **SOCKET_ERROR**, and a specific error code is returned in the *lpErrno* parameter.

ERROR CODE	MEANING
WSAEFAULT	One or more of the arguments is not in a valid part of the user address space.
WSAEINVAL	One or more of the arguments are invalid.
WSANO_RECOVERY	A nonrecoverable error occurred. This error is returned under several conditions including the following: the user lacks the administrative privileges required to access the Winsock registry, or a failure occurred when opening a Winsock catalog entry or an application ID entry.

Remarks

WSCSetApplicationCategory is used to set the LSP category flags associated with an application instance. Applications can determine which LSP behaviors are acceptable within the application's context. Therefore, through specifying permitted LSP categories, an application can permit only those layered service providers which implement acceptable behaviors to be loaded.

The *Extra* parameter is required when the command line is used to distinguish between different instances of an application or service hosted within the same executable. Each instance can have different application categorization needs. Svchost.exe and Rundll32.exe are two examples where the command line is required to differentiate between different process instances. For SvcHost.exe, the **-k <svcinstance>** switch defines the process instance.

For services, using the Service Name is not sufficient, because the Winsock Catalog is global to a given process, and a process may host several services.

If the **WSCSetApplicationCategory** function is called on the same application (the same fullpath, EXE name, and parameters) multiple times, then the categories are ORed together. For example if you categorized "c:\foo.exe -param" with LSP_SYSTEM and then called the **WSCSetApplicationCategory** function again with LSP_REDIRECTOR, the resulting entry for htis application contains LSP_SYSTEM | LSP_REDIRECTOR. This behavior is designed to support a single executable file that hosts multiple applications in a single EXE (the Windows system services svchost.exe, for example).

Window sockets determine an application's identity and retrieves the permitted LSP categories during the first call to WSAStartup. This will be the set of permitted LSP categories for the duration of the application instance. Subsequent changes to the permitted LSP categories for a given application identity will not be picked up until the next instance of the application. The permitted LSP categories is not mutable during the lifetime of the application instance.

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 refers to 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.

During LSP initialization, the LSP must provide pointers to a number of Winsock SPI functions. These functions will be called during normal processing by the layer directly above the LSP (either another LSP or Ws2_32.dll).

An LSP that implements an installable file system (IFS) can selectively choose to provide pointers to functions which are implemented by itself, or pass back the pointers provided by the layer directly below the LSP. Non-IFS LSPs, because they provide their own handles, must implement all of the Winsock SPI functions. This is because each SPI will require the LSP to map all of the socket handles it created to the socket handle of the lower provider (either another LSP or the base protocol).

However, all LSPs perform their specific work by doing extra processing on only a subset of the Winsock SPI functions.

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, 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 into which an LSP can be classified.

LSP CATEGORY	DESCRIPTION
LSP_CRYPTO_COMPRESS	The LSP is a cryptography or data compression provider.
LSP_FIREWALL	The LSP is a firewall provider.
LSP_LOCAL_CACHE	The LSP is a local cache provider.
LSP_INBOUND_MODIFY	The LSP modifies inbound data.
LSP_INSPECTOR	The LSP inspects or filters data.
LSP_OUTBOUND_MODIFY	The LSP modifies outbound data.
LSP_PROXY	The LSP acts as a proxy and redirects packets.
LSP_REDIRECTOR	The LSP is a network redirector.
LSP_SYSTEM	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).

The **WSCSetApplicationCategory** function can only be called by a user logged on as a member of the Administrators group. If **WSCSetApplicationCategory** is called by a user that is not a member of the Administrators group, the function call will fail and **WSANO_RECOVERY** is returned in the *lpErrno* parameter. This function can also fail because of user account control (UAC). If an application that contains this function 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 on Windows Vista or Windows Server 2008 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.

Any file installation or service provider-specific configuration must be performed by the caller.

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows
Header	ws2spi.h
Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

Categorizing Layered Service Providers and Applications

WSAStartup

WSCGetApplicationCategory

WSCGetProviderInfo

WSCGetProviderInfo32

WSCSetProviderInfo

WSCSetProviderInfo32

WSCSetProviderInfo function (ws2spi.h)

10/22/2022 • 6 minutes to read • Edit Online

Note Layered Service Providers are deprecated. Starting with Windows 8 and Windows Server 2012, use Windows Filtering Platform.

The **WSCSetProviderInfo** function sets the data value for the specified information class for a layered service provider (LSP).

Syntax

```
int WSCSetProviderInfo(
  [in]  LPGUID                 lpProviderId,
  [in]  WSC_PROVIDER_INFO_TYPE InfoType,
  [in]  PBYTE                  Info,
  [in]  size_t                 InfoSize,
  [in]  DWORD                  Flags,
  [out] LPINT                  lpErrno
);
```

Parameters

`[in] lpProviderId`

A pointer to a globally unique identifier (GUID) for the provider.

`[in] InfoType`

The information class to be set for this LSP protocol entry.

`[in] Info`

A pointer to a buffer that contains the information class data to set for the LSP protocol entry.

`[in] InfoSize`

The size, in bytes, of the buffer pointed to by the *Info* parameter.

`[in] Flags`

The flags used to modify the behavior of the **WSCSetProviderInfo** function call.

`[out] lpErrno`

A pointer to the error code if the function fails.

Return value

If no error occurs, **WSCSetProviderInfo** returns **ERROR_SUCCESS** (zero). Otherwise, it returns **SOCKET_ERROR**, and a specific error code is returned in the *lpErrno* parameter.

ERROR CODE	MEANING

ERROR_CALL_NOT_IMPLEMENTED	The call is not implemented. This error is returned if **ProviderInfoAudit** is specified in the *InfoType* parameter.
WSAEFAULT	One or more of the arguments is not in a valid part of the user address space.
WSAEINVAL	One or more of the arguments are invalid.
WSANO_RECOVERY	A nonrecoverable error occurred. This error is returned under several conditions including the following: the user lacks the administrative privileges required to write to the Winsock registry, or a failure occurred when opening a Winsock catalog entry.
WSA_NOT_ENOUGH_MEMORY	Insufficient memory was available. This error is returned when there is insufficient memory to allocate a new catalog entry.

Remarks

WSCSetProviderInfo is used to set the information class data for a layered service provider. When the *InfoType* parameter is set to **ProviderInfoLspCategories**, on success **WSCSetProviderInfo** sets appropriate LSP category flags implemented by the provider based on the value passed in the *Info* parameter.

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 refers to 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. A protocol chain would then be defined as one or more layered protocols strung together and anchored by a base protocol. A base protocol has the **ChainLen** member of the WSAPROTOCOL_INFO structure set to **BASE_PROTOCOL** which is defined to be 1. A layered protocol has the **ChainLen** member of the **WSAPROTOCOL_INFO** structure set to **LAYERED_PROTOCOL** which is defined to be zero. A protocol chain has the **ChainLen** member of the **WSAPROTOCOL_INFO** structure set to greater than 1.

During LSP initialization, the LSP must provide pointers to a number of Winsock SPI functions. These functions will be called during normal processing by the layer directly above the LSP (either another LSP or Ws2_32.dll).

An LSP that implements an installable file system (IFS) can selectively choose to provide pointers to functions which are implemented by itself, or pass back the pointers provided by the layer directly below the LSP. Non-IFS LSPs, because they provide their own handles, must implement all of the Winsock SPI functions. This is because each SPI will require the LSP to map all of the socket handles it created to the socket handle of the lower provider (either another LSP or the base protocol).

However, all LSPs perform their specific work by doing extra processing on only a subset of the Winsock SPI functions.

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, 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 into which an LSP can be classified.

LSP CATEGORY	DESCRIPTION
LSP_CRYPTO_COMPRESS	The LSP is a cryptography or data compression provider.
LSP_FIREWALL	The LSP is a firewall provider.
LSP_LOCAL_CACHE	The LSP is a local cache provider.
LSP_INBOUND_MODIFY	The LSP modifies inbound data.
LSP_INSPECTOR	The LSP inspects or filters data.
LSP_OUTBOUND_MODIFY	The LSP modifies outbound data.
LSP_PROXY	The LSP acts as a proxy and redirects packets.
LSP_REDIRECTOR	The LSP is a network redirector.
LSP_SYSTEM	The LSP is acceptable for use in services and system processes.

An LSP may belong to more than one category. For example, firewall/security LSP could belong to both the inspector (**LSP_INSPECTOR**) and firewall (**LSP_FIREWALL**) categories.

If an LSP does not have 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).

The **WSCSetProviderInfo** function can only be called by a user logged on as a member of the Administrators group. If **WSCSetProviderInfo** is called by a user that is not a member of the Administrators group, the function call will fail and **WSANO_RECOVERY** is returned in the *lpErrno* parameter. This function can also fail because of user account control (UAC). If an application that contains this function 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 on Windows Vista or Windows Server 2008 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.

> **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).

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows
Header	ws2spi.h
Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

Categorizing Layered Service Providers and Applications WSAPROTOCOL_INFO WSCGetApplicationCategory WSCGetProviderInfo WSCSetApplicationCategory WSC_PROVIDER_INFO_TYPE

WSCSetProviderInfo32 function (ws2spi.h)

10/22/2022 • 6 minutes to read • Edit Online

Note Layered Service Providers are deprecated. Starting with Windows 8 and Windows Server 2012, use Windows Filtering Platform.

The **WSCSetProviderInfo32** function sets the data value for specified information class for a layered service provider (LSP).

Note This call is a strictly 32-bit version of WSCSetProviderInfo for use on 64-bit platforms. It is provided to allow 64-bit processes to access the 32-bit catalogs.

Syntax

```
int WSCSetProviderInfo32(
  [in]  LPGUID                 lpProviderId,
  [in]  WSC_PROVIDER_INFO_TYPE InfoType,
  [in]  PBYTE                  Info,
  [in]  size_t                 InfoSize,
  [in]  DWORD                  Flags,
  [out] LPINT                  lpErrno
);
```

Parameters

`[in] lpProviderId`

A pointer to a globally unique identifier (GUID) for the provider.

`[in] InfoType`

The information class to be set for this LSP protocol entry.

`[in] Info`

A pointer to a buffer that contains the information class data to set for the LSP protocol entry.

`[in] InfoSize`

The size, in bytes, of the buffer pointed to by the *Info* parameter.

`[in] Flags`

The flags used to modify the behavior of the **WSCSetProviderInfo32** function call.

`[out] lpErrno`

A pointer to the error code if the function fails.

Return value

If no error occurs, **WSCSetProviderInfo32** returns **ERROR_SUCCESS** (zero). Otherwise, it returns **SOCKET_ERROR**, and a specific error code is returned in the *lpErrno* parameter.

ERROR CODE	MEANING
ERROR_CALL_NOT_IMPLEMENTED	The call is not implemented. This error is returned if **ProviderInfoAudit** is specified in the *InfoType* parameter.
WSAEFAULT	One or more of the arguments is not in a valid part of the user address space.
WSAEINVAL	One or more of the arguments are invalid.
WSANO_RECOVERY	A nonrecoverable error occurred. This error is returned under several conditions including the following: the user lacks the administrative privileges required to write to the Winsock registry, or a failure occurred when opening a Winsock catalog entry.
WSA_NOT_ENOUGH_MEMORY	Insufficient memory was available. This error is returned when there is insufficient memory to allocate a new catalog entry.

Remarks

WSCSetProviderInfo32 is a strictly 32-bit version of WSCSetProviderInfo. On a 64-bit computer, all calls not specifically 32-bit (for example, all functions that do not end in "32") operate on the native 64-bit catalog. Processes that execute on a 64-bit computer must use the specific 32-bit function calls to operate on a strictly 32-bit catalog and preserve compatibility. The definitions and semantics of the specific 32-bit calls are the same as their native counterparts.

WSCSetProviderInfo32 is used to set the information class data for a 32-bit layered service provider. When the *InfoType* parameter is set to **ProviderInfoLspCategories**, on success **WSCSetProviderInfo32** sets appropriate LSP category flags implemented by the provider based on the value passed in the *Info* parameter.

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 refers to 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. A protocol chain would then be defined as one or more layered protocols strung together and anchored by a base protocol. A base protocol has the **ChainLen** member of the WSAProtocol_Info structure set to **BASE_PROTOCOL** which is defined to be 1. A layered protocol has the **ChainLen** member of the **WSAPROTOCOL_INFO** structure set to **LAYERED_PROTOCOL** which is defined to be zero. A protocol chain has the **ChainLen** member of the **WSAPROTOCOL_INFO** structure set to greater than 1.

During LSP initialization, the LSP must provide pointers to a number of Winsock SPI functions. These functions will be called during normal processing by the layer directly above the LSP (either another LSP or Ws2_32.dll).

An LSP that implements an installable file system (IFS) can selectively choose to provide pointers to functions which are implemented by itself, or pass back the pointers provided by the layer directly below the LSP. Non-IFS LSPs, because they provide their own handles, must implement all of the Winsock SPI functions. This is because each SPI will require the LSP to map all of the socket handles it created to the socket handle of the lower provider (either another LSP or the base protocol).

However, all LSPs perform their specific work by doing extra processing on only a subset of the Winsock SPI functions.

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, 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 into which an LSP can be classified.

LSP CATEGORY	DESCRIPTION
LSP_CRYPTO_COMPRESS	The LSP is a cryptography or data compression provider.
LSP_FIREWALL	The LSP is a firewall provider.
LSP_LOCAL_CACHE	The LSP is a local cache provider.
LSP_INBOUND_MODIFY	The LSP modifies inbound data.
LSP_INSPECTOR	The LSP inspects or filters data.
LSP_OUTBOUND_MODIFY	The LSP modifies outbound data.
LSP_PROXY	The LSP acts as a proxy and redirects packets.
LSP_REDIRECTOR	The LSP is a network redirector.
LSP_SYSTEM	The LSP is acceptable for use in services and system processes.

An LSP may belong to more than one category. For example, firewall/security LSP could belong to both the inspector (**LSP_INSPECTOR**) and firewall (**LSP_FIREWALL**) categories.

If an LSP does not have 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).

The **WSCSetProviderInfo32** function can only be called by a user logged on as a member of the Administrators group. If **WSCSetProviderInfo32** is called by a user that is not a member of the Administrators group, the function call will fail and **WSANO_RECOVERY** is returned in the *lpErrno* parameter. This function can also fail because of user account control (UAC). If an application that contains this function 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 on Windows Vista or Windows Server 2008 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.

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows
Header	ws2spi.h
Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

Categorizing Layered Service Providers and Applications

WSAProtocol_Info

WSCGetApplicationCategory

WSCGetProviderInfo32

WSCSetApplicationCategory

WSCSetProviderInfo

WSC_PROVIDER_INFO_TYPE

WSCUnInstallNameSpace function (ws2spi.h)

10/22/2022 • 2 minutes to read • Edit Online

The **WSCUnInstallNameSpace** function uninstalls the indicated name-space provider.

Syntax

```
INT WSCUnInstallNameSpace(
  [in] LPGUID lpProviderId
);
```

Parameters

`[in] lpProviderId`

A pointer to a globally unique identifier (GUID) for the name-space provider to be uninstalled.

Return value

If no error occurs, **WSCUnInstallNameSpace** returns **NO_ERROR** (zero). Otherwise, it returns **SOCKET_ERROR** if the function fails, and you must retrieve the appropriate error code using the WSAGetLastError function.

ERROR CODE	MEANING
WSAEFAULT	The *lpProviderId* parameter points to memory that is not in a valid part of the user address space.
WSAEINVAL	The specified namespace–provider identifier is invalid.
WSASYSCALLFAILURE	A system call that should never fail has failed.
WSA_NOT_ENOUGH_MEMORY	Insufficient memory was available. This error is returned when there is insufficient memory to allocate a new catalog entry.

Remarks

The namespace configuration functions do not affect applications that are already running. Newly installed name-space providers will not be visible to applications nor will the changes in a name-space provider's activation state. Applications launched after the call to **WSCUnInstallNameSpace** will see the changes.

On success, **WSCUnInstallNameSpace** will attempt to alert all interested applications that have registered for notification of the change by calling WSAProviderConfigChange.

The **WSCUnInstallNameSpace** function can only be called by a user logged on as a member of the Administrators group. If **WSCUnInstallNameSpace** is called by a user that is not a member of the Administrators group, the function call will fail and **WSANO_RECOVERY** is returned in the *lpErrno* parameter.

For computers running on Windows Vista or Windows Server 2008, this function can also fail because of user account control (UAC). If an application that contains this function 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 on Windows Vista or Windows Server 2008 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.

The caller of this function must remove any additional files or service provider–specific configuration information that is required to completely uninstall the service provider.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	ws2spi.h
Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

WSAProviderConfigChange

WSCDeinstallProvider

WSCEnumProtocols

WSCInstallNameSpace

WSCUnInstallNameSpace32

WSCUnInstallNameSpace32 function (ws2spi.h)

10/22/2022 • 2 minutes to read • Edit Online

The **WSCUnInstallNameSpace32** function uninstalls a specific 32-bit namespace provider.

> **Note** This call is a strictly 32-bit version of WSCUnInstallNameSpace for use on 64-bit platforms. It is provided to allow 64-bit processes to access the 32-bit catalogs.

Syntax

```
INT WSCUnInstallNameSpace32(
  [in] LPGUID lpProviderId
);
```

Parameters

`[in] lpProviderId`

A pointer to a globally unique identifier (GUID) for the name-space provider to be uninstalled.

Return value

If no error occurs, **WSCUnInstallNameSpace32** returns **NO_ERROR** (zero). Otherwise, it returns **SOCKET_ERROR** if the function fails, and you must retrieve the appropriate error code using the WSAGetLastError function.

ERROR CODE	MEANING
WSAEFAULT	The *lpProviderId* parameter points to memory that is not in a valid part of the user address space.
WSAEINVAL	The specified namespace–provider identifier is invalid.
WSASYSCALLFAILURE	A system call that should never fail has failed.
WSA_NOT_ENOUGH_MEMORY	Insufficient memory was available. This error is returned when there is insufficient memory to allocate a new catalog entry.

Remarks

WSCUnInstallNameSpace32 is a strictly 32-bit version of WSCUnInstallNameSpace. On a 64-bit computer, all calls not specifically 32-bit (for example, all functions that do not end in "32") operate on the native 64-bit catalog. Processes that execute on a 64-bit computer must use the specific 32-bit function calls to operate on a strictly 32-bit catalog and preserve compatibility. The definitions and semantics of the specific 32-bit calls are

the same as their native counterparts.

The namespace configuration functions do not affect applications that are already running. Newly installed name-space providers will not be visible to applications nor will the changes in a name-space provider's activation state. Applications launched after the call to **WSCUnInstallNameSpace32** will recognize the changes.

On success, **WSCUnInstallNameSpace32** will attempt to alert all interested applications that have registered for notification of the change by calling WSAProviderConfigChange.

The **WSCUnInstallNameSpace32** function can only be called by a user logged on as a member of the Administrators group. If **WSCUnInstallNameSpace32** is called by a user that is not a member of the Administrators group, the function call will fail and **WSANO_RECOVERY** is returned in the *lpErrno* parameter.

For computers running on Windows Vista or Windows Server 2008, this function can also fail because of user account control (UAC). If an application that contains this function 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 on Windows Vista or Windows Server 2008 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.

The caller of this function must remove any additional files or service provider–specific configuration information that is required to completely uninstall the service provider.

Requirements

Minimum supported client	Windows Vista, Windows XP Professional x64 Edition [desktop apps only]
Minimum supported server	Windows Server 2008, Windows Server 2003 x64 Edition [desktop apps only]
Target Platform	Windows
Header	ws2spi.h
Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

WSAProviderConfigChange

WSCDeinstallProvider32

WSCEnumProtocols32

WSCInstallNameSpace32

WSCUnInstallNameSpace

WSCUpdateProvider function (ws2spi.h)

10/22/2022 • 3 minutes to read • Edit Online

The **WSCUpdateProvider** function modifies the specified transport provider in the system configuration database.

Syntax

```
int WSCUpdateProvider(
  [in]  LPGUID                    lpProviderId,
  [in]  const WCHAR               *lpszProviderDllPath,
  [in]  const LPWSAPROTOCOL_INFOW lpProtocolInfoList,
  [in]  DWORD                     dwNumberOfEntries,
  [out] LPINT                     lpErrno
);
```

Parameters

`[in] lpProviderId`

A pointer to a globally unique identifier (GUID) for the provider.

`[in] lpszProviderDllPath`

A pointer to a Unicode string that contains the load path to the provider 64-bit DLL. This string observes the usual rules for path resolution and can contain embedded environment strings (such as *%SystemRoot%*). Such environment strings are expanded when the Ws2_32.dll must subsequently load the provider DLL on behalf of an application. After any embedded environment strings are expanded, the Ws2_32.dll passes the resulting string to the LoadLibrary function which loads the provider into memory. For more information, see **LoadLibrary**.

`[in] lpProtocolInfoList`

A pointer to an array of WSAProtocol_Info structures. Each structure specifies or modifies a protocol, address family, and socket type supported by the provider.

`[in] dwNumberOfEntries`

The number of entries in the *lpProtocolInfoList* array.

`[out] lpErrno`

A pointer to the error code if the function fails.

Return value

If no error occurs, **WSCUpdateProvider** returns zero. Otherwise, it returns **SOCKET_ERROR**, and a specific error code is returned in the *lpErrno* parameter.

ERROR CODE	MEANING
WSAEFAULT	One or more of the arguments are not in a valid part of the user address space.

WSAEINVAL	One or more of the arguments are invalid.
WSANO_RECOVERY	A nonrecoverable error occurred. This error is returned under several conditions including the following: the user lacks the administrative privileges required to write to the Winsock registry, or a failure occurred when opening or writing a catalog entry.
WSA_NOT_ENOUGH_MEMORY	Insufficient memory was available. This error is returned when there is insufficient memory to allocate a new catalog entry.

Remarks

The **WSCUpdateProvider** function modifies Windows Sockets 2 configuration information for the specified provider. It is applicable to base protocols, layered protocols, and protocol chains.

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 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 capable of performing data communications with a remote endpoint. 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. A base protocol has the **ChainLen** member of the WSAProtocol_Info structure set to **BASE_PROTOCOL** which is defined to be 1. A layered protocol has the **ChainLen** member of the **WSAPROTOCOL_INFO** structure set to **LAYERED_PROTOCOL** which is defined to be zero. A protocol chain has the **ChainLen** member of the **WSAPROTOCOL_INFO** structure set to greater than 1.

On success, **WSCUpdateProvider** will attempt to alert all interested applications that have registered for notification of the change by calling WSAProviderConfigChange.

The **WSCUpdateProvider** function can only be called by a user logged on as a member of the Administrators group. If **WSCUpdateProvider** is called by a user that is not a member of the Administrators group, the function call will fail.

For computers running on Windows Vista or Windows Server 2008, this function can also fail because of user account control (UAC). If an application that contains this function 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 on Windows Vista or Windows Server 2008 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.

Any file installation or service provider-specific configuration must be performed by the caller.

Requirements

Minimum supported client	Windows XP [desktop apps only]

Minimum supported server	Windows Server 2003 [desktop apps only]
Target Platform	Windows
Header	ws2spi.h
Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

WSAProtocol_Info

WSAProviderConfigChange

WSAStartup

WSCDeinstallProvider

WSCEnumProtocols

WSCInstallProvider

WSCUpdateProvider32 function (ws2spi.h)

10/22/2022 • 4 minutes to read • Edit Online

The **WSCUpdateProvider32** function modifies the specified 32-bit transport provider in the system configuration database.

> **Note** This call is a strictly 32-bit version of WSCUpdateProvider for use on 64-bit platforms. It is provided to allow 64-bit processes to access the 32-bit catalogs.

Syntax

```
int WSCUpdateProvider32(
  [in]  LPGUID                    lpProviderId,
  [in]  const WCHAR               *lpszProviderDllPath,
  [in]  const LPWSAPROTOCOL_INFOW lpProtocolInfoList,
  [in]  DWORD                     dwNumberOfEntries,
  [out] LPINT                     lpErrno
);
```

Parameters

`[in] lpProviderId`

A pointer to a globally unique identifier (GUID) for the provider.

`[in] lpszProviderDllPath`

A pointer to a Unicode string that contains the load path to the provider 64-bit DLL. This string observes the usual rules for path resolution and can contain embedded environment strings (such as *%SystemRoot%*). Such environment strings are expanded when the Ws2_32.dll must subsequently load the provider DLL on behalf of an application. After any embedded environment strings are expanded, the Ws2_32.dll passes the resulting string to the LoadLibrary function which loads the provider into memory. For more information, see **LoadLibrary**.

`[in] lpProtocolInfoList`

A pointer to an array of WSAProtocol_Info structures. Each structure specifies or modifies a protocol, address family, and socket type supported by the provider.

`[in] dwNumberOfEntries`

The number of entries in the *lpProtocolInfoList* array.

`[out] lpErrno`

A pointer to the error code if the function fails.

Return value

If no error occurs, **WSCUpdateProvider32** returns zero. Otherwise, it returns SOCKET_ERROR, and a specific

error code is available in *lpErrno*.

ERROR CODE	MEANING
WSAEFAULT	One or more of the arguments is not in a valid part of the user address space.
WSAEINVAL	One or more of the arguments are invalid.
WSANO_RECOVERY	A nonrecoverable error occurred. This error is returned under several conditions including the following: the user lacks the administrative privileges required to write to the Winsock registry, or a failure occurred when opening or writing a catalog entry.
WSA_NOT_ENOUGH_MEMORY	Insufficient memory was available. This error is returned when there is insufficient memory to allocate a new catalog entry.

Remarks

WSCUpdateProvider32 is a strictly 32-bit version of WSCUpdateProvider. On a 64-bit computer, all calls not specifically 32-bit (for example, all functions that do not end in "32") operate on the native 64-bit catalog. Processes that execute on a 64-bit computer must use the specific 32-bit function calls to operate on a strictly 32-bit catalog and preserve compatibility. The definitions and semantics of the specific 32-bit calls are the same as their native counterparts.

This function modifies Windows Sockets 2 configuration information for the specified 32-bit provider. It is applicable to base protocols, layered protocols, and protocol chains.

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 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 capable of performing data communications with a remote endpoint. 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. A base protocol has the **ChainLen** member of the WSAProtocol_Info structure set to **BASE_PROTOCOL** which is defined to be 1. A layered protocol has the **ChainLen** member of the **WSAPROTOCOL_INFO** structure set to **LAYERED_PROTOCOL** which is defined to be zero. A protocol chain has the **ChainLen** member of the **WSAPROTOCOL_INFO** structure set to greater than 1.

On success, **WSCUpdateProvider32** will attempt to alert all interested applications that have registered for notification of the change by calling WSAProviderConfigChange.

The **WSCUpdateProvider32** function can only be called by a user logged on as a member of the Administrators group. If **WSCUpdateProvider32** is called by a user that is not a member of the Administrators group, the function call will fail.

For computers running Windows Vista or Windows Server 2008, this function can also fail because of user account control (UAC). If an application that contains this function 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 on Windows Vista or Windows Server 2008 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.

Any file installation or service provider-specific configuration must be performed by the caller.

Requirements

Minimum supported client	Windows Vista, Windows XP Professional x64 Edition [desktop apps only]
Minimum supported server	Windows Server 2008, Windows Server 2003 x64 Edition [desktop apps only]
Target Platform	Windows
Header	ws2spi.h
Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

WSAProtocol_Info

WSAProviderConfigChange

WSAStartup

WSCDeinstallProvider32

WSCEnumProtocols32

WSCInstallProvider64_32

WSCInstallProviderAndChains64_32

WSCUpdateProvider

WSPDATA structure (ws2spi.h)

10/22/2022 • 2 minutes to read • Edit Online

The **WSPDATA** structure contains service provider information.

Syntax

```
typedef struct WSPData {
  WORD  wVersion;
  WORD  wHighVersion;
  WCHAR szDescription[WSPDESCRIPTION_LEN + 1];
} WSPDATA, *LPWSPDATA;
```

Members

`wVersion`

Version of the Windows Sockets SPI specification that the Windows Sockets service provider expects the caller to use.

`wHighVersion`

Highest version of the Windows Sockets SPI specification that this service provider can support (also encoded as above). Normally this will be the same as **wVersion**.

`szDescription[WSPDESCRIPTION_LEN + 1]`

Null-terminated Unicode string into which the Windows Sockets provider copies a description of itself. The text (up to 256 characters in length) can contain any characters except control and formatting characters: the most likely use to which an SPI client will put this is to display it (possibly truncated) in a status message.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	ws2spi.h

See also

WSPStartup

WSPPROC_TABLE structure (ws2spi.h)

10/22/2022 • 2 minutes to read • Edit Online

The **WSPPROC_TABLE** structure contains a table of pointers to service provider functions.

Syntax

```
typedef struct _WSPPROC_TABLE {
  LPWSPACCEPT               lpWSPAccept;
  LPWSPADDRESSTOSTRING      lpWSPAddressToString;
  LPWSPASYNCSELECT          lpWSPAsyncSelect;
  LPWSPBIND                 lpWSPBind;
  LPWSPCANCELBLOCKINGCALL   lpWSPCancelBlockingCall;
  LPWSPCLEANUP              lpWSPCleanup;
  LPWSPCLOSESOCKET          lpWSPCloseSocket;
  LPWSPCONNECT              lpWSPConnect;
  LPWSPDUPLICATESOCKET      lpWSPDuplicateSocket;
  LPWSPENUMNETWORKEVENTS    lpWSPEnumNetworkEvents;
  LPWSPEVENTSELECT          lpWSPEventSelect;
  LPWSPGETOVERLAPPEDRESULT lpWSPGetOverlappedResult;
  LPWSPGETPEERNAME          lpWSPGetPeerName;
  LPWSPGETSOCKNAME          lpWSPGetSockName;
  LPWSPGETSOCKOPT           lpWSPGetSockOpt;
  LPWSPGETQOSBYNAME         lpWSPGetQOSByName;
  LPWSPIOCTL                lpWSPIoctl;
  LPWSPJOINLEAF             lpWSPJoinLeaf;
  LPWSPLISTEN               lpWSPListen;
  LPWSPRECV                 lpWSPRecv;
  LPWSPRECVDISCONNECT       lpWSPRecvDisconnect;
  LPWSPRECVFROM             lpWSPRecvFrom;
  LPWSPSELECT               lpWSPSelect;
  LPWSPSEND                 lpWSPSend;
  LPWSPSENDDISCONNECT       lpWSPSendDisconnect;
  LPWSPSENDTO               lpWSPSendTo;
  LPWSPSETSOCKOPT           lpWSPSetSockOpt;
  LPWSPSHUTDOWN             lpWSPShutdown;
  LPWSPSOCKET               lpWSPSocket;
  LPWSPSTRINGTOADDRESS      lpWSPStringToAddress;
} WSPPROC_TABLE, *LPWSPPROC_TABLE;
```

Members

`lpWSPAccept`

A pointer to the **LPWSPAccept** function.

`lpWSPAddressToString`

A pointer to the **LPWSPAddressToString** function.

`lpWSPAsyncSelect`

A pointer to the **LPWSPAsyncSelect** function.

`lpWSPBind`

A pointer to the **LPWSPBind** function.

`lpWSPCancelBlockingCall`

A pointer to the **LPWSPCancelBlockingCall** function.

`lpWSPCleanup`

A pointer to the **WSPCleanup** function.

`lpWSPCloseSocket`

A pointer to the **LPWSPCloseSocket** function.

`lpWSPConnect`

A pointer to the **LPWSPConnect** function.

`lpWSPDuplicateSocket`

A pointer to the **WSPDuplicateSocket** function.

`lpWSPEnumNetworkEvents`

A pointer to the **WSPEnumNetworkEvents** function.

`lpWSPEventSelect`

A pointer to the **LPWSPEventSelect** function.

`lpWSPGetOverlappedResult`

A pointer to the **LPWSPGetOverlappedResult** function.

`lpWSPGetPeerName`

A pointer to the function.

`lpWSPGetSockName`

A pointer to the **WSPGetSockName** function.

`lpWSPGetSockOpt`

A pointer to the **LPWSPGetSockopt** function.

`lpWSPGetQOSByName`

A pointer to the **WSPGetQOSByName** function.

`lpWSPIoctl`

A pointer to the **LPWSPIoctl** function.

`lpWSPJoinLeaf`

A pointer to the **LPWSPJoinLeaf** function.

`lpWSPListen`

A pointer to the **LPWSPListen** function.

`lpWSPRecv`

A pointer to the **LPWSPRecv** function.

`lpWSPRecvDisconnect`

A pointer to the **WSPRecvDisconnect** function.

`lpWSPRecvFrom`

A pointer to the **LPWSPRecvFrom** function.

`lpWSPSelect`

A pointer to the **LPWSPSelect** function.

`lpWSPSend`

A pointer to the **LPWSPSend** function.

`lpWSPSendDisconnect`

A pointer to the **WSPSendDisconnect** function.

`lpWSPSendTo`

A pointer to the **LPWSPSendTo** function.

`lpWSPSetSockOpt`

A pointer to the **LPWSPSetSockOpt** function.

`lpWSPShutdown`

A pointer to the **LPWSPShutdown** function.

`lpWSPSocket`

A pointer to the LPWSPSocket function.

`lpWSPStringToAddress`

A pointer to the LPWSPStringToAddress function.

Remarks

The **WSPPROC_TABLE** structure contains a table of pointers to service provider functions that are returned by the WSPStartup function.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	ws2spi.h

See also

LPWSPAccept

LPWSPStringToAddress**

LPWSPAsyncSelect

LPWSPBind

LPWSPCancelBlockingCall

LPWSPCleanup

LPWSPCloseSocket

LPWSPConnect

LPWSPDuplicateSocket

LPWSPEnumNetworkEvents

LPWSPGetOverlappedResult

LPWSPGetQOSByName

LPWSPGetSockName

LPWSPGetSockopt

LPWSPIoctl

LPWSPJoinLeaf

LPWSPListen

LPWSPRecv

LPWSPRecvDisconnect

LPWSPRecvFrom

LPWSPSend

LPWSPSendDisconnect

LPWSPSendTo

LPWSPSetSockOpt

LPWSPShutdown

LPWSPSocket

WSPStartup

WSAStringToAddress

WSPStartup function (ws2spi.h)

10/22/2022 • 11 minutes to read • Edit Online

The **WSPStartup** function initiates use of a Windows Sockets service provider interface (SPI) by a client.

Syntax

```
int WSPStartup(
  [in]  WORD                wVersionRequested,
  [out] LPWSPDATA           lpWSPData,
  [in]  LPWSAPROTOCOL_INFOW lpProtocolInfo,
  [in]  WSPUPCALLTABLE      UpcallTable,
  [out] LPWSPPROC_TABLE     lpProcTable
);
```

Parameters

`[in] wVersionRequested`

The highest version of Windows Sockets SPI support that the caller can use. The high-order byte specifies the minor version (revision) number; the low-order byte specifies the major version number.

`[out] lpWSPData`

A pointer to a WSPDATA data structure that receives information about the Windows Sockets service provider.

`[in] lpProtocolInfo`

A pointer to a WSAProtocol_Info structure that defines the characteristics of the desired protocol. This is especially useful when a single provider DLL is capable of instantiating multiple different service providers.

`[in] UpcallTable`

The Winsock 2 DLL (Ws2_32.dll) upcall dispatch table passed in a WSPUpCallTable structure.

`[out] lpProcTable`

A pointer to the table of SPI function pointers. This table is returned as an WSPProc_Table structure.

Return value

The **WSPStartup** function returns zero if successful. Otherwise, it returns one of the error codes listed below.

ERROR CODE	MEANING
WSASYSNOTREADY	Network subsystem is unavailable. This error is returned if the Windows Sockets implementation cannot function at this time because the underlying system it uses to provide network services is currently unavailable.
WSAVERNOTSUPPORTED	The Winsock.dll version is out of range. This error is returned if the version of Windows Sockets SPI support requested is not provided by this particular Windows Sockets service provider.

WSAEINPROGRESS	A blocking Windows Sockets 1.1 operation is in progress.
WSAEPROCLIM	A limit on the number of tasks supported by the Windows Sockets implementation has been reached.
WSAEFAULT	The *lpWSPData* or *lpProcTable* parameter is invalid.

Remarks

Windows Sockets 2 transport service providers are DLLs with a single exported procedure entry point, **WSPStartup**, used for the service provider initialization function. All other service provider functions are made accessible to the Winsock 2 DLL via the service provider's dispatch table passed in the *lpProcTable* parameter to the **WSPStartup** function. Service provider DLL's are loaded into memory by the WinSock 2 DLL only when needed, and are unloaded when their services are no longer required.

The service provider interface also defines several circumstances in which a transport service provider calls up into the Winsock 2 DLL (upcalls) to obtain DLL support services. The transport service provider is returned the upcall dispatch table for the Winsock 2 DLL in the *UpcallTable* parameter passed to the **WSPStartup** function.

The **WSPStartup** function must be the first Windows Sockets SPI function called by a Windows Sockets SPI client on a per-process basis. It allows the client to specify the version of Windows Sockets SPI required and to provide its upcall dispatch table. All upcalls (that is, functions prefixed with WPU) made by the Windows Sockets service provider are invoked through the client's upcall dispatch table. This function also allows the client to retrieve details of the specific Windows Sockets service provider implementation. The Windows Sockets SPI client can only issue further Windows Sockets SPI functions after a successful **WSPStartup** invocation. A table of pointers to the rest of the SPI functions is retrieved through the *lpProcTable* parameter that returns a WSPProc_Table structure.

The Winsock 2 DLL loads the service provider's interface DLL into the system by using the standard Windows dynamic library loading mechanisms, and initializes it by calling the **WSPStartup** function. This is usually triggered by an application calling the socket or WSASocket function in order to create a new socket that is to be associated with a service provider whose interface DLL is not currently loaded into memory.

In order to support future versions of the Windows Sockets SPI and the Ws2_32.dll, which may have functional differences from the current Windows Sockets SPI, a negotiation takes place in **WSPStartup**. The caller of **WSPStartup** (either the Ws2_32.dll or a layered protocol) and the Windows Sockets service provider indicate to each other the highest version of Windows Sockets that they can support, and each confirms that the other's highest version is acceptable. Upon entry to **WSPStartup**, the Windows Sockets service provider examines the version requested by the client. If this version is equal to or higher than the lowest version supported by the service provider, the call succeeds and the service provider returns in the **wHighVersion** member of the WSPDATA structure the highest version it supports and in the **wVersion** member the minimum of its high version and version specified in the *wVersionRequested* parameter. The Windows Sockets service provider then assumes that the Windows Sockets SPI client will use the version of Windows Sockets specified in the **wVersion** member. If the **wVersion** member of the WSPDATA structure is unacceptable to the caller, it should call LPWSPCleanup and either search for another Windows Sockets service provider or fail to initialize.

This negotiation allows both a Windows Sockets service provider and a Windows Sockets SPI client to support a range of Windows Sockets versions. A client can successfully utilize a Windows Sockets service provider if there is any overlap in the version ranges.

The current version of the Windows Sockets specification is version 2.2. The current Winsock DLL, *Ws2_32.dll*, supports applications that request any of the following versions of Windows Sockets specification:

- 1.0
- 1.1
- 2.0
- 2.1
- 2.2

To get full access to the new syntax of a higher version of the Windows Sockets specification, the application must negotiate for this higher version. In this case, the *wVersionRequested* parameter should be set to request version 2.2. The application must also fully conform to that higher version of the Windows Socket specification, such as compiling against the appropriate header file, linking with a new library, or other special cases. The *Winsock2.h header* file for Winsock 2 support is included with the Microsoft Windows Software Development Kit (SDK).

Windows Sockets version 2.2 is supported on Windows Server 2008, Windows Vista, Windows Server 2003, Windows XP, Windows 2000, Windows NT 4.0 with Service Pack 4 (SP4) and later, Windows Me, Windows 98, and Windows 95 OSR2. Windows Sockets version 2.2 is also supported on
Windows 95 with the Windows Socket 2 Update. Applications on these platforms should normally request Winsock 2.2 by setting the *wVersionRequested* parameter accordingly.

On Windows 95 and versions of Windows NT 3.51 and earlier, Windows Sockets version 1.1 is the highest version of the Windows Sockets specification supported.

It is legal and possible for an application or DLL written to use a lower version of the Windows Sockets specification that is supported by the Winsock DLL to successfully negotiate this lower version using the **WSPStartup** function. For example, an application can request version 1.1 in the *wVersionRequested* parameter passed to the **WSPStartup** function on a platform with the Winsock 2.2 DLL. In this case, the application should only rely on features that fit within the version requested. New Ioctl codes, new behavior of existing functions, and new functions should not be used. The version negotiation provided by the **WSPStartup** was primarily used to allow older Winsock 1.1 applications developed for Windows 95 and Windows NT 3.51 and earlier to run with the same behavior on later versions of Windows. The *Winsock.h header* file for Winsock 1.1 support is included with the Windows SDK.

The following chart gives examples of how **WSPStartup** works in conjunction with different WS2_32.DLL and Windows Sockets service provider (SP) versions.

DLL VERSIONS	SP VERSIONS	*WVERSIONREQUESTED*	WVERSION	WHIGHVERSION	END RESULT
1.1	1.1	1.1	1.1	1.1	use 1.1
1.0 1.1	1.0	1.1	1.0	1.0	use 1.0
1.0	1.0 1.1	1.0	1.0	1.1	use 1.0
1.1	1.0 1.1	1.1	1.1	1.1	use 1.1
1.1	1.0	1.1	1.0	1.0	DLL fails
1.0	1.1	1.0	---	---	WSAVERNOTSUPPORTED
1.0 1.1	1.0 1.1	1.1	1.1	1.1	use 1.1

1.0 1.1 2.0	1.1	2.0	1.1	1.1	use 1.1
1.0 1.1 2.0	2.0	2.0	2.0	2.0	use 2.0
1.0 1.1 2.0 2.1 2.2	2.2	2.2	2.2	2.2	use 2.2

The following code fragment demonstrates how a Windows Sockets SPI client, which supports only version 2 of Windows Sockets SPI, makes a **WSPStartup** call:

```
WORD wVersionRequested;
WSPDATA WSPData;

int err;

WSPUPCALLTABLE upcallTable =
{
    /* initialize upcallTable with function pointers */
};

LPWSPPROC_TABLE lpProcTable =
{
    /* allocate memory for the ProcTable */
};

wVersionRequested = MAKEWORD( 2, 2 );

err = WSPStartup( wVersionRequested, &WSPData, lpProtocolBuffer, upcallTable, lpProcTable );
if ( err != 0 ) {
    /* Tell the user that we could not find a usable */
    /* Windows Sockets service provider.                     */
    return;
}

/* Confirm that the Windows Sockets service provider supports 2.2.*/
/* Note that if the service provider supports versions */
/* greater than 2.2 in addition to 2.2, it will still */
/* return 2.2 in wVersion since that is the version we  */
/* requested.                                           */

if ( LOBYTE( WSPData.wVersion ) != 2 ||
         HIBYTE( WSPData.wVersion ) != 2 ) {
    /* Tell the user that we could not find a usable */
    /* Windows Sockets service provider.                     */
    LPWSPCleanup( );
    return;
}

/* The Windows Sockets service provider is acceptable. Proceed. */
```

And this code fragment demonstrates how a Windows Sockets service provider that supports only version 2.2 performs the **WSPStartup** negotiation:

```
/* Make sure that the version requested is >= 2.2.  */
/* The low byte is the major version and the high   */
/* byte is the minor version.                       */

if ( (LOBYTE( wVersionRequested ) < 2) ||
     ((LOBYTE( wVersionRequested ) == 2) &&
     (HIBYTE( wVersionRequested ) < 2))) {
    return WSAVERNOTSUPPORTED;
}

/* Since we only support 2.2, set both wVersion and  */
/* wHighVersion to 2.2.                              */

lpWSPData->wVersion = MAKEWORD( 2, 2 );
lpWSPData->wHighVersion = MAKEWORD( 2, 2 );
```

Once the Windows Sockets SPI client has made a successful **WSPStartup** call, it can proceed to make other Windows Sockets SPI calls as needed. When it has finished using the services of the Windows Sockets service provider, the client must call LPWSPCleanup in order to allow the Windows Sockets service provider to free any resources allocated for the client.

The **WSPStartup** function must be called at least once by each client process, and may be called multiple times by the Winsock 2 DLL or other entities. A matching LPWSPCleanup function 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 service provider DLL.

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

A Windows Sockets SPI client can call **WSPStartup** more than once if it needs to obtain the WSPDATA structure information more than once. On each such call the client can specify any version number supported by the provider.

There must be one LPWSPCleanup call corresponding to every successful **WSPStartup** call to allow third-party DLLs to make use of a Windows Sockets provider. This means, for example, that if **WSPStartup** is called three times, the corresponding call to **LPWSPCleanup** must occur three times. The first two calls to **LPWSPCleanup** do nothing except decrement an internal counter; the final **LPWSPCleanup** call does all necessary resource deallocation.

This function (and most other service provider functions) can be invoked in a thread that started out as a 16-bit process if the client is a 16-bit Windows Sockets 1.1 client. One important limitation of 16-bit processes is that a 16-bit process cannot create threads. This is significant to service provider implementers that plan to use an internal service thread as part of the implementation.

Fortunately, there are usually only two areas where the conditions for a service thread are strong:

- In the implementation of overlapped I/O completion.
- In the implementation of LPWSPEventSelect.

Both of these areas are only accessible through new Windows Sockets 2 functions, which can only be invoked by 32-bit processes.

A service thread can be safely used if these two design rules are carefully followed:

- Use a service thread only for functionality that is unavailable to 16-bit Windows Sockets 1.1 clients.
- Create the service thread only on demand.

Several other cautions apply to the use of internal service threads. First, threads generally carry some performance penalty. Use as few as possible, and avoid thread transitions wherever possible. Second, your code should always check for errors in creating threads and fail gracefully and informatively (for example, with WSAEOPNOTSUPP) in case some execution event you did not expect results in a 16-bit process executing a code path that needs threads.

A layered service provider supplies an implementation of this function, but it is also a client of this function when it calls **WSPStartup** to initialize the next layer in the protocol chain. The call to the next layer's **WSPStartup** may happen during the execution of this layer's **WSPStartup** or it may be delayed and called on demand, such as when LPWSPSocket is called. In any case, some special considerations apply to this function's *lpProtocolInfo* parameter as it is propagated down through the layers of the protocol chain.

The layered provider searches the **ProtocolChain** of the structure referenced by *lpProtocolInfo* to determine its own location in the chain (by searching for the layer's own catalog entry **Id**) and the identity of the next element in the chain. If the next element is another layer, then, when the next layer's **WSPStartup** is called, this layer must pass to the next layer a *lpProtocolInfo* that references the same unmodified WSAProtocol_Info structure with the same unmodified chain information. However, if the next layer is the base protocol (that is, the last element in the chain), this layer performs a substitution when calling the base provider's **WSPStartup**. In this case, the base provider's **WSAPROTOCOL_INFO** structure should be referenced by the *lpProtocolInfo* parameter.

One vital benefit of this policy is that base service providers do not have to be aware of protocol chains.

This same propagation policy applies when propagating a **WSAPROTOCOL_INFO** structure through a layered sequence of other functions such as LPWSPAddressToString, LPWSPDuplicateSocket, LPWSPSocket, or LPWSPStringToAddress.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	ws2spi.h

See also

WSAProtocol_Info

LPWSPAddressToString

LPWSPStringToAddress

LPWSPCleanup

LPWSPDuplicateSocket

LPWSPEventSelect

WSPProc_Table

LPWSPSend

LPWSPSendTo

LPWSPSocket

WSPUpCallTable

WSPUPCALLTABLE structure (ws2spi.h)

10/22/2022 • 2 minutes to read • Edit Online

The **WSPUPCALLTABLE** structure contains a table of pointers to service provider upcall functions.

Syntax

```
typedef struct _WSPUPCALLTABLE {
  LPWPUCLOSEEVENT               lpWPUCloseEvent;
  LPWPUCLOSESOCKETHANDLE        lpWPUCloseSocketHandle;
  LPWPUCREATEEVENT              lpWPUCreateEvent;
  LPWPUCREATESOCKETHANDLE       lpWPUCreateSocketHandle;
  LPWPUFDISSET                  lpWPUFDIsSet;
  LPWPUGETPROVIDERPATH          lpWPUGetProviderPath;
  LPWPUMODIFYIFSHANDLE          lpWPUModifyIFSHandle;
  LPWPUPOSTMESSAGE              lpWPUPostMessage;
  LPWPUQUERYBLOCKINGCALLBACK    lpWPUQueryBlockingCallback;
  LPWPUQUERYSOCKETHANDLECONTEXT lpWPUQuerySocketHandleContext;
  LPWPUQUEUEAPC                 lpWPUQueueApc;
  LPWPURESETEVENT               lpWPUResetEvent;
  LPWPUSETEVENT                 lpWPUSetEvent;
  LPWPUOPENCURRENTTHREAD        lpWPUOpenCurrentThread;
  LPWPUCLOSETHREAD              lpWPUCloseThread;
} WSPUPCALLTABLE, *LPWSPUPCALLTABLE;
```

Members

`lpWPUCloseEvent`

Type: **LPWPUCLOSEEVENT**

A pointer to the WPUCloseEvent function.

`lpWPUCloseSocketHandle`

Type: **LPWPUCLOSESOCKETHANDLE**

A pointer to the WPUCloseSocketHandle function.

`lpWPUCreateEvent`

Type: **LPWPUCREATEEVENT**

A pointer to the WPUCreateEvent function.

`lpWPUCreateSocketHandle`

Type: **LPWPUCREATESOCKETHANDLE**

A pointer to the WPUCreateSocketHandle function.

`lpWPUFDIsSet`

Type: **LPWPUFDISSET**

A pointer to the WPUFDIsSet function.

`lpWPUGetProviderPath`

Type: **LPWPUGETPROVIDERPATH**

A pointer to the WPUGetProviderPath function.

`lpWPUModifyIFSHandle`

Type: **LPWPUMODIFYIFSHANDLE**

A pointer to the WPUModifyIFSHandle function.

`lpWPUPostMessage`

Type: **LPWPUPOSTMESSAGE**

A pointer to the WPUPostMessage function.

`lpWPUQueryBlockingCallback`

Type: **LPWPUQUERYBLOCKINGCALLBACK**

A pointer to the WPUQueryBlockingCallback function.

`lpWPUQuerySocketHandleContext`

Type: **LPWPUQUERYSOCKETHANDLECONTEXT**

A pointer to the WPUQuerySocketHandleContext function.

`lpWPUQueueApc`

Type: **LPWPUQUEUEAPC**

A pointer to the WPUQueueApc function.

`lpWPUResetEvent`

Type: **LPWPURESETEVENT**

A pointer to the WPUResetEvent function.

`lpWPUSetEvent`

Type: **LPWPUSETEVENT**

A pointer to the WPUSetEvent function.

`lpWPUOpenCurrentThread`

Type: **LPWPUOPENCURRENTTHREAD**

A pointer to the WPUOpenCurrentThread function.

`lpWPUCloseThread`

Type: **LPWPUCLOSETHREAD**

A pointer to the WPUCloseThread function.

Remarks

The **WSPUPCALLTABLE** structure contains a table of pointers to service provider upcall functions that are passed to the WSPStartup function.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	ws2spi.h

See also

WPUCloseEvent

WPUCloseSocketHandle

WPUCloseThread

WPUCreateEvent

WPUCreateSocketHandle

WPUFDIsSet

WPUGetProviderPath

WPUModifyIFSHandle

WPUOpenCurrentThread

WPUPostMessage

WPUQueryBlockingCallback

WPUQuerySocketHandleContext

WPUQueueApc

WPUResetEvent

WPUSetEvent

WSPStartup

ws2tcpip.h header

10/22/2022 • 3 minutes to read • Edit Online

This header is used by Windows Sockets 2. For more information, see:

- Windows Sockets 2

ws2tcpip.h contains the following programming interfaces:

Functions

freeaddrinfo Frees address information that the getaddrinfo function dynamically allocates in addrinfo structures.
FreeAddrInfoEx The FreeAddrInfoEx function (ws2tcpip.h) frees address information that the GetAddrInfoEx function dynamically allocates in addrinfoex structures.
FreeAddrInfoExW The FreeAddrInfoExW (Unicode) function (ws2tcpip.h) frees address information that the GetAddrInfoEx function dynamically allocates in addrinfoex structures.
FreeAddrInfoW Frees address information that the GetAddrInfoW function dynamically allocates in addrinfoW structures.
gai_strerrorA The gai_strerror function assists in printing error messages based on the EAI_* errors returned by the getaddrinfo function. (ANSI)
gai_strerrorW The gai_strerror function assists in printing error messages based on the EAI_* errors returned by the getaddrinfo function. (Unicode)
getaddrinfo Provides protocol-independent translation from an ANSI host name to an address.
GetAddrInfoExA Provides protocol-independent name resolution with additional parameters to qualify which namespace providers should handle the request. (ANSI)
GetAddrInfoExCancel Cancels an asynchronous operation by the GetAddrInfoEx function.

GetAddrInfoExOverlappedResult

Gets the return code for an OVERLAPPED structure used by an asynchronous operation for the GetAddrInfoEx function.

GetAddrInfoExW

Provides protocol-independent name resolution with additional parameters to qualify which namespace providers should handle the request. (Unicode)

GetAddrInfoW

Provides protocol-independent translation from a Unicode host name to an address.

getipv4sourcefilter

Retrieves the multicast filter state for an IPv4 socket.

getnameinfo

Provides protocol-independent name resolution from an address to an ANSI host name and from a port number to the ANSI service name.

GetNameInfoW

Provides protocol-independent name resolution from an address to a Unicode host name and from a port number to the Unicode service name.

getsourcefilter

Retrieves the multicast filter state for an IPv4 or IPv6 socket.

inet_ntop

The InetNtop function converts an IPv4 or IPv6 Internet network address into a string in Internet standard format. The ANSI version of this function is inet_ntop. (InetNtopA or inet_ntop)

inet_pton

The InetPton function 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 inet_pton. (InetPtonA or inet_pton)

InetNtopW

The InetNtop function converts an IPv4 or IPv6 Internet network address into a string in Internet standard format. The ANSI version of this function is inet_ntop. (InetNtopW)

InetPtonW

The InetPton function 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 inet_pton. (InetPtonW)

SetAddrInfoExA

Registers or deregisters a name, a service name, and associated addresses with a specific namespace provider. (ANSI)

SetAddrInfoExW

Registers or deregisters a name, a service name, and associated addresses with a specific namespace provider. (Unicode)

setipv4sourcefilter

Sets the multicast filter state for an IPv4 socket.

setsourcefilter

Sets the multicast filter state for an IPv4 or IPv6 socket.

WSADeleteSocketPeerTargetName

Removes the association between a peer target name and an IP address for a socket. After a successful return, there will be no future association between the IP address and the target name.

WSAGetFailConnectOnIcmpError

Queries the state of the **TCP_FAIL_CONNECT_ON_ICMP_ERROR** socket option.

WSAGetIcmpErrorInfo

Retrieves information about an ICMP error received on a TCP socket during connection setup.

WSAGetIPUserMtu

Retrieves the user-defined IP layer MTU for a socket.

WSAGetRecvIPEcn

TBD (WSAGetRecvIPEcn)

WSAGetUdpRecvMaxCoalescedSize

Retrieves the maximum size of a received, coalesced message for a UDP socket.

WSAGetUdpSendMessageSize

Retrieves the segmentation message size for a UDP socket.

WSAImpersonateSocketPeer

Used to impersonate the security principal corresponding to a socket peer in order to perform application-level authorization.

WSAQuerySocketSecurity

Queries information about the security applied to a connection on a socket.

WSARevertImpersonation

Terminates the impersonation of a socket peer. This must be called after calling WSAImpersonateSocketPeer and finishing any access checks.

WSASetFailConnectOnIcmpError	Sets the state of the **TCP_FAIL_CONNECT_ON_ICMP_ERROR** socket option.
WSASetIPUserMtu	Sets the user-defined IP layer MTU on a socket.
WSASetRecvIPEcn	Specifies whether the IP stack should populate the control buffer with a message containing the explicit congestion notification (ECN) codepoint of the Type of Service IPv4 header field (or Traffic Class IPv6 header field) on a received datagram.
WSASetSocketPeerTargetName	Is 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.
WSASetSocketSecurity	Enables and applies security for a socket.
WSASetUdpRecvMaxCoalescedSize	Sets the maximum size of a coalesced message set on a UDP socket.
WSASetUdpSendMessageSize	Sets the segmentation message size on a UDP socket.

freeaddrinfo function (ws2tcpip.h)

10/22/2022 • 2 minutes to read • Edit Online

The **freeaddrinfo** function frees address information that the getaddrinfo function dynamically allocates in addrinfo structures.

Syntax

```
VOID WSAAPI freeaddrinfo(
  [in] PADDRINFOA pAddrInfo
);
```

Parameters

`[in] pAddrInfo`

A pointer to the addrinfo structure or linked list of **addrinfo** structures to be freed. All dynamic storage pointed to within the **addrinfo** structure or structures is also freed.

Return value

This function does not return a value.

Remarks

The **freeaddrinfo** function frees addrinfo structures dynamically allocated by the ANSI getaddrinfo function. The **freeaddrinfo** function frees the initial **addrinfo** structure pointed to in the *ai* parameter, including any buffers to which structure members point, then continues freeing any **addrinfo** structures linked by the **ai_next** member of the **addrinfo** structure. The **freeaddrinfo** function continues freeing linked structures until a **NULL ai_next** member is encountered.

Macros in the Winsock header file define a mixed-case function name of **FreeAddrInfo** and an **ADDRINFOT** structure. This **FreeAddrInfo** function should be called with the *ai* parameter of a pointer of type **ADDRINFOT**. When UNICODE or _UNICODE is not defined, **FreeAddrInfo** is defined to **freeaddrinfo**, the ANSI version of the function, and **ADDRINFOT** is defined to the addrinfo structure. When UNICODE or _UNICODE is defined, **FreeAddrInfo** is defined to FreeAddrInfoW, the Unicode version of the function, and **ADDRINFOT** is defined to the addrinfoW structure.

Support for freeaddrinfo on earlier versions of Windows

The **freeaddrinfo** function was added to the *Ws2_32.dll* on Windows XP and later.

The FreeAddrInfoW function is the Unicode version of **freeaddrinfo**. The **FreeAddrInfoW** function was added to the *Ws2_32.dll* in Windows XP with Service Pack 2 (SP2). The **FreeAddrInfoW** function cannot be used on versions of Windows earlier than Windows XP with SP2.

Windows Phone 8: The **freeaddrinfo** function is supported for Windows Phone Store apps on Windows Phone 8 and later.

Windows 8.1 and **Windows Server 2012 R2**: The **freeaddrinfo** and FreeAddrInfoW functions are supported for Windows Store apps on Windows 8.1, Windows Server 2012 R2, and later.

Requirements

Minimum supported client	Windows 8.1, Windows Vista [desktop apps | UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps | UWP apps]
Target Platform	Windows
Header	ws2tcpip.h
DLL	Ws2_32.dll

See also

FreeAddrInfoW

GetAddrInfoW

Winsock Functions

addrinfo

addrinfoW

getaddrinfo

FreeAddrInfoEx function (ws2tcpip.h)

10/22/2022 • 2 minutes to read • Edit Online

The **FreeAddrInfoEx** function frees address information that the GetAddrInfoEx function dynamically allocates in addrinfoex structures.

Syntax

```
void WSAAPI FreeAddrInfoEx(
  [in] PADDRINFOEXA pAddrInfoEx
);
```

Parameters

`[in] pAddrInfoEx`

A pointer to the addrinfoex structure or linked list of **addrinfoex** structures to be freed. All dynamic storage pointed to within the **addrinfoex** structure or structures is also freed.

Return value

This function does not return a value.

Remarks

The **FreeAddrInfoEx** function frees addrinfoex structures dynamically allocated by the GetAddrInfoEx function. The **FreeAddrInfoEx** function frees the initial **addrinfoex** structure pointed to in the *pAddrInfo* parameter, including any buffers to which structure members point, then continues freeing any **addrinfoex** structures linked by the **ai_next** member of the **addrinfoex** structure. The **FreeAddrInfoEx** function continues freeing linked structures until a **NULL ai_next** member is encountered.

When UNICODE or _UNICODE is defined, **FreeAddrInfoEx** is defined to **FreeAddrInfoExW**, the Unicode version of the function, and **ADDRINFOEX** is defined to the addrinfoexW structure. When UNICODE or _UNICODE is not defined, **FreeAddrInfoEx** is defined to **FreeAddrInfoExA**, the ANSI version of the function, and **ADDRINFOEX** is defined to the **addrinfoexA** structure.

Windows 8.1 and **Windows Server 2012 R2**: The **FreeAddrInfoExW** function is supported for Windows Store apps on Windows 8.1, Windows Server 2012 R2, and later.

Requirements

Minimum supported client	Windows 8.1, Windows Vista [desktop apps | UWP apps]
Minimum supported server	Windows Server 2008 [desktop apps | UWP apps]
Target Platform	Windows

Header	ws2tcpip.h
Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

GetAddrInfoEx

addrinfoex

FreeAddrInfoExW function (ws2tcpip.h)

10/22/2022 • 2 minutes to read • Edit Online

The **FreeAddrInfoEx** function frees address information that the GetAddrInfoEx function dynamically allocates in addrinfoex structures.

Syntax

```
void WSAAPI FreeAddrInfoExW(
  [in] PADDRINFOEXW pAddrInfoEx
);
```

Parameters

`[in] pAddrInfoEx`

A pointer to the addrinfoex structure or linked list of **addrinfoex** structures to be freed. All dynamic storage pointed to within the **addrinfoex** structure or structures is also freed.

Return value

This function does not return a value.

Remarks

The **FreeAddrInfoEx** function frees addrinfoex structures dynamically allocated by the GetAddrInfoEx function. The **FreeAddrInfoEx** function frees the initial **addrinfoex** structure pointed to in the *pAddrInfo* parameter, including any buffers to which structure members point, then continues freeing any **addrinfoex** structures linked by the **ai_next** member of the **addrinfoex** structure. The **FreeAddrInfoEx** function continues freeing linked structures until a **NULL ai_next** member is encountered.

When UNICODE or _UNICODE is defined, **FreeAddrInfoEx** is defined to **FreeAddrInfoExW**, the Unicode version of the function, and **ADDRINFOEX** is defined to the addrinfoexW structure. When UNICODE or _UNICODE is not defined, **FreeAddrInfoEx** is defined to **FreeAddrInfoExA**, the ANSI version of the function, and **ADDRINFOEX** is defined to the **addrinfoexA** structure.

Windows 8.1 and **Windows Server 2012 R2**: The **FreeAddrInfoExW** function is supported for Windows Store apps on Windows 8.1, Windows Server 2012 R2, and later.

> **NOTE**
>
> The ws2tcpip.h header defines FreeAddrInfoEx as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see Conventions for Function Prototypes.

Requirements

Minimum supported client	Windows 8.1, Windows Vista [desktop apps | UWP apps]
Minimum supported server	Windows Server 2008 [desktop apps | UWP apps]
Target Platform	Windows
Header	ws2tcpip.h
Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

GetAddrInfoEx

addrinfoex

FreeAddrInfoW function (ws2tcpip.h)

10/22/2022 • 2 minutes to read • Edit Online

The **FreeAddrInfoW** function frees address information that the GetAddrInfoW function dynamically allocates in addrinfoW structures.

Syntax

```
VOID WSAAPI FreeAddrInfoW(
  [in] PADDRINFOW pAddrInfo
);
```

Parameters

`[in] pAddrInfo`

A pointer to the addrinfoW structure or linked list of **addrinfoW** structures to be freed. All dynamic storage pointed to within the **addrinfoW** structure or structures is also freed.

Return value

This function does not return a value.

Remarks

The **FreeAddrInfoW** function frees addrinfoW structures dynamically allocated by the Unicode GetAddrInfoW function. The **FreeAddrInfoW** function frees the initial **addrinfoW** structure pointed to in the *pAddrInfo* parameter, including any buffers to which structure members point, then continues freeing any **addrinfoW** structures linked by the **ai_next** member of the **addrinfoW** structure. The **FreeAddrInfoW** function continues freeing linked structures until a **NULL ai_next** member is encountered.

Macros in the Winsock header file define a mixed-case function name of **FreeAddrInfo** and an **ADDRINFOT** structure. This **FreeAddrInfo** function should be called with the *pAddrInfo* parameter of a pointer of type **ADDRINFOT**. When UNICODE or _UNICODE is defined, **FreeAddrInfo** is defined to **FreeAddrInfoW**, the Unicode version of the function, and **ADDRINFOT** is defined to the addrinfoW structure. When UNICODE or _UNICODE is not defined, **FreeAddrInfo** is defined to freeaddrinfo, the ANSI version of the function, and **ADDRINFOT** is defined to the addrinfo structure.

Windows 8.1 and **Windows Server 2012 R2**: This function is supported for Windows Store apps on Windows 8.1, Windows Server 2012 R2, and later.

> **NOTE**
>
> The ws2tcpip.h header defines FreeAddrInfo as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see Conventions for Function Prototypes.

Requirements

Minimum supported client	Windows 8.1, Windows Vista [desktop apps | UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps | UWP apps]
Target Platform	Windows
Header	ws2tcpip.h
Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

GetAddrInfoW

Winsock Functions

addrinfo

addrinfoW

freeaddrinfo

getaddrinfo

gai_strerrorA function (ws2tcpip.h)

10/22/2022 • 2 minutes to read • Edit Online

The **gai_strerror** function assists in printing error messages based on the EAI_* errors returned by the getaddrinfo function. Note that the **gai_strerror** function is not thread safe, and therefore, use of traditional Windows Sockets functions such as the WSAGetLastError function is recommended.

Syntax

```
char * gai_strerrorA(
  [in] int ecode
);
```

Parameters

`[in] ecode`

Error code from the list of available getaddrinfo error codes. For a complete listing of error codes, see the **getaddrinfo** function.

Return value

Returns a pointer to a string containing the error message.

Remarks

If the *ecode* parameter is not an error code value that getaddrinfo returns, the **gai_strerror** function returns a pointer to a string that indicates an unknown error.

> **NOTE**
>
> The ws2tcpip.h header defines gai_strerror as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see Conventions for Function Prototypes.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	ws2tcpip.h

See also

WSAGetLastError

Winsock Functions

Winsock Reference

getaddrinfo

gai_strerrorW function (ws2tcpip.h)

10/22/2022 • 2 minutes to read • Edit Online

The **gai_strerror** function assists in printing error messages based on the EAI_* errors returned by the getaddrinfo function. Note that the **gai_strerror** function is not thread safe, and therefore, use of traditional Windows Sockets functions such as the WSAGetLastError function is recommended.

Syntax

```
WCHAR * gai_strerrorW(
  [in] int ecode
);
```

Parameters

`[in] ecode`

Error code from the list of available getaddrinfo error codes. For a complete listing of error codes, see the **getaddrinfo** function.

Return value

None

Remarks

If the *ecode* parameter is not an error code value that getaddrinfo returns, the **gai_strerror** function returns a pointer to a string that indicates an unknown error.

> **NOTE**
>
> The ws2tcpip.h header defines gai_strerror as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see Conventions for Function Prototypes.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	ws2tcpip.h

See also

WSAGetLastError

Winsock Functions

Winsock Reference

getaddrinfo

getaddrinfo function (ws2tcpip.h)

10/22/2022 • 23 minutes to read • Edit Online

The **getaddrinfo** function provides protocol-independent translation from an ANSI host name to an address.

Syntax

```
INT WSAAPI getaddrinfo(
  [in, optional] PCSTR           pNodeName,
  [in, optional] PCSTR           pServiceName,
  [in, optional] const ADDRINFOA *pHints,
  [out]          PADDRINFOA      *ppResult
);
```

Parameters

`[in, optional] pNodeName`

A pointer to a **NULL**-terminated ANSI string that contains a host (node) name or a numeric host address string. For the Internet protocol, the numeric host address string is a dotted-decimal IPv4 address or an IPv6 hex address.

`[in, optional] pServiceName`

A pointer to a **NULL**-terminated ANSI string that contains either a service name or port number represented as a string.

A service name is a string alias for a port number. For example, "http" is an alias for port 80 defined by the Internet Engineering Task Force (IETF) as the default port used by web servers for the HTTP protocol. Possible values for the *pServiceName* parameter when a port number is not specified are listed in the following file:

`%WINDIR%\system32\drivers\etc\services`

`[in, optional] pHints`

A pointer to an addrinfo structure that provides hints about the type of socket the caller supports.

The **ai_addrlen**, **ai_canonname**, **ai_addr**, and **ai_next** members of the addrinfo structure pointed to by the *pHints* parameter must be zero or **NULL**. Otherwise the GetAddrInfoEx function will fail with WSANO_RECOVERY.

See the Remarks for more details.

`[out] ppResult`

A pointer to a linked list of one or more addrinfo structures that contains response information about the host.

Return value

Success returns zero. Failure returns a nonzero Windows Sockets error code, as found in the Windows Sockets Error Codes.

Most nonzero error codes returned by the **getaddrinfo** function map to the set of errors outlined by Internet Engineering Task Force (IETF) recommendations. The following table lists these error codes and their WSA

equivalents. It is recommended that the WSA error codes be used, as they offer familiar and comprehensive error information for Winsock programmers.

ERROR VALUE	WSA EQUIVALENT	DESCRIPTION
EAI_AGAIN	WSATRY_AGAIN	A temporary failure in name resolution occurred.
EAI_BADFLAGS	WSAEINVAL	An invalid value was provided for the **ai_flags** member of the *pHints* parameter.
EAI_FAIL	WSANO_RECOVERY	A nonrecoverable failure in name resolution occurred.
EAI_FAMILY	WSAEAFNOSUPPORT	The **ai_family** member of the *pHints* parameter is not supported.
EAI_MEMORY	WSA_NOT_ENOUGH_MEMORY	A memory allocation failure occurred.
EAI_NONAME	WSAHOST_NOT_FOUND	The name does not resolve for the supplied parameters or the *pNodeName* and *pServiceName* parameters were not provided.
EAI_SERVICE	WSATYPE_NOT_FOUND	The *pServiceName* parameter is not supported for the specified **ai_socktype** member of the *pHints* parameter.
EAI_SOCKTYPE	WSAESOCKTNOSUPPORT	The **ai_socktype** member of the *pHints* parameter is not supported.

Use the gai_strerror function to print error messages based on the EAI codes returned by the **getaddrinfo** function. The **gai_strerror** function is provided for compliance with IETF recommendations, but it is not thread safe. Therefore, use of traditional Windows Sockets functions such as WSAGetLastError is recommended.

ERROR CODE	MEANING
WSA_NOT_ENOUGH_MEMORY	There was insufficient memory to perform the operation.
WSAEAFNOSUPPORT	An address incompatible with the requested protocol was used. This error is returned if the **ai_family** member of the addrinfo structure pointed to by the *pHints* parameter is not supported.
WSAEINVAL	An invalid argument was supplied. This error is returned if an invalid value was provided for the **ai_flags** member of the addrinfo structure pointed to by the *pHints* parameter.
WSAESOCKTNOSUPPORT	The support for the specified socket type does not exist in this address family. This error is returned if the **ai_socktype** member of the addrinfo structure pointed to by the *pHints* parameter is not supported.

WSAHOST_NOT_FOUND	No such host is known. This error is returned if the name does not resolve for the supplied parameters or the *pNodeName* and *pServiceName* parameters were not provided.
WSANO_DATA	The requested name is valid, but no data of the requested type was found.
WSANO_RECOVERY	A nonrecoverable error occurred during a database lookup. This error is returned if nonrecoverable error in name resolution occurred.
WSANOTINITIALISED	A successful WSAStartup call must occur before using this function.
WSATRY_AGAIN	This is usually a temporary error during hostname resolution and means that the local server did not receive a response from an authoritative server. This error is returned when a temporary failure in name resolution occurred.
WSATYPE_NOT_FOUND	The specified class was not found. The *pServiceName* parameter is not supported for the specified **ai_socktype** member of the addrinfo structure pointed to by the *pHints* parameter.

Remarks

The **getaddrinfo** function is the ANSI version of a function that provides protocol-independent translation from host name to address. The Unicode version of this function is GetAddrInfoW. Developers are encouraged to use the **GetAddrInfoW** Unicode function rather than the **getaddrinfo** ANSI function.

The **getaddrinfo** function returns results for the **NS_DNS** namespace. The **getaddrinfo** function aggregates all responses if more than one namespace provider returns information. For use with the IPv6 and IPv4 protocol, name resolution can be by the Domain Name System (DNS), a local *hosts* file, or by other naming mechanisms for the **NS_DNS** namespace.

Another name that can be used for the **getaddrinfo** function is **GetAddrInfoA**. Macros in the *Ws2tcpip.h* header file define **GetAddrInfoA** to **getaddrinfo**.

Macros in the *Ws2tcpip.h* header file define a mixed-case function name of **GetAddrInfo** and a **ADDRINFOT** structure. This **GetAddrInfo** function should be called with the *pNodeName* and *pServiceName* parameters of a pointer of type **TCHAR** and the *pHints* and *ppResult* parameters of a pointer of type **ADDRINFOT**. When UNICODE or _UNICODE is not defined, **GetAddrInfo** is defined to **getaddrinfo**, the ANSI version of the function, and **ADDRINFOT** is defined to the addrinfo structure. When **UNICODE** or **_UNICODE** is defined, **GetAddrInfo** is defined to GetAddrInfoW, the Unicode version of the function, and **ADDRINFOT** is defined to the addrinfoW structure.

The parameter names and parameter types for the **getaddrinfo** function defined in the *Ws2tcpip.h* header file on the Platform Software Development Kit (SDK) for Windows Server 2003, and Windows XP were different.

One or both of the *pNodeName* or *pServiceName* parameters must point to a **NULL**-terminated ANSI string; generally both are provided.

Upon success, a linked list of addrinfo structures is returned in the *ppResult* parameter. The list can be processed by following the pointer provided in the **ai_next** member of each returned **addrinfo** structure until a **NULL** pointer is encountered. In each returned **addrinfo** structure, the **ai_family**, **ai_socktype**, and **ai_protocol**

members correspond to respective arguments in a socket or WSASocket function call. Also, the **ai_addr** member in each returned **addrinfo** structure points to a filled-in socket address structure, the length of which is specified in its **ai_addrlen** member.

If the *pNodeName* parameter points to a computer name, all permanent addresses for the computer that can be used as a source address are returned. On Windows Vista and later, these addresses would include all unicast IP addresses returned by the GetUnicastIpAddressTable or GetUnicastIpAddressEntry functions in which the **SkipAsSource** member is set to false in the MIB_UNICASTIPADDRESS_ROW structure.

If the *pNodeName* parameter points to a string equal to "localhost", all loopback addresses on the local computer are returned.

If the *pNodeName* parameter contains an empty string, all registered addresses on the local computer are returned.

On Windows Server 2003 and later if the *pNodeName* parameter points to a string equal to "..localmachine", all registered addresses on the local computer are returned.

If the *pNodeName* parameter refers to a cluster virtual server name, only virtual server addresses are returned. On Windows Vista and later, these addresses would include all unicast IP addresses returned by the GetUnicastIpAddressTable or GetUnicastIpAddressEntry functions in which the **SkipAsSource** member is set to true in the MIB_UNICASTIPADDRESS_ROW structure. See Windows Clustering for more information about clustering.

Windows 7 with Service Pack 1 (SP1) and Windows Server 2008 R2 with Service Pack 1 (SP1) add support to Netsh.exe for setting the SkipAsSource attribute on an IP address. This also changes the behavior such that if the **SkipAsSource** member in the MIB_UNICASTIPADDRESS_ROW structure is set to false, the IP address will be registered in DNS. If the **SkipAsSource** member is set to true, the IP address is not registered in DNS.

A hotfix is available for Windows 7 and Windows Server 2008 R2 that adds support to Netsh.exe for setting the SkipAsSource attribute on an IP address. This hotfix also changes behavior such that if the **SkipAsSource** member in the MIB_UNICASTIPADDRESS_ROW structure is set to false, the IP address will be registered in DNS. If the **SkipAsSource** member is set to true, the IP address is not registered in DNS. For more information, see Knowledge Base (KB) 2386184.

A similar hotfix is also available for Windows Vista with Service Pack 2 (SP2) and Windows Server 2008 with Service Pack 2 (SP2) that adds support to Netsh.exe for setting the SkipAsSource attribute on an IP address. This hotfix also changes behavior such that if the **SkipAsSource** member in the MIB_UNICASTIPADDRESS_ROW structure is set to false, the IP address will be registered in DNS. If the **SkipAsSource** member is set to true, the IP address is not registered in DNS.

Callers of the **getaddrinfo** function can provide hints about the type of socket supported through an addrinfo structure pointed to by the *pHints* parameter. When the *pHints* parameter is used, the following rules apply to its associated **addrinfo** structure:

- A value of **AF_UNSPEC** for **ai_family** indicates the caller will accept only the **AF_INET** and **AF_INET6** address families. Note that **AF_UNSPEC** and **PF_UNSPEC** are the same.
- A value of zero for **ai_socktype** indicates the caller will accept any socket type.
- A value of zero for **ai_protocol** indicates the caller will accept any protocol.
- The **ai_addrlen** member must be set to zero.
- The **ai_canonname** member must be set to **NULL**.
- The **ai_addr** member must be set to **NULL**.
- The **ai_next** member must be set to **NULL**.

A value of **AF_UNSPEC** for **ai_family** indicates the caller will accept any protocol family. This value can be used to return both IPv4 and IPv6 addresses for the host name pointed to by the *pNodeName* parameter. On

Windows Server 2003 and Windows XP, IPv6 addresses are returned only if IPv6 is installed on the local computer.

Other values in the addrinfo structure provided in the *pHints* parameter indicate specific requirements. For example, if the caller handles only IPv4 and does not handle IPv6, the **ai_family** member should be set to AF_INET. For another example, if the caller handles only TCP and does not handle UDP, the **ai_socktype** member should be set to **SOCK_STREAM**.

If the *pHints* parameter is a **NULL** pointer, the **getaddrinfo** function treats it as if the addrinfo structure in *pHints* were initialized with its **ai_family** member set to **AF_UNSPEC** and all other members set to zero.

On Windows Vista and later when **getaddrinfo** is called from a service, if the operation is the result of a user process calling the service, then the service should impersonate the user. This is to allow security to be properly enforced.

The **getaddrinfo** function can be used to convert a text string representation of an IP address to an addrinfo structure that contains a sockaddr structure for the IP address and other information. To be used in this way, the string pointed to by the *pNodeName* parameter must contain a text representation of an IP address and the **addrinfo** structure pointed to by the *pHints* parameter must have the AI_NUMERICHOST flag set in the **ai_flags** member. The string pointed to by the *pNodeName* parameter may contain a text representation of either an IPv4 or an IPv6 address. The text IP address is converted to an **addrinfo** structure pointed to by the *ppResult* parameter. The returned **addrinfo** structure contains a **sockaddr** structure for the IP address along with addition information about the IP address. For this method to work with an IPv6 address string on Windows Server 2003 and Windows XP, the IPv6 protocol must be installed on the local computer. Otherwise, the WSAHOST_NOT_FOUND error is returned.

Freeing Address Information from Dynamic Allocation

All information returned by the **getaddrinfo** function pointed to by the *ppResult* parameter is dynamically allocated, including all addrinfo structures, socket address structures, and canonical host name strings pointed to by **addrinfo** structures. Memory allocated by a successful call to this function must be released with a subsequent call to freeaddrinfo.

Example Code

The following code example shows how to use the **getaddrinfo** function.

```
#undef UNICODE

#include <winsock2.h>
#include <ws2tcpip.h>
#include <stdio.h>

// link with Ws2_32.lib
#pragma comment (lib, "Ws2_32.lib")

int __cdecl main(int argc, char **argv)
{

    //-----------------------------------------
    // Declare and initialize variables
    WSADATA wsaData;
    int iResult;
    INT iRetval;

    DWORD dwRetval;

    int i = 1;

    struct addrinfo *result = NULL;
    struct addrinfo *ptr = NULL;
    struct addrinfo hints;
```

```
    struct sockaddr_in  *sockaddr_ipv4;
//    struct sockaddr_in6 *sockaddr_ipv6;
    LPSOCKADDR sockaddr_ip;

    char ipstringbuffer[46];
    DWORD ipbufferlength = 46;

    // Validate the parameters
    if (argc != 3) {
        printf("usage: %s <hostname> <servicename>\n", argv[0]);
        printf("getaddrinfo provides protocol-independent translation\n");
        printf("   from an ANSI host name to an IP address\n");
        printf("%s example usage\n", argv[0]);
        printf("   %s www.contoso.com 0\n", argv[0]);
        return 1;
    }

    // Initialize Winsock
    iResult = WSAStartup(MAKEWORD(2, 2), &wsaData);
    if (iResult != 0) {
        printf("WSAStartup failed: %d\n", iResult);
        return 1;
    }

    //--------------------------------
    // Setup the hints address info structure
    // which is passed to the getaddrinfo() function
    ZeroMemory( &hints, sizeof(hints) );
    hints.ai_family = AF_UNSPEC;
    hints.ai_socktype = SOCK_STREAM;
    hints.ai_protocol = IPPROTO_TCP;

    printf("Calling getaddrinfo with following parameters:\n");
    printf("\tnodename = %s\n", argv[1]);
    printf("\tservname (or port) = %s\n\n", argv[2]);

//--------------------------------
// Call getaddrinfo(). If the call succeeds,
// the result variable will hold a linked list
// of addrinfo structures containing response
// information
    dwRetval = getaddrinfo(argv[1], argv[2], &hints, &result);
    if ( dwRetval != 0 ) {
        printf("getaddrinfo failed with error: %d\n", dwRetval);
        WSACleanup();
        return 1;
    }

    printf("getaddrinfo returned success\n");

    // Retrieve each address and print out the hex bytes
    for(ptr=result; ptr != NULL ;ptr=ptr->ai_next) {

        printf("getaddrinfo response %d\n", i++);
        printf("\tFlags: 0x%x\n", ptr->ai_flags);
        printf("\tFamily: ");
        switch (ptr->ai_family) {
            case AF_UNSPEC:
                printf("Unspecified\n");
                break;
            case AF_INET:
                printf("AF_INET (IPv4)\n");
                sockaddr_ipv4 = (struct sockaddr_in *) ptr->ai_addr;
                printf("\tIPv4 address %s\n",
                    inet_ntoa(sockaddr_ipv4->sin_addr) );
                break;
            case AF_INET6:
                printf("AF_INET6 (IPv6)\n");
```

```
            // the InetNtop function is available on Windows Vista and later
            // sockaddr_ipv6 = (struct sockaddr_in6 *) ptr->ai_addr;
            // printf("\tIPv6 address %s\n",
            //    InetNtop(AF_INET6, &sockaddr_ipv6->sin6_addr, ipstringbuffer, 46) );

            // We use WSAAddressToString since it is supported on Windows XP and later
            sockaddr_ip = (LPSOCKADDR) ptr->ai_addr;
            // The buffer length is changed by each call to WSAAddresstoString
            // So we need to set it for each iteration through the loop for safety
            ipbufferlength = 46;
            iRetval = WSAAddressToString(sockaddr_ip, (DWORD) ptr->ai_addrlen, NULL,
                ipstringbuffer, &ipbufferlength );
            if (iRetval)
                printf("WSAAddressToString failed with %u\n", WSAGetLastError() );
            else
                printf("\tIPv6 address %s\n", ipstringbuffer);
            break;
        case AF_NETBIOS:
            printf("AF_NETBIOS (NetBIOS)\n");
            break;
        default:
            printf("Other %ld\n", ptr->ai_family);
            break;
    }
    printf("\tSocket type: ");
    switch (ptr->ai_socktype) {
        case 0:
            printf("Unspecified\n");
            break;
        case SOCK_STREAM:
            printf("SOCK_STREAM (stream)\n");
            break;
        case SOCK_DGRAM:
            printf("SOCK_DGRAM (datagram) \n");
            break;
        case SOCK_RAW:
            printf("SOCK_RAW (raw) \n");
            break;
        case SOCK_RDM:
            printf("SOCK_RDM (reliable message datagram)\n");
            break;
        case SOCK_SEQPACKET:
            printf("SOCK_SEQPACKET (pseudo-stream packet)\n");
            break;
        default:
            printf("Other %ld\n", ptr->ai_socktype);
            break;
    }
    printf("\tProtocol: ");
    switch (ptr->ai_protocol) {
        case 0:
            printf("Unspecified\n");
            break;
        case IPPROTO_TCP:
            printf("IPPROTO_TCP (TCP)\n");
            break;
        case IPPROTO_UDP:
            printf("IPPROTO_UDP (UDP) \n");
            break;
        default:
            printf("Other %ld\n", ptr->ai_protocol);
            break;
    }
    printf("\tLength of this sockaddr: %d\n", ptr->ai_addrlen);
    printf("\tCanonical name: %s\n", ptr->ai_canonname);
}

freeaddrinfo(result);
WSACleanup();
```

```
    return 0;
}
```

> **Note** Ensure that the development environment targets the newest version of *Ws2tcpip.h* which includes structure and function definitions for addrinfo and **getaddrinfo**, respectively.

Internationalized Domain Names

Internet host names typically consist of a very restricted set of characters:

- Upper and lower case ASCII letters from the English alphabet.
- Digits from 0 to 9.
- ASCII hyphen characters.

With the growth of the Internet, there is a growing need to identify Internet host names for other languages not represented by the ASCII character set. Identifiers which facilitate this need and allow non-ASCII characters (Unicode) to be represented as special ASCII character strings are known as Internationalized Domain Names (IDNs). A mechanism called Internationalizing Domain Names in Applications (IDNA) is used to handle IDNs in a standard fashion. The specifications for IDNs and IDNA are documented in RFC 3490, RTF 5890, and RFC 6365 published by the Internet Engineering Task Force (IETF).

On Windows 8 and Windows Server 2012, the **getaddrinfo** function provides support for Internationalized Domain Name (IDN) parsing applied to the name passed in the *pNodeName* parameter. Winsock performs Punycode/IDN encoding and conversion. This behavior can be disabled using the **AI_DISABLE_IDN_ENCODING** flag discussed below.

On Windows 7 and Windows Server 2008 R2 or earlier, the **getaddrinfo** function does not currently provide support IDN parsing applied to the name passed in the *pNodeName* parameter. Winsock does not perform any Punycode/IDN conversion. The wide character version of the **GetAddrInfo** function does not use Punycode to convert an IDN as per RFC 3490. The wide character version of the **GetAddrInfo** function when querying DNS encodes the Unicode name in UTF-8 format, the format used by Microsoft DNS servers in an enterprise environment.

Several functions on Windows Vista and later support conversion between Unicode labels in an IDN to their ASCII equivalents. The resulting representation of each Unicode label contains only ASCII characters and starts with the xn-- prefix if the Unicode label contained any non-ASCII characters. The reason for this is to support existing DNS servers on the Internet, since some DNS tools and servers only support ASCII characters (see RFC 3490).

The IdnToAscii function uses Punycode to convert an IDN to the ASCII representation of the original Unicode string using the standard algorithm defined in RFC 3490. The IdnToUnicode function converts the ASCII form of an IDN to the normal Unicode UTF-16 encoding syntax. For more information and links to related draft standards, see Handling Internationalized Domain Names (IDNs).

The IdnToAscii function can be used to convert an IDN name to the ASCII form that then can be passed in the *pNodeName* parameter to the **getaddrinfo** function. To pass this IDN name to the **GetAddrInfo** function when the wide character version of this function is used (when UNICODE or _UNICODE is defined), you can use the MultiByteToWideChar function to convert the **CHAR** string into a **WCHAR** string.

Use of ai_flags in the pHints parameter

Flags in the **ai_flags** member of the optional addrinfo structure provided in the *pHints* parameter modify the behavior of the function.

These flag bits are defined in the *Ws2def.h* header file on the Microsoft Windows Software Development Kit (SDK) for Windows 7. These flag bits are defined in the *Ws2tcpip.h* header file on the Windows SDK for Windows Server 2008 and Windows Vista. These flag bits are defined in the *Ws2tcpip.h* header file on the Platform SDK for Windows Server 2003, and Windows XP.

The flag bits can be a combination of the following:

FLAG BITS	DESCRIPTION
AI_PASSIVE	Setting the **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 *pNodeName* is a **NULL** pointer, the IP address portion of the socket address structure is set to **INADDR_ANY** for IPv4 addresses and **IN6ADDR_ANY_INIT** for IPv6 addresses. When the **AI_PASSIVE** flag is not set, the returned socket address structure is ready for a call to the connect function for a connection-oriented protocol, or ready for a call to either the **connect**, sendto, or send functions for a connectionless protocol. If the *pNodeName* parameter is a **NULL** pointer in this case, the IP address portion of the socket address structure is set to the loopback address.
AI_CANONNAME	If neither **AI_CANONNAME** nor **AI_NUMERICHOST** is used, the **getaddrinfo** function attempts resolution. If a literal string is passed **getaddrinfo** attempts to convert the string, and if a host name is passed the **getaddrinfo** function attempts to resolve the name to an address or multiple addresses. When the **AI_CANONNAME** bit is set, the *pNodeName* parameter cannot be **NULL**. Otherwise the **getaddrinfo** function will fail with WSANO_RECOVERY. When the **AI_CANONNAME** bit is set and the **getaddrinfo** function returns success, the **ai_canonname** member in the *ppResult* parameter points to a **NULL**-terminated string that contains the canonical name of the specified node. **Note** The **getaddrinfo** function can return success when the **AI_CANONNAME** flag is set, yet the **ai_canonname** member in the associated addrinfo structure is **NULL**. Therefore, the recommended use of the **AI_CANONNAME** flag includes testing whether the **ai_canonname** member in the associated **addrinfo** structure is **NULL**.
AI_NUMERICHOST	When the **AI_NUMERICHOST** bit is set, the *pNodeName* parameter must contain a non-**NULL** numeric host address string, otherwise the **EAI_NONAME** error is returned. This flag prevents a name resolution service from being called.

AI_NUMERICSERV	When the **AI_NUMERICSERV** bit is set, the *pServiceName* parameter must contain a non-**NULL** numeric port number, otherwise the **EAI_NONAME** error is returned. This flag prevents a name resolution service from being called. The **AI_NUMERICSERV** flag is defined on Windows SDK for Windows Vista and later. The **AI_NUMERICSERV** flag is not supported by Microsoft providers.
AI_ALL	If the **AI_ALL** bit is set, a request is made for IPv6 addresses and IPv4 addresses with **AI_V4MAPPED**. The **AI_ALL** flag is defined on the Windows SDK for Windows Vista and later. The **AI_ALL** flag is supported on Windows Vista and later.
AI_ADDRCONFIG	If the **AI_ADDRCONFIG** bit is set, **getaddrinfo** will resolve only if a global address is configured. If **AI_ADDRCONFIG** flag is specified, IPv4 addresses shall be returned only if an IPv4 address is configured on the local system, and IPv6 addresses shall be returned only if an IPv6 address is configured on the local system. The IPv4 or IPv6 loopback address is not considered a valid global address. The **AI_ADDRCONFIG** flag is defined on the Windows SDK for Windows Vista and later. The **AI_ADDRCONFIG** flag is supported on Windows Vista and later.
AI_V4MAPPED	If the **AI_V4MAPPED** bit is set and a request for IPv6 addresses fails, a name service request is made for IPv4 addresses and these addresses are converted to IPv4-mapped IPv6 address format. The **AI_V4MAPPED** flag is defined on the Windows SDK for Windows Vista and later. The **AI_V4MAPPED** flag is supported on Windows Vista and later.
AI_NON_AUTHORITATIVE	If the **AI_NON_AUTHORITATIVE** bit is set, the **NS_EMAIL** namespace provider returns both authoritative and non-authoritative results. If the **AI_NON_AUTHORITATIVE** bit is not set, the **NS_EMAIL** namespace provider returns only authoritative results. The **AI_NON_AUTHORITATIVE** flag is defined on the Windows SDK for Windows Vista and later. The **AI_NON_AUTHORITATIVE** flag is supported on Windows Vista and later and applies only to the **NS_EMAIL** namespace.
AI_SECURE	If the **AI_SECURE** bit is set, the **NS_EMAIL** namespace provider will return results that were obtained with enhanced security to minimize possible spoofing. The **AI_SECURE** flag is defined on the Windows SDK for Windows Vista and later. The **AI_SECURE** flag is supported on Windows Vista and later and applies only to the **NS_EMAIL** namespace.

AI_RETURN_PREFERRED_NAMES	If the **AI_RETURN_PREFERRED_NAMES** is set, then no name should be provided in the *pNodeName* parameter. The **NS_EMAIL** namespace provider will return preferred names for publication. The **AI_RETURN_PREFERRED_NAMES** flag is defined on the Windows SDK for Windows Vista and later. The **AI_RETURN_PREFERRED_NAMES** flag is supported on Windows Vista and later and applies only to the **NS_EMAIL** namespace.
AI_FQDN	If the **AI_FQDN** is set and a flat name (single label) is specified, **getaddrinfo** will return the fully qualified domain name that the name eventually resolved to. The fully qualified domain name is returned in the **ai_canonname** member in the associated addrinfo structure. This is different than **AI_CANONNAME** bit flag that returns the canonical name registered in DNS which may be different than the fully qualified domain name that the flat name resolved to. Only one of the **AI_FQDN** and **AI_CANONNAME** bits can be set. The **getaddrinfo** function will fail if both flags are present with **EAI_BADFLAGS**. When the **AI_FQDN** bit is set, the *pNodeName* parameter cannot be **NULL**. Otherwise the GetAddrInfoEx function will fail with WSANO_RECOVERY. **Windows 7:** The **AI_FQDN** flag is defined on the Windows SDK for Windows 7 and later. The **AI_FQDN** flag is supported on Windows 7 and later.
AI_FILESERVER	If the **AI_FILESERVER** is set, this is a hint to the namespace provider that the hostname being queried is being used in file share scenario. The namespace provider may ignore this hint. **Windows 7:** The **AI_FILESERVER** flag is defined on the Windows SDK for Windows 7 and later. The **AI_FILESERVER** flag is supported on Windows 7 and later.

Example code using AI_NUMERICHOST

The following code example shows how to use the **getaddrinfo** function to convert a text string representation of an IP address to an addrinfo structure that contains a sockaddr structure for the IP address and other information.

```
#undef UNICODE

#include <winsock2.h>
#include <ws2tcpip.h>
#include <stdio.h>

// link with Ws2_32.lib
#pragma comment (lib, "Ws2_32.lib")

int __cdecl main(int argc, char **argv)
{

    //-----------------------------------------
    // Declare and initialize variables
    WSADATA wsaData;
    int iResult;
```

```
    DWORD dwRetval;

    int i = 1;

    struct addrinfo *result = NULL;
    struct addrinfo *ptr = NULL;
    struct addrinfo hints;


    // Validate the parameters
    if (argc != 2) {
        printf("usage: %s <IP Address String>\n", argv[0]);
        printf("  getaddrinfo determines the IP binary network address\n");
        printf("       %s 207.46.197.32\n", argv[0]);  /* www.contoso.com */
        return 1;
    }
    // Initialize Winsock
    iResult = WSAStartup(MAKEWORD(2, 2), &wsaData);
    if (iResult != 0) {
        printf("WSAStartup failed: %d\n", iResult);
        return 1;
    }

    //--------------------------------
    // Setup the hints address info structure
    // which is passed to the getaddrinfo() function
    ZeroMemory( &hints, sizeof(hints) );
    hints.ai_flags = AI_NUMERICHOST;
    hints.ai_family = AF_UNSPEC;
//    hints.ai_socktype = SOCK_STREAM;
//    hints.ai_protocol = IPPROTO_TCP;


//--------------------------------
// Call getaddrinfo(). If the call succeeds,
// the result variable will hold a linked list
// of addrinfo structures containing response
// information
    dwRetval = getaddrinfo(argv[1], NULL, &hints, &result);
    if ( dwRetval != 0 ) {
        printf("getaddrinfo failed with error: %d\n", dwRetval);
        WSACleanup();
        return 1;
    }

    printf("getaddrinfo returned success\n");

    // Retrieve each address and print out the hex bytes
    for(ptr=result; ptr != NULL ;ptr=ptr->ai_next) {

        printf("getaddrinfo response %d\n", i++);
        printf("\tFlags: 0x%x\n", ptr->ai_flags);
        printf("\tFamily: ");
        switch (ptr->ai_family) {
            case AF_UNSPEC:
                printf("Unspecified\n");
                break;
            case AF_INET:
                printf("AF_INET (IPv4)\n");
                break;
            case AF_INET6:
                printf("AF_INET6 (IPv6)\n");
                break;
            case AF_NETBIOS:
                printf("AF_NETBIOS (NetBIOS)\n");
                break;
            default:
                printf("Other %ld\n", ptr->ai_family);
                break;
```

```
            break;
        }
        printf("\tSocket type: ");
        switch (ptr->ai_socktype) {
            case 0:
                printf("Unspecified\n");
                break;
            case SOCK_STREAM:
                printf("SOCK_STREAM (stream)\n");
                break;
            case SOCK_DGRAM:
                printf("SOCK_DGRAM (datagram) \n");
                break;
            case SOCK_RAW:
                printf("SOCK_RAW (raw) \n");
                break;
            case SOCK_RDM:
                printf("SOCK_RDM (reliable message datagram)\n");
                break;
            case SOCK_SEQPACKET:
                printf("SOCK_SEQPACKET (pseudo-stream packet)\n");
                break;
            default:
                printf("Other %ld\n", ptr->ai_socktype);
                break;
        }
        printf("\tProtocol: ");
        switch (ptr->ai_protocol) {
            case 0:
                printf("Unspecified\n");
                break;
            case IPPROTO_TCP:
                printf("IPPROTO_TCP (TCP)\n");
                break;
            case IPPROTO_UDP:
                printf("IPPROTO_UDP (UDP) \n");
                break;
            default:
                printf("Other %ld\n", ptr->ai_protocol);
                break;
        }
        printf("\tLength of this sockaddr: %d\n", ptr->ai_addrlen);
        printf("\tCanonical name: %s\n", ptr->ai_canonname);
    }

    freeaddrinfo(result);
    WSACleanup();

    return 0;
}
```

Support for getaddrinfo on Windows 2000 and older versions

The **getaddrinfo** function was added to the Ws2_32.dll on Windows XP and later. To execute an application that uses this function on earlier versions of Windows, then you need to include the *Ws2tcpip.h* and *Wspiapi.h* files. When the *Wspiapi.h* include file is added, the **getaddrinfo** function is defined to the **WspiapiGetAddrInfo** inline function in the *Wspiapi.h* file. At runtime, the **WspiapiGetAddrInfo** function is implemented in such a way that if the Ws2_32.dll or the Wship6.dll (the file containing **getaddrinfo** in the IPv6 Technology Preview for Windows 2000) does not include **getaddrinfo**, then a version of **getaddrinfo** is implemented inline based on code in the Wspiapi.h header file. This inline code will be used on older Windows platforms that do not natively support the **getaddrinfo** function.

The IPv6 protocol is supported on Windows 2000 when the IPv6 Technology Preview for Windows 2000 is installed. Otherwise **getaddrinfo** support on versions of Windows earlier than Windows XP is limited to handling IPv4 name resolution.

The GetAddrInfoW function is the Unicode version of **getaddrinfo**. The **GetAddrInfoW** function was added to the Ws2_32.dll in Windows XP with Service Pack 2 (SP2). The **GetAddrInfoW** function cannot be used on versions of Windows earlier than Windows XP with SP2.

Windows Phone 8: This function is supported for Windows Phone Store apps on Windows Phone 8 and later.

Windows 8.1 and **Windows Server 2012 R2**: This function is supported for Windows Store apps on Windows 8.1, Windows Server 2012 R2, and later.

Requirements

Minimum supported client	Windows 8.1, Windows Vista [desktop apps | UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps | UWP apps]
Target Platform	Windows
Header	ws2tcpip.h
Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

GetAddrInfoEx

GetAddrInfoW

IdnToAscii

IdnToUnicode

WSAGetLastError

WSASocket

Winsock Functions

Winsock Reference

addrinfo

addrinfoW

addrinfoex

addrinfoex2

bind

connect

freeaddrinfo

gai_strerror

send

sendto

socket

GetAddrInfoExA function (ws2tcpip.h)

10/22/2022 • 30 minutes to read • Edit Online

The **GetAddrInfoEx** function provides protocol-independent name resolution with additional parameters to qualify which namespace providers should handle the request.

Syntax

```
INT WSAAPI GetAddrInfoExA(
  [in, optional]  PCSTR                              pName,
  [in, optional]  PCSTR                              pServiceName,
  [in]            DWORD                              dwNameSpace,
  [in, optional]  LPGUID                             lpNspId,
  [in, optional]  const ADDRINFOEXA                  *hints,
  [out]           PADDRINFOEXA                       *ppResult,
  [in, optional]  timeval                            *timeout,
  [in, optional]  LPOVERLAPPED                       lpOverlapped,
  [in, optional]  LPLOOKUPSERVICE_COMPLETION_ROUTINE lpCompletionRoutine,
  [out, optional] LPHANDLE                           lpNameHandle
);
```

Parameters

`[in, optional] pName`

A pointer to a **NULL**-terminated string containing a host (node) name or a numeric host address string. For the Internet protocol, the numeric host address string is a dotted-decimal IPv4 address or an IPv6 hex address.

`[in, optional] pServiceName`

A pointer to an optional **NULL**-terminated string that contains either a service name or port number represented as a string.

A service name is a string alias for a port number. For example, "http" is an alias for port 80 defined by the Internet Engineering Task Force (IETF) as the default port used by web servers for the HTTP protocol. Possible values for the *pServiceName* parameter when a port number is not specified are listed in the following file:

`%WINDIR%\system32\drivers\etc\services`

`[in] dwNameSpace`

An optional namespace identifier that determines which namespace providers are queried. Passing a specific namespace identifier will result in only namespace providers that support the specified namespace being queried. Specifying **NS_ALL** will result in all installed and active namespace providers being queried.

Options for the *dwNameSpace* parameter are listed in the *Winsock2.h* include file. Several namespace providers are added on Windows Vista and later. Other namespace providers can be installed, so the following possible values are only those commonly available. Many other values are possible.

VALUE	MEANING

NS_ALL 0	All installed and active namespaces.
NS_DNS 12	The domain name system (DNS) namespace.
NS_NETBT 13	The NetBIOS over TCP/IP (NETBT) namespace.
NS_WINS 14	The Windows Internet Naming Service (NS_WINS) namespace.
NS_NLA 15	The network location awareness (NLA) namespace. This namespace identifier is supported on Windows XP and later.
NS_BTH 16	The Bluetooth namespace. This namespace identifier is supported on Windows Vista and later.
NS_NTDS 32	The Windows NT Directory Services (NS_NTDS) namespace.
NS_EMAIL 37	The email namespace. This namespace identifier is supported on Windows Vista and later.
NS_PNRPNAME 38	The peer-to-peer namespace for a specific peer name. This namespace identifier is supported on Windows Vista and later.
NS_PNRPCLOUD 39	The peer-to-peer namespace for a collection of peer names. This namespace identifier is supported on Windows Vista and later.

`[in, optional] lpNspId`

A pointer to an optional GUID of a specific namespace provider to query in the case where multiple namespace providers are registered under a single namespace such as **NS_DNS**. Passing the GUID for specific namespace provider will result in only the specified namespace provider being queried. The WSAEnumNameSpaceProviders function can be called to retrieve the GUID for a namespace provider.

`[in, optional] hints`

A pointer to an addrinfoex structure that provides hints about the type of socket the caller supports.

The **ai_addrlen**, **ai_canonname**, **ai_addr**, and **ai_next** members of the addrinfoex structure pointed to by the *pHints* parameter must be zero or **NULL**. Otherwise the **GetAddrInfoEx** function will fail with

WSANO_RECOVERY.

See the Remarks for more details.

`[out] ppResult`

A pointer to a linked list of one or more addrinfoex structures that contains response information about the host.

`[in, optional] timeout`

An optional parameter indicating the time, in milliseconds, to wait for a response from the namespace provider before aborting the call.

This parameter is only supported when the **UNICODE** or **_UNICODE** macro has been defined in the sources before calling the **GetAddrInfoEx** function. Otherwise, this parameter is currently reserved and must be set to **NULL** since a *timeout* option is not supported.

`[in, optional] lpOverlapped`

An optional pointer to an overlapped structure used for asynchronous operation.

This parameter is only supported when the **UNICODE** or **_UNICODE** macro has been defined in the sources before calling the **GetAddrInfoEx** function.

On Windows 8 and Windows Server 2012, if no *lpCompletionRoutine* parameter is specified, the **hEvent** member of the **OVERLAPPED** structure must be set to a manual-reset event to be called upon completion of an asynchronous call. If a completion routine has been specified, the **hEvent** member must be NULL. When the event specified by **hEvent** has been set, the result of the operation can be retrieved by calling GetAddrInfoExOverlappedResult function.

On Windows 8 and Windows Server 2012 whenever the **UNICODE** or **_UNICODE** macro is not defined, this parameter is currently reserved and must be set to **NULL**.

On Windows 7 and Windows Server 2008 R2 or earlier, this parameter is currently reserved and must be set to **NULL** since asynchronous operations are not supported.

`[in, optional] lpCompletionRoutine`

An optional pointer to a function to be invoked upon successful completion for asynchronous operations.

This parameter is only supported when the **UNICODE** or **_UNICODE** macro has been defined in the sources before calling the **GetAddrInfoEx** function.

On Windows 8 and Windows Server 2012, if this parameter is specified, it must be a pointer to a function with the following signature:

```
typedef
void
(CALLBACK * LPLOOKUPSERVICE_COMPLETION_ROUTINE)(
    __in      DWORD    dwError,
    __in      DWORD    dwBytes,
    __in      LPWSAOVERLAPPED lpOverlapped
    );
```

When the asynchronous operation has completed, the completion routine will be invoked with *lpOverlapped* parameter set to the value of *lpOverlapped* parameter passed to **GetAddrInfoEx**. The **Pointer** member of the OVERLAPPED structure will be set to the value of the *ppResult* parameter of the original call. If the **Pointer** member points to a non-NULL pointer to the addrinfoex structure, it is the caller's responsibility to call

FreeAddrInfoEx to free the **addrinfoex** structure. The *dwError* parameter passed to the completion routine will be set to a Winsock error code. The *dwBytes* parameter is reserved for future use and must be ignored.

On Windows 8 and Windows Server 2012 whenever the **UNICODE** or **_UNICODE** macro is not defined, this parameter is currently reserved and must be set to **NULL**.

On Windows 7 and Windows Server 2008 R2 or earlier, this parameter is currently reserved and must be set to **NULL** since asynchronous operations are not supported.

`[out, optional] lpNameHandle`

An optional pointer used only for asynchronous operations.

This parameter is only supported when the **UNICODE** or **_UNICODE** macro has been defined in the sources before calling the **GetAddrInfoEx** function.

On Windows 8 and Windows Server 2012, if the **GetAddrInfoEx** function will complete asynchronously, the pointer returned in this field may be used with the **GetAddrInfoExCancel** function. The handle returned is valid when **GetAddrInfoEx** returns until the completion routine is called, the event is triggered, or GetAddrInfoExCancel function is called with this handle.

On Windows 8 and Windows Server 2012 whenever the **UNICODE** or **_UNICODE** macro is not defined, this parameter is currently reserved and must be set to **NULL**.

On Windows 7 and Windows Server 2008 R2 or earlier, this parameter is currently reserved and must be set to **NULL** since asynchronous operations are not supported.

Return value

On success, **GetAddrInfoEx** returns **NO_ERROR** (0). Failure returns a nonzero Windows Sockets error code, as found in the Windows Sockets Error Codes.

Most nonzero error codes returned by the **GetAddrInfoEx** function map to the set of errors outlined by Internet Engineering Task Force (IETF) recommendations. The following table shows these error codes and their WSA equivalents. It is recommended that the WSA error codes be used, as they offer familiar and comprehensive error information for Winsock programmers.

ERROR VALUE	WSA EQUIVALENT	DESCRIPTION
EAI_AGAIN	WSATRY_AGAIN	A temporary failure in name resolution occurred.
EAI_BADFLAGS	WSAEINVAL	An invalid parameter was provided. This error is returned if any of the reserved parameters are not **NULL**. This error is also returned if an invalid value was provided for the **ai_flags** member of the *pHints* parameter.
EAI_FAIL	WSANO_RECOVERY	A nonrecoverable failure in name resolution occurred.
EAI_FAMILY	WSAEAFNOSUPPORT	The **ai_family** member of the *pHints* parameter is not supported.
EAI_MEMORY	WSA_NOT_ENOUGH_MEMORY	A memory allocation failure occurred.

EAI_NONAME	WSAHOST_NOT_FOUND	The name does not resolve for the supplied parameters or the *pName* and *pServiceName* parameters were not provided.
EAI_SERVICE	WSATYPE_NOT_FOUND	The *pServiceName* parameter is not supported for the specified **ai_socktype** member of the *pHints* parameter.
EAI_SOCKTYPE	WSAESOCKTNOSUPPORT	The **ai_socktype** member of the *pHints* parameter is not supported.

Use the gai_strerror function to print error messages based on the EAI codes returned by the **GetAddrInfoEx** function. The **gai_strerror** function is provided for compliance with IETF recommendations, but it is not thread safe. Therefore, use of traditional Windows Sockets functions such as WSAGetLastError is recommended.

ERROR CODE	MEANING
WSA_NOT_ENOUGH_MEMORY	There was insufficient memory to perform the operation.
WSAEAFNOSUPPORT	An address incompatible with the requested protocol was used. This error is returned if the **ai_family** member of the addrinfoex structure pointed to by the *pHints* parameter is not supported.
WSAEINVAL	An invalid argument was supplied. This error is returned if an invalid value was provided for the **ai_flags** member of the addrinfoex structure pointed to by the *pHints* parameter. This error is also returned when the *dwNameSpace* parameter is NS_PNRPNAME or NS_PNRPCLOUD and the peer-to-peer name service is not operating.
WSAESOCKTNOSUPPORT	The support for the specified socket type does not exist in this address family. This error is returned if the **ai_socktype** member of the addrinfoex structure pointed to by the *pHints* parameter is not supported.
WSAHOST_NOT_FOUND	No such host is known. This error is returned if the name does not resolve for the supplied parameters or the *pName* and *pServiceName* parameters were not provided.
WSANO_DATA	The requested name is valid, but no data of the requested type was found.
WSANO_RECOVERY	A nonrecoverable error occurred during a database lookup. This error is returned if nonrecoverable error in name resolution occurred.
WSANOTINITIALISED	A successful WSAStartup call must occur before using this function.

WSASERVICE_NOT_FOUND	No such service is known. The service cannot be found in the specified name space. This error is returned if the *pName* or *pServiceName* parameter is not found for the namespace specified in the *dwNameSpace* parameter or the namespace specified in the *dwNameSpace* parameter is not installed.
WSATRY_AGAIN	This is usually a temporary error during hostname resolution and means that the local server did not receive a response from an authoritative server. This error is returned when a temporary failure in name resolution occurred.
WSATYPE_NOT_FOUND	The specified class was not found. The *pServiceName* parameter is not supported for the specified **ai_socktype** member of the addrinfoex structure pointed to by the *pHints* parameter.

Remarks

The **GetAddrInfoEx** function provides protocol-independent translation from host name to address and from service name to port number. The **GetAddrInfoEx** function is an enhanced version of the getaddrinfo and GetAddrInfoW functions. The **GetAddrInfoEx** function allows specifying the namespace provider to resolve the query.

The **GetAddrInfoEx** function aggregates and returns results from multiple namespace providers, unless a specific namespace provider is specified. For use with the IPv6 and IPv4 protocol, name resolution can be by the Domain Name System (DNS), a local *hosts* file, an email provider (the **NS_EMAIL** namespace), or by other naming mechanisms.

When UNICODE or _UNICODE is defined, **GetAddrInfoEx** is defined to **GetAddrInfoExW**, the Unicode version of this function. The string parameters are defined to the **PWSTR** data type and the **ADDRINFOEXW** structure is used. On Windows 8 and Windows Server 2012, the *timeout*, *lpOverlapped*, *lpCompletionRoutine*, and *lpNameHandle* parameters may be used to call the **GetAddrInfoEx** function so that it can complete asynchronously.

When UNICODE or _UNICODE is not defined, **GetAddrInfoEx** is defined to **GetAddrInfoExA**, the ANSI version of this function. The string parameters are of the **PCSTR** data type and the **ADDRINFOEXA** structure is used. The *timeout*, *lpOverlapped*, *lpCompletionRoutine*, and *lpNameHandle* parameters must be set to **NULL**.

One or both of the *pName* or *pServiceName* parameters must point to a **NULL**-terminated string. Generally both are provided.

Upon success, a linked list of addrinfoex structures is returned in the *ppResult* parameter. The list can be processed by following the pointer provided in the **ai_next** member of each returned **addrinfoex** structure until a **NULL** pointer is encountered. In each returned **addrinfoex** structure, the **ai_family**, **ai_socktype**, and **ai_protocol** members correspond to respective arguments in a socket or WSASocket function call. Also, the **ai_addr** member in each returned **addrinfoex** structure points to a filled-in socket address structure, the length of which is specified in its **ai_addrlen** member.

If the *pName* parameter points to a computer name, all permanent addresses for the computer that can be used as a source address are returned. On Windows Vista and later, these addresses would include all unicast IP addresses returned by the GetUnicastIpAddressTable or GetUnicastIpAddressEntry functions in which the **SkipAsSource** member is set to false in the MIB_UNICASTIPADDRESS_ROW structure.

If the *pName* parameter points to a string equal to "localhost", all loopback addresses on the local computer are returned.

If the *pName* parameter contains an empty string, all registered addresses on the local computer are returned.

On Windows Server 2003 and later if the *pName* parameter points to a string equal to "..localmachine", all registered addresses on the local computer are returned.

If the *pName* parameter refers to a cluster virtual server name, only virtual server addresses are returned. On Windows Vista and later, these addresses would include all unicast IP addresses returned by the GetUnicastIpAddressTable or GetUnicastIpAddressEntry functions in which the **SkipAsSource** member is set to true in the MIB_UNICASTIPADDRESS_ROW structure. See Windows Clustering for more information about clustering.

Windows 7 with Service Pack 1 (SP1) and Windows Server 2008 R2 with Service Pack 1 (SP1) add support to Netsh.exe for setting the SkipAsSource attribute on an IP address. This also changes the behavior such that if the **SkipAsSource** member in the MIB_UNICASTIPADDRESS_ROW structure is set to false, the IP address will be registered in DNS. If the **SkipAsSource** member is set to true, the IP address is not registered in DNS.

A hotfix is available for Windows 7 and Windows Server 2008 R2 that adds support to Netsh.exe for setting the SkipAsSource attribute on an IP address. This hotfix also changes behavior such that if the **SkipAsSource** member in the MIB_UNICASTIPADDRESS_ROW structure is set to false, the IP address will be registered in DNS. If the **SkipAsSource** member is set to true, the IP address is not registered in DNS. For more information, see Knowledge Base (KB) 2386184.

A similar hotfix is also available for Windows Vista with Service Pack 2 (SP2) and Windows Server 2008 with Service Pack 2 (SP2) that adds support to Netsh.exe for setting the SkipAsSource attribute on an IP address. This hotfix also changes behavior such that if the **SkipAsSource** member in the MIB_UNICASTIPADDRESS_ROW structure is set to false, the IP address will be registered in DNS. If the **SkipAsSource** member is set to true, the IP address is not registered in DNS.

Callers of the **GetAddrInfoEx** function can provide hints about the type of socket supported through an addrinfoex structure pointed to by the *pHints* parameter. When the *pHints* parameter is used, the following rules apply to its associated **addrinfoex** structure:

- A value of **AF_UNSPEC** for **ai_family** indicates the caller will accept only the **AF_INET** and **AF_INET6** address families. Note that **AF_UNSPEC** and **PF_UNSPEC** are the same.
- A value of zero for **ai_socktype** indicates the caller will accept any socket type.
- A value of zero for **ai_protocol** indicates the caller will accept any protocol.
- The **ai_addrlen** member must be set to zero.
- The **ai_canonname** member must be set to **NULL**.
- The **ai_addr** member must be set to **NULL**.
- The **ai_next** member must be set to **NULL**.

Other values in the addrinfoex structure provided in the *pHints* parameter indicate specific requirements. For example, if the caller handles only IPv4 and does not handle IPv6, the **ai_family** member should be set to **AF_INET**. For another example, if the caller handles only TCP and does not handle UDP, the **ai_socktype** member should be set to **SOCK_STREAM**.

If the *pHints* parameter is a **NULL** pointer, the **GetAddrInfoEx** function treats it as if the addrinfoex structure in *pHints* were initialized with its **ai_family** member set to **AF_UNSPEC** and all other members set to **NULL** or zero.

When **GetAddrInfoEx** is called from a service, if the operation is the result of a user process calling the service, the service should impersonate the user. This is to allow security to be properly enforced.

The **GetAddrInfoEx** function can be used to convert a text string representation of an IP address to an addrinfoex structure that contains a sockaddr structure for the IP address and other information. To be used in this way, the string pointed to by the *pName* parameter must contain a text representation of an IP address and the **addrinfoex** structure pointed to by the *pHints* parameter must have the AI_NUMERICHOST flag set in the

ai_flags member. The string pointed to by the *pName* parameter may contain a text representation of either an IPv4 or an IPv6 address. The text IP address is converted to an **addrinfoex** structure pointed to by the *ppResult* parameter. The returned **addrinfoex** structure contains a **sockaddr** structure for the IP address along with additional information about the IP address.

Multiple namespace providers may be installed on a local computer for the same namespace. For example, the base Windows TCP/IP networking software registers for the NS_DNS namespace. The Microsoft Forefront Threat Management Gateway (TMG) and the older Microsoft Internet Security and Acceleration (ISA) Server include Firewall Client software that also registers for the NS_DNS namespace. When the *dwNameSpace* parameter is set to a value (NS_DNS, for example) and the *lpNspId* parameter is **NULL**, the results returned by the **GetAddrInfoEx** function are the merged results from all namespace providers that register for the specified namespace with duplicate results eliminated. The *lpNspId* parameter should be set to the GUID of the specific namespace provider if only a single namespace provider is to be queried.

If the *pNameSpace* parameter is set to NS_ALL, then the results from querying all namespace providers is merged and returned. In this case, duplicate responses may be returned in the results pointed to by the *ppResult* parameter if multiple namespace providers return the same information.

On Windows 8 and Windows Server 2012, if the **GetAddrInfoEx** function will complete asynchronously, the pointer returned in the *lpNameHandle* parameter may be used with the **GetAddrInfoExCancel** function. The handle returned is valid when **GetAddrInfoEx** returns until the completion routine is called, the event is triggered, or GetAddrInfoExCancel function is called with this handle.

Freeing Address Information from Dynamic Allocation

All information returned by the **GetAddrInfoEx** function pointed to by the *ppResult* parameter is dynamically allocated, including all addrinfoex structures, socket address structures, and canonical host name strings pointed to by **addrinfoex** structures. Memory allocated by a successful call to this function must be released with a subsequent call to FreeAddrInfoEx.

Example Code

The following example demonstrates the use of the **GetAddrInfoEx** function.

```
#ifndef UNICODE
#define UNICODE
#endif

#include <winsock2.h>
#include <ws2tcpip.h>
#include <objbase.h>
#include <stdio.h>

// Need to link with Ws2_32.lib
#pragma comment(lib, "ws2_32.lib")

// Need to link with Ole32.lib to print GUID
#pragma comment(lib, "ole32.lib")

int __cdecl wmain(int argc, wchar_t ** argv)
{

    //-----------------------------------------
    // Declare and initialize variables
    WSADATA wsaData;
    int iResult;

    DWORD dwRetval;

    int i = 1;

    DWORD dwNamespace = NS_ALL;
    LPGUID lpNspid = NULL;
```

```
    ADDRINFOEX *result = NULL;
    ADDRINFOEX *ptr = NULL;
    ADDRINFOEX hints;

    // LPSOCKADDR sockaddr_ip;
    struct sockaddr_in *sockaddr_ipv4;
    struct sockaddr_in6 *sockaddr_ipv6;

    // DWORD ipbufferlength = 46;
    wchar_t ipstringbuffer[46];

    // variables needed to print namespace provider GUID
    int iRet = 0;
    WCHAR GuidString[40] = { 0 };

    // Validate the parameters
    if (argc != 4) {
        wprintf(L"usage: %ws <hostname> <servicename> <namespace>\n", argv[0]);
        wprintf(L"getaddrinfoex provides protocol-independent translation\n");
        wprintf(L"   from a host name to an IP address\n");
        wprintf(L"%ws example usage\n", argv[0]);
        wprintf(L"   %ws www.contoso.com 0 12\n", argv[0]);
        wprintf(L"   looks up the www.contoso.com in the NS_DNS namespace\n",
                argv[0]);
        return 1;
    }
    // Initialize Winsock
    iResult = WSAStartup(MAKEWORD(2, 2), &wsaData);
    if (iResult != 0) {
        wprintf(L"WSAStartup failed: %d\n", iResult);
        return 1;
    }
    //--------------------------------
    // Setup the hints address info structure
    // which is passed to the getaddrinfo() function
    ZeroMemory(&hints, sizeof (hints));
    hints.ai_family = AF_UNSPEC;
    hints.ai_socktype = SOCK_STREAM;
    hints.ai_protocol = IPPROTO_TCP;

    dwNamespace = (DWORD) _wtoi(argv[3]);

    wprintf(L"Calling GetAddrInfoEx with following parameters:\n");
    wprintf(L"\tName = %ws\n", argv[1]);
    wprintf(L"\tServiceName (or port) = %ws\n", argv[2]);
    wprintf(L"\tNamespace = %s (", argv[3]);
    switch (dwNamespace) {
    case NS_ALL:
        wprintf(L"(NS_ALL)\n");
        break;
    case NS_DNS:
        wprintf(L"(NS_DNS)\n");
        break;
    case NS_NETBT:
        wprintf(L"NS_NETBT");
        break;
    case NS_WINS:
        wprintf(L"NS_WINS");
        break;
    case NS_NLA:
        wprintf(L"NS_NLA");
        break;
    case NS_BTH:
        wprintf(L"NS_BTH");
        break;
    case NS_NTDS:
        wprintf(L"NS_NTDS");
        break;
```

```
        break;
    case NS_EMAIL:
        wprintf(L"NS_EMAIL");
        break;
    case NS_PNRPNAME:
        wprintf(L"NS_PNRPNAME");
        break;
    case NS_PNRPCLOUD:
        wprintf(L"NS_PNRPCLOUD");
        break;
    default:
        wprintf(L"Other");
        break;
    }
    wprintf(L")\n\n");

//--------------------------------
// Call getaddrinfoex(). If the call succeeds,
// the result variable will hold a linked list
// of addrinfo structures containing response
// information
    dwRetval =
        GetAddrInfoEx(argv[1], argv[2], dwNamespace, lpNspid, &hints, &result,
                      NULL, NULL, NULL, NULL);
    if (dwRetval != 0) {
        wprintf(L"GetAddrInfoEx failed with error: %d\n", dwRetval);
        WSACleanup();
        return 1;
    }

    wprintf(L"GetAddrInfoEx returned success\n");

    // Retrieve each address and print out the hex bytes
    for (ptr = result; ptr != NULL; ptr = ptr->ai_next) {

        wprintf(L"GetAddrInfoEx response %d\n", i++);
        wprintf(L"\tFlags: 0x%x\n", ptr->ai_flags);
        wprintf(L"\tFamily: ");
        switch (ptr->ai_family) {
        case AF_UNSPEC:
            wprintf(L"Unspecified\n");
            break;
        case AF_INET:
            wprintf(L"AF_INET (IPv4)\n");
            // the InetNtop function is available on Windows Vista and later
            sockaddr_ipv4 = (struct sockaddr_in *) ptr->ai_addr;
            wprintf(L"\tIPv4 address %ws\n",
                    InetNtop(AF_INET, &sockaddr_ipv4->sin_addr, ipstringbuffer,
                             46));

            // We could also use the WSAAddressToString function
            // sockaddr_ip = (LPSOCKADDR) ptr->ai_addr;
            // The buffer length is changed by each call to WSAAddresstoString
            // So we need to set it for each iteration through the loop for safety
            // ipbufferlength = 46;
            // iRetval = WSAAddressToString(sockaddr_ip, (DWORD) ptr->ai_addrlen, NULL,
            //    ipstringbuffer, &ipbufferlength );
            // if (iRetval)
            //    wprintf(L"WSAAddressToString failed with %u\n", WSAGetLastError() );
            // else
            //    wprintf(L"\tIPv4 address %ws\n", ipstringbuffer);
            break;
        case AF_INET6:
            wprintf(L"AF_INET6 (IPv6)\n");
            // the InetNtop function is available on Windows Vista and later
            sockaddr_ipv6 = (struct sockaddr_in6 *) ptr->ai_addr;
            wprintf(L"\tIPv6 address %ws\n",
                    InetNtop(AF_INET6, &sockaddr_ipv6->sin6_addr,
                             ipstringbuffer, 46));
```

```
            // We could also use WSAAddressToString which also returns the scope ID
            // sockaddr_ip = (LPSOCKADDR) ptr->ai_addr;
            // The buffer length is changed by each call to WSAAddresstoString
            // So we need to set it for each iteration through the loop for safety
            // ipbufferlength = 46;
            //iRetval = WSAAddressToString(sockaddr_ip, (DWORD) ptr->ai_addrlen, NULL,
            //    ipstringbuffer, &ipbufferlength );
            //if (iRetval)
            //    wprintf(L"WSAAddressToString failed with %u\n", WSAGetLastError() );
            //else
            //    wprintf(L"\tIPv6 address %ws\n", ipstringbuffer);
            break;
        default:
            wprintf(L"Other %ld\n", ptr->ai_family);
            break;
        }
        wprintf(L"\tSocket type: ");
        switch (ptr->ai_socktype) {
        case 0:
            wprintf(L"Unspecified\n");
            break;
        case SOCK_STREAM:
            wprintf(L"SOCK_STREAM (stream)\n");
            break;
        case SOCK_DGRAM:
            wprintf(L"SOCK_DGRAM (datagram) \n");
            break;
        case SOCK_RAW:
            wprintf(L"SOCK_RAW (raw) \n");
            break;
        case SOCK_RDM:
            wprintf(L"SOCK_RDM (reliable message datagram)\n");
            break;
        case SOCK_SEQPACKET:
            wprintf(L"SOCK_SEQPACKET (pseudo-stream packet)\n");
            break;
        default:
            wprintf(L"Other %ld\n", ptr->ai_socktype);
            break;
        }
        wprintf(L"\tProtocol: ");
        switch (ptr->ai_protocol) {
        case 0:
            wprintf(L"Unspecified\n");
            break;
        case IPPROTO_TCP:
            wprintf(L"IPPROTO_TCP (TCP)\n");
            break;
        case IPPROTO_UDP:
            wprintf(L"IPPROTO_UDP (UDP) \n");
            break;
        default:
            wprintf(L"Other %ld\n", ptr->ai_protocol);
            break;
        }
        wprintf(L"\tLength of this sockaddr: %d\n", ptr->ai_addrlen);
        wprintf(L"\tCanonical name: %s\n", ptr->ai_canonname);

        if (ptr->ai_blob == NULL)
            wprintf(L"\tBlob: (null)\n");
        else
            wprintf(L"\tLength of the blob: %u\n",
                    (DWORD) ptr->ai_bloblen);

        if (ptr->ai_provider == NULL)
            wprintf(L"\tNamespace provider GUID: (null)\n");
        else {
            iRet =
```

```
                StringFromGUID2(*(ptr->ai_provider), (LPOLESTR) & GuidString,
                                39);
            // For c rather than C++ source code, the above line needs to be
            // iRet = StringFromGUID2(&ptr.ai_provider, (LPOLESTR) &GuidString, 39);
            if (iRet == 0)
                wprintf(L"StringFromGUID2 failed\n");
            else {
                wprintf(L"\tNamespace provider: %ws\n", GuidString);
            }
        }
    }

    FreeAddrInfoEx(result);
    WSACleanup();

    return 0;
}
```

The following example demonstrates how to use the **GetAddrInfoEx** function asynchronous to resolve a name to an IP address..

```
//
//    This sample demonstrates how to use asynchronous GetAddrInfoEx to
//    resolve a name to an IP address.
//
//    ResolveName <QueryName>
//

#ifndef UNICODE
#define UNICODE
#endif

#include <winsock2.h>
#include <ws2tcpip.h>
#include <stdio.h>
#include <stdlib.h>

// Need to link with Ws2_32.lib
#pragma comment(lib, "ws2_32.lib")

#define MAX_ADDRESS_STRING_LENGTH   64

//
//  Asynchronous query context structure.
//

typedef struct _QueryContext
{
    OVERLAPPED      QueryOverlapped;
    PADDRINFOEX     QueryResults;
    HANDLE          CompleteEvent;
}QUERY_CONTEXT, *PQUERY_CONTEXT;

VOID
WINAPI
QueryCompleteCallback(
    _In_ DWORD Error,
    _In_ DWORD Bytes,
    _In_ LPOVERLAPPED Overlapped
    );

int
__cdecl
wmain(
    _In_ int Argc, PWCHAR Argv[]
```

```
    )
{
    INT                 Error = ERROR_SUCCESS;
    WSADATA             wsaData;
    BOOL                IsWSAStartupCalled = FALSE;
    ADDRINFOEX          Hints;
    QUERY_CONTEXT       QueryContext;
    HANDLE              CancelHandle = NULL;
    DWORD               QueryTimeout = 5 * 1000; // 5 seconds

    ZeroMemory(&QueryContext, sizeof(QueryContext));

    //
    //  Validate the parameters
    //

    if (Argc != 2)
    {
        wprintf(L"Usage: ResolveName <QueryName>\n");
        goto exit;
    }

    //
    //  All Winsock functions require WSAStartup() to be called first
    //

    Error = WSAStartup(MAKEWORD(2, 2), &wsaData);
    if (Error != 0)
    {
        wprintf(L"WSAStartup failed with %d\n", Error);
        goto exit;
    }

    IsWSAStartupCalled = TRUE;

    ZeroMemory(&Hints, sizeof(Hints));
    Hints.ai_family = AF_UNSPEC;

    //
    //  Note that this is a simple sample that waits/cancels a single
    //  asynchronous query. The reader may extend this to support
    //  multiple asynchronous queries.
    //

    QueryContext.CompleteEvent = CreateEvent(NULL, TRUE, FALSE, NULL);

    if (QueryContext.CompleteEvent == NULL)
    {
        Error = GetLastError();
        wprintf(L"Failed to create completion event: Error %d\n",  Error);
        goto exit;
    }

    //
    //  Initiate asynchronous GetAddrInfoExW.
    //
    //  Note GetAddrInfoEx can also be invoked asynchronously using an event
    //  in the overlapped object (Just set hEvent in the Overlapped object
    //  and set NULL as completion callback.)
    //
    //  This sample uses the completion callback method.
    //

    Error = GetAddrInfoExW(Argv[1],
                           NULL,
                           NS_DNS,
                           NULL,
                           &Hints,
                           &QueryContext.QueryResults,
```

```
                            NULL,
                            &QueryContext.QueryOverlapped,
                            QueryCompleteCallback,
                            &CancelHandle);

    //
    //  If GetAddrInfoExW() returns  WSA_IO_PENDING, GetAddrInfoExW will invoke
    //  the completion routine. If GetAddrInfoExW returned anything else we must
    //  invoke the completion directly.
    //

    if (Error != WSA_IO_PENDING)
    {
        QueryCompleteCallback(Error, 0, &QueryContext.QueryOverlapped);
        goto exit;
    }

    //
    //  Wait for query completion for 5 seconds and cancel the query if it has
    //  not yet completed.
    //

    if (WaitForSingleObject(QueryContext.CompleteEvent,
                            QueryTimeout)  == WAIT_TIMEOUT )
    {

        //
        //  Cancel the query: Note that the GetAddrInfoExCancelcancel call does
        //  not block, so we must wait for the completion routine to be invoked.
        //  If we fail to wait, WSACleanup() could be called while an
        //  asynchronous query is still in progress, possibly causing a crash.
        //

        wprintf(L"The query took longer than %d seconds to complete; "
                L"cancelling the query...\n", QueryTimeout/1000);

        GetAddrInfoExCancel(&CancelHandle);

        WaitForSingleObject(QueryContext.CompleteEvent,
                            INFINITE);
    }

exit:

    if (IsWSAStartupCalled)
    {
        WSACleanup();
    }

    if (QueryContext.CompleteEvent)
    {
        CloseHandle(QueryContext.CompleteEvent);
    }

    return Error;
}

//
// Callback function called by Winsock as part of asynchronous query complete
//

VOID
WINAPI
QueryCompleteCallback(
    _In_ DWORD Error,
    _In_ DWORD Bytes,
    _In_ LPOVERLAPPED Overlapped
    )
{
```

```
{
    PQUERY_CONTEXT  QueryContext = NULL;
    PADDRINFOEX     QueryResults = NULL;
    WCHAR           AddrString[MAX_ADDRESS_STRING_LENGTH];
    DWORD           AddressStringLength;

    UNREFERENCED_PARAMETER(Bytes);

    QueryContext = CONTAINING_RECORD(Overlapped,
                                     QUERY_CONTEXT,
                                     QueryOverlapped);

    if (Error != ERROR_SUCCESS)
    {
        wprintf(L"ResolveName failed with %d\n", Error);
        goto exit;
    }

    wprintf(L"ResolveName succeeded. Query Results:\n");

    QueryResults = QueryContext->QueryResults;

    while(QueryResults)
    {
        AddressStringLength = MAX_ADDRESS_STRING_LENGTH;

        WSAAddressToString(QueryResults->ai_addr,
                           (DWORD)QueryResults->ai_addrlen,
                           NULL,
                           AddrString,
                           &AddressStringLength);

        wprintf(L"Ip Address: %s\n", AddrString);
        QueryResults = QueryResults->ai_next;
    }

exit:

    if (QueryContext->QueryResults)
    {
        FreeAddrInfoEx(QueryContext->QueryResults);
    }

    //
    //  Notify caller that the query completed
    //

    SetEvent(QueryContext->CompleteEvent);
    return;
}
```

> **Note** Ensure that the development environment targets the newest version of *Ws2tcpip.h* which includes structure and function definitions for addrinfoex and **GetAddrInfoEx**, respectively.

Internationalized Domain Names

Internet host names typically consist of a very restricted set of characters:

- Upper and lower case ASCII letters from the English alphabet.
- Digits from 0 to 9.
- ASCII hyphen characters.

With the growth of the Internet, there is a growing need to identify Internet host names for other languages not

represented by the ASCII character set. Identifiers which facilitate this need and allow non-ASCII characters (Unicode) to be represented as special ASCII character strings are known as Internationalized Domain Names (IDNs). A mechanism called Internationalizing Domain Names in Applications (IDNA) is used to handle IDNs in a standard fashion. The specifications for IDNs and IDNA are documented in RFC 3490, RTF 5890, and RFC 6365 published by the Internet Engineering Task Force (IETF).

On Windows 8 and Windows Server 2012, the **GetAddrInfoEx** function provides support for Internationalized Domain Name (IDN) parsing applied to the name passed in the *pName* parameter. Winsock performs Punycode/IDN encoding and conversion. This behavior can be disabled using the **AI_DISABLE_IDN_ENCODING** flag discussed below.

On Windows 7 and Windows Server 2008 R2 or earlier, the **GetAddrInfoEx** function does not currently provide support for IDN parsing applied to the name passed in the *pName* parameter. The wide character version of the **GetAddrInfoEx** function does not use Punycode to convert an IDN Punycode format as per RFC 3490. The wide character version of the **GetAddrInfoEx** function when querying DNS encodes the Unicode name in UTF-8 format, the format used by Microsoft DNS servers in an enterprise environment.

Several functions on Windows Vista and later support conversion between Unicode labels in an IDN to their ASCII equivalents. The resulting representation of each Unicode label contains only ASCII characters and starts with the xn-- prefix if the Unicode label contained any non-ASCII characters. The reason for this is to support existing DNS servers on the Internet, since some DNS tools and servers only support ASCII characters (see RFC 3490).

The IdnToAscii function uses Punycode to convert an IDN to the ASCII representation of the original Unicode string using the standard algorithm defined in RFC 3490. The IdnToUnicode function converts the ASCII form of an IDN to the normal Unicode UTF-16 encoding syntax. For more information and links to related draft standards, see Handling Internationalized Domain Names (IDNs).

The IdnToAscii function can be used to convert an IDN name to an ASCII form that then can be passed in the *pName* parameter to the **GetAddrInfoEx** function when the ASCII version of this function is used (when UNICODE and _UNICODE are not defined). To pass this IDN name to the **GetAddrInfoEx** function when the wide character version of this function is used (when UNICODE or _UNICODE is defined), you can use the MultiByteToWideChar function to convert the **CHAR** string into a **WCHAR** string.

Use of ai_flags in the hints parameter

Flags in the **ai_flags** member of the optional addrinfoex structure provided in the *hints* parameter modify the behavior of the function.

These flag bits are defined in the *Ws2def.h* header file on the Microsoft Windows Software Development Kit (SDK) for Windows 7. These flag bits are defined in the *Ws2tcpip.h* header file on the Windows SDK for Windows Server 2008 and Windows Vista. These flag bits are defined in the *Ws2tcpip.h* header file on the Platform Software Development Kit (SDK) for Windows Server 2003, and Windows XP.

The flag bits can be a combination of the following:

FLAG BITS	DESCRIPTION

AI_PASSIVE	Setting the **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 *pName* is a **NULL** pointer, the IP address portion of the socket address structure is set to **INADDR_ANY** for IPv4 addresses and **IN6ADDR_ANY_INIT** for IPv6 addresses. When the **AI_PASSIVE** flag is not set, the returned socket address structure is ready for a call to the connect function for a connection-oriented protocol, or ready for a call to either the **connect**, sendto, or send functions for a connectionless protocol. If the *pName* parameter is a **NULL** pointer in this case, the IP address portion of the socket address structure is set to the loopback address.
AI_CANONNAME	If neither **AI_CANONNAME** nor **AI_NUMERICHOST** is used, the **GetAddrInfoEx** function attempts resolution. If a literal string is passed **GetAddrInfoEx** attempts to convert the string, and if a host name is passed the **GetAddrInfoEx** function attempts to resolve the name to an address or multiple addresses. When the **AI_CANONNAME** bit is set, the *pName* parameter cannot be **NULL**. Otherwise the **GetAddrInfoEx** function will fail with WSANO_RECOVERY. When the **AI_CANONNAME** bit is set and the **GetAddrInfoEx** function returns success, the **ai_canonname** member in the *ppResult* parameter points to a **NULL**-terminated string that contains the canonical name of the specified node. **Note** The **GetAddrInfoEx** function can return success when the **AI_CANONNAME** flag is set, yet the **ai_canonname** member in the associated addrinfo structure is **NULL**. Therefore, the recommended use of the **AI_CANONNAME** flag includes testing whether the **ai_canonname** member in the associated addrinfoex structure is **NULL**.
AI_NUMERICHOST	When the **AI_NUMERICHOST** bit is set, the *pName* parameter must contain a non-**NULL** numeric host address string, otherwise the **EAI_NONAME** error is returned. This flag prevents a name resolution service from being called.
AI_NUMERICSERV	When the **AI_NUMERICSERV** bit is set, the *pServiceName* parameter must contain a non-**NULL** numeric port number, otherwise the **EAI_NONAME** error is returned. This flag prevents a name resolution service from being called. The **AI_NUMERICSERV** flag is defined on Windows SDK for Windows Vista and later. The **AI_NUMERICSERV** flag is not supported by Microsoft providers.

AI_ALL	If the **AI_ALL** bit is set, a request is made for IPv6 addresses and IPv4 addresses with **AI_V4MAPPED**. The **AI_ALL** flag is defined on the Windows SDK for Windows Vista and later. The **AI_ALL** flag is supported on Windows Vista and later.
AI_ADDRCONFIG	If the **AI_ADDRCONFIG** bit is set, **GetAddrInfoEx** will resolve only if a global address is configured. If **AI_ADDRCONFIG** flag is specified, IPv4 addresses shall be returned only if an IPv4 address is configured on the local system, and IPv6 addresses shall be returned only if an IPv6 address is configured on the local system. The IPv4 or IPv6 loopback address is not considered a valid global address. The **AI_ADDRCONFIG** flag is defined on the Windows SDK for Windows Vista and later. The **AI_ADDRCONFIG** flag is supported on Windows Vista and later.
AI_V4MAPPED	If the **AI_V4MAPPED** bit is set and a request for IPv6 addresses fails, a name service request is made for IPv4 addresses and these addresses are converted to IPv4-mapped IPv6 address format. The **AI_V4MAPPED** flag is defined on the Windows SDK for Windows Vista and later. The **AI_V4MAPPED** flag is supported on Windows Vista and later.
AI_NON_AUTHORITATIVE	If the **AI_NON_AUTHORITATIVE** bit is set, the **NS_EMAIL** namespace provider returns both authoritative and non-authoritative results. If the **AI_NON_AUTHORITATIVE** bit is not set, the **NS_EMAIL** namespace provider returns only authoritative results. The **AI_NON_AUTHORITATIVE** flag is defined on the Windows SDK for Windows Vista and later. The **AI_NON_AUTHORITATIVE** flag is supported on Windows Vista and later and applies only to the **NS_EMAIL** namespace.
AI_SECURE	If the **AI_SECURE** bit is set, the **NS_EMAIL** namespace provider will return results that were obtained with enhanced security to minimize possible spoofing. The **AI_SECURE** flag is defined on the Windows SDK for Windows Vista and later. The **AI_SECURE** flag is supported on Windows Vista and later and applies only to the **NS_EMAIL** namespace.
AI_RETURN_PREFERRED_NAMES	If the **AI_RETURN_PREFERRED_NAMES** is set, then no name should be provided in the *pName* parameter. The **NS_EMAIL** namespace provider will return preferred names for publication. The **AI_RETURN_PREFERRED_NAMES** flag is defined on the Windows SDK for Windows Vista and later. The **AI_RETURN_PREFERRED_NAMES** flag is supported on Windows Vista and later and applies only to the **NS_EMAIL** namespace.

AI_FQDN	If the **AI_FQDN** is set and a flat name (single label) is specified, **GetAddrInfoEx** will return the fully qualified domain name that the name eventually resolved to. The fully qualified domain name is returned in the **ai_canonname** member in the associated addrinfoex structure. This is different than **AI_CANONNAME** bit flag that returns the canonical name registered in DNS which may be different than the fully qualified domain name that the flat name resolved to. When the **AI_FQDN** bit is set, the *pName* parameter cannot be **NULL**. Otherwise the **GetAddrInfoEx** function will fail with WSANO_RECOVERY. On Windows 8 and Windows Server 2012, both the **AI_FQDN** and **AI_CANONNAME** bits can be set. If the **GetAddrInfoEx** function is called with both the **AI_FQDN** and **AI_CANONNAME** bits, the *ppResult* parameter returns a pointer to an addrinfoex2 structure, not an addrinfoex structure. On Windows 7 and Windows Server 2008 R2, only one of the **AI_FQDN** and **AI_CANONNAME** bits can be set. The **GetAddrInfoEx** function will fail if both flags are present with **EAI_BADFLAGS**. **Windows 7:** The **AI_FQDN** flag is defined on the Windows SDK for Windows 7 and later. The **AI_FQDN** flag is supported on Windows 7 and later.
AI_FILESERVER	If the **AI_FILESERVER** is set, this is a hint to the namespace provider that the hostname being queried is being used in file share scenario. The namespace provider may ignore this hint. **Windows 7:** The **AI_FILESERVER** flag is defined on the Windows SDK for Windows 7 and later. The **AI_FILESERVER** flag is supported on Windows 7 and later.
AI_DISABLE_IDN_ENCODING	If the **AI_DISABLE_IDN_ENCODING** is set, this disables the automatic International Domain Name encoding using Punycode in the name resolution functions called by the **GetAddrInfoEx** function. **Windows 8:** The **AI_DISABLE_IDN_ENCODING** flag is defined on the Windows SDK for Windows 8 and later. The **AI_DISABLE_IDN_ENCODING** flag is supported on Windows 8 and later.

NOTE

The ws2tcpip.h header defines GetAddrInfoEx as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see Conventions for Function Prototypes.

Requirements

Minimum supported client	Windows Vista [desktop apps | UWP apps]
Minimum supported server	Windows Server 2008 [desktop apps | UWP apps]
Target Platform	Windows
Header	ws2tcpip.h
Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

FreeAddrInfoEx

GetAddrInfoExCancel

GetAddrInfoExOverlappedResult

GetAddrInfoW

IdnToAscii

IdnToUnicode

WSAEnumNameSpaceProviders

WSAGetLastError

Windows Sockets Error Codes

addrinfoex

addrinfoex2

gai_strerror

getaddrinfo

GetAddrInfoExCancel function (ws2tcpip.h)

10/22/2022 • 2 minutes to read • Edit Online

The **GetAddrInfoExCancel** function cancels an asynchronous operation by the GetAddrInfoEx function.

Syntax

```
INT WSAAPI GetAddrInfoExCancel(
  [in] LPHANDLE lpHandle
);
```

Parameters

`[in] lpHandle`

The handle of the asynchronous operation to cancel. This is the handle returned in the *lpNameHandle* parameter by the GetAddrInfoEx function.

Return value

On success, **GetAddrInfoExCancel** returns **NO_ERROR** (0). Failure returns a nonzero Windows Sockets error code, as found in the Windows Sockets Error Codes.

Remarks

The **GetAddrInfoExCancel** function cancels an asynchronous GetAddrInfoEx operation. The result is that the user's completion mechanism, either a callback or an event, is immediately invoked. No results are returned, and the error code returned for the **GetAddrInfoEx** asynchronous operation is set to **WSA_E_CANCELLED**. If the **GetAddrInfoEx** request has already completed or timed out, or the handle is invalid, and **WSA_INVALID_HANDLE** will be returned by **GetAddrInfoExCancel** function.

Since many of the underlying operations (legacy name service providers, for example) are synchronous, these operations will not actually be cancelled. These operations will continue running and consuming resources. Once the last outstanding name service provider request has completed, the resources will be released.

Windows 8.1 and **Windows Server 2012 R2**: This function is supported for Windows Store apps on Windows 8.1, Windows Server 2012 R2, and later.

Requirements

Minimum supported client	Windows 8.1, Windows 8 [desktop apps | UWP apps]
Minimum supported server	Windows Server 2012 [desktop apps | UWP apps]
Target Platform	Windows
Header	ws2tcpip.h

Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

GetAddrInfoEx

GetAddrInfoExOverlappedResult function (ws2tcpip.h)

10/22/2022 • 2 minutes to read • Edit Online

The **GetAddrInfoExOverlappedResult** function gets the return code for an **OVERLAPPED** structure used by an asynchronous operation for the GetAddrInfoEx function.

Syntax

```
INT WSAAPI GetAddrInfoExOverlappedResult(
  LPOVERLAPPED lpOverlapped
);
```

Parameters

`lpOverlapped`

A pointer to an **OVERLAPPED** structure for the asynchronous operation.

Return value

On success, the **GetAddrInfoExOverlappedResult** function returns **NO_ERROR** (0). When the underlying operation hasn't yet completed, the **GetAddrInfoExOverlappedResult** function returns **WSAEINPROGRESS**. On failure, the **GetAddrInfoExOverlappedResult** function returns **WSAEINVAL**.

Remarks

The **GetAddrInfoExOverlappedResult** function is used with the GetAddrInfoEx function for asynchronous operations.

If the **GetAddrInfoExOverlappedResult** function returns **WSAEINVAL**, the only way to distinguish whether **GetAddrInfoExOverlappedResult** function or the asynchronous operation returned the error is to check that the *lpOverlapped* parameter was not NULL. If the *lpOverlapped* parameter was NULL, then the **GetAddrInfoExOverlappedResult** function was passed a NULL pointer and failed.

Windows 8.1 and **Windows Server 2012 R2**: This function is supported for Windows Store apps on Windows 8.1, Windows Server 2012 R2, and later.

Requirements

Minimum supported client	Windows 8.1, Windows 8 [desktop apps | UWP apps]
Minimum supported server	Windows Server 2012 [desktop apps | UWP apps]
Target Platform	Windows

Header	ws2tcpip.h
Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

GetAddrInfoEx

GetAddrInfoExW function (ws2tcpip.h)

10/22/2022 • 29 minutes to read • Edit Online

The **GetAddrInfoEx** function provides protocol-independent name resolution with additional parameters to qualify which namespace providers should handle the request.

Syntax

```
INT WSAAPI GetAddrInfoExW(
  [in, optional] PCWSTR                             pName,
  [in, optional] PCWSTR                             pServiceName,
  [in]           DWORD                              dwNameSpace,
  [in, optional] LPGUID                             lpNspId,
  [in, optional] const ADDRINFOEXW                  *hints,
  [out]          PADDRINFOEXW                       *ppResult,
  [in, optional] timeval                            *timeout,
  [in, optional] LPOVERLAPPED                       lpOverlapped,
  [in, optional] LPLOOKUPSERVICE_COMPLETION_ROUTINE lpCompletionRoutine,
                 LPHANDLE                           lpHandle
);
```

Parameters

`[in, optional] pName`

A pointer to a **NULL**-terminated string containing a host (node) name or a numeric host address string. For the Internet protocol, the numeric host address string is a dotted-decimal IPv4 address or an IPv6 hex address.

`[in, optional] pServiceName`

A pointer to an optional **NULL**-terminated string that contains either a service name or port number represented as a string.

A service name is a string alias for a port number. For example, "http" is an alias for port 80 defined by the Internet Engineering Task Force (IETF) as the default port used by web servers for the HTTP protocol. Possible values for the *pServiceName* parameter when a port number is not specified are listed in the following file:

`%WINDIR%\system32\drivers\etc\services`

`[in] dwNameSpace`

An optional namespace identifier that determines which namespace providers are queried. Passing a specific namespace identifier will result in only namespace providers that support the specified namespace being queried. Specifying **NS_ALL** will result in all installed and active namespace providers being queried.

Options for the *dwNameSpace* parameter are listed in the *Winsock2.h* include file. Several namespace providers are added on Windows Vista and later. Other namespace providers can be installed, so the following possible values are only those commonly available. Many other values are possible.

VALUE	MEANING

NS_ALL 0	All installed and active namespaces.
NS_DNS 12	The domain name system (DNS) namespace.
NS_NETBT 13	The NetBIOS over TCP/IP (NETBT) namespace.
NS_WINS 14	The Windows Internet Naming Service (NS_WINS) namespace.
NS_NLA 15	The network location awareness (NLA) namespace. This namespace identifier is supported on Windows XP and later.
NS_BTH 16	The Bluetooth namespace. This namespace identifier is supported on Windows Vista and later.
NS_NTDS 32	The Windows NT Directory Services (NS_NTDS) namespace.
NS_EMAIL 37	The email namespace. This namespace identifier is supported on Windows Vista and later.
NS_PNRPNAME 38	The peer-to-peer namespace for a specific peer name. This namespace identifier is supported on Windows Vista and later.
NS_PNRPCLOUD 39	The peer-to-peer namespace for a collection of peer names. This namespace identifier is supported on Windows Vista and later.

`[in, optional] lpNspId`

A pointer to an optional GUID of a specific namespace provider to query in the case where multiple namespace providers are registered under a single namespace such as **NS_DNS**. Passing the GUID for specific namespace provider will result in only the specified namespace provider being queried. The WSAEnumNameSpaceProviders function can be called to retrieve the GUID for a namespace provider.

`[in, optional] hints`

A pointer to an addrinfoex structure that provides hints about the type of socket the caller supports.

The **ai_addrlen**, **ai_canonname**, **ai_addr**, and **ai_next** members of the addrinfoex structure pointed to by the *pHints* parameter must be zero or **NULL**. Otherwise the **GetAddrInfoEx** function will fail with

WSANO_RECOVERY.

See the Remarks for more details.

`[out] ppResult`

A pointer to a linked list of one or more addrinfoex structures that contains response information about the host.

`[in, optional] timeout`

An optional parameter indicating the time, in milliseconds, to wait for a response from the namespace provider before aborting the call.

This parameter is only supported when the **UNICODE** or **_UNICODE** macro has been defined in the sources before calling the **GetAddrInfoEx** function. Otherwise, this parameter is currently reserved and must be set to **NULL** since a *timeout* option is not supported.

`[in, optional] lpOverlapped`

An optional pointer to an overlapped structure used for asynchronous operation.

This parameter is only supported when the **UNICODE** or **_UNICODE** macro has been defined in the sources before calling the **GetAddrInfoEx** function.

On Windows 8 and Windows Server 2012, if no *lpCompletionRoutine* parameter is specified, the **hEvent** member of the **OVERLAPPED** structure must be set to a manual-reset event to be called upon completion of an asynchronous call. If a completion routine has been specified, the **hEvent** member must be NULL. When the event specified by **hEvent** has been set, the result of the operation can be retrieved by calling GetAddrInfoExOverlappedResult function.

On Windows 8 and Windows Server 2012 whenever the **UNICODE** or **_UNICODE** macro is not defined, this parameter is currently reserved and must be set to **NULL**.

On Windows 7 and Windows Server 2008 R2 or earlier, this parameter is currently reserved and must be set to **NULL** since asynchronous operations are not supported.

`[in, optional] lpCompletionRoutine`

Type: _In_opt_ **LPWSAOVERLAPPED_COMPLETION_ROUTINE**

An optional pointer to a function to be invoked upon successful completion for asynchronous operations.

This parameter is only supported when the **UNICODE** or **_UNICODE** macro has been defined in the sources before calling the **GetAddrInfoEx** function.

On Windows 8 and Windows Server 2012, if this parameter is specified, it must be a pointer to a function with the following signature:

```
typedef
void
(CALLBACK * LPLOOKUPSERVICE_COMPLETION_ROUTINE)(
    __in      DWORD    dwError,
    __in      DWORD    dwBytes,
    __in      LPWSAOVERLAPPED lpOverlapped
    );
```

When the asynchronous operation has completed, the completion routine will be invoked with *lpOverlapped* parameter set to the value of *lpOverlapped* parameter passed to **GetAddrInfoEx**. The **Pointer** member of the

OVERLAPPED structure will be set to the value of the *ppResult* parameter of the original call. If the **Pointer** member points to a non-NULL pointer to the addrinfoex structure, it is the caller's responsibility to call FreeAddrInfoEx to free the **addrinfoex** structure. The *dwError* parameter passed to the completion routine will be set to a Winsock error code. The *dwBytes* parameter is reserved for future use and must be ignored.

On Windows 8 and Windows Server 2012 whenever the **UNICODE** or **_UNICODE** macro is not defined, this parameter is currently reserved and must be set to **NULL**.

On Windows 7 and Windows Server 2008 R2 or earlier, this parameter is currently reserved and must be set to **NULL** since asynchronous operations are not supported.

`lpHandle`

TBD

Return value

On success, **GetAddrInfoEx** returns **NO_ERROR** (0). Failure returns a nonzero Windows Sockets error code, as found in the Windows Sockets Error Codes.

Most nonzero error codes returned by the **GetAddrInfoEx** function map to the set of errors outlined by Internet Engineering Task Force (IETF) recommendations. The following table shows these error codes and their WSA equivalents. It is recommended that the WSA error codes be used, as they offer familiar and comprehensive error information for Winsock programmers.

ERROR VALUE	WSA EQUIVALENT	DESCRIPTION
EAI_AGAIN	WSATRY_AGAIN	A temporary failure in name resolution occurred.
EAI_BADFLAGS	WSAEINVAL	An invalid parameter was provided. This error is returned if any of the reserved parameters are not **NULL**. This error is also returned if an invalid value was provided for the **ai_flags** member of the *pHints* parameter.
EAI_FAIL	WSANO_RECOVERY	A nonrecoverable failure in name resolution occurred.
EAI_FAMILY	WSAEAFNOSUPPORT	The **ai_family** member of the *pHints* parameter is not supported.
EAI_MEMORY	WSA_NOT_ENOUGH_MEMORY	A memory allocation failure occurred.
EAI_NONAME	WSAHOST_NOT_FOUND	The name does not resolve for the supplied parameters or the *pName* and *pServiceName* parameters were not provided.
EAI_SERVICE	WSATYPE_NOT_FOUND	The *pServiceName* parameter is not supported for the specified **ai_socktype** member of the *pHints* parameter.
EAI_SOCKTYPE	WSAESOCKTNOSUPPORT	The **ai_socktype** member of the *pHints* parameter is not supported.

Use the gai_strerror function to print error messages based on the EAI codes returned by the **GetAddrInfoEx** function. The **gai_strerror** function is provided for compliance with IETF recommendations, but it is not thread safe. Therefore, use of traditional Windows Sockets functions such as WSAGetLastError is recommended.

ERROR CODE	MEANING
WSA_NOT_ENOUGH_MEMORY	There was insufficient memory to perform the operation.
WSAEAFNOSUPPORT	An address incompatible with the requested protocol was used. This error is returned if the **ai_family** member of the addrinfoex structure pointed to by the *pHints* parameter is not supported.
WSAEINVAL	An invalid argument was supplied. This error is returned if an invalid value was provided for the **ai_flags** member of the addrinfoex structure pointed to by the *pHints* parameter. This error is also returned when the *dwNameSpace* parameter is NS_PNRPNAME or NS_PNRPCLOUD and the peer-to-peer name service is not operating.
WSAESOCKTNOSUPPORT	The support for the specified socket type does not exist in this address family. This error is returned if the **ai_socktype** member of the addrinfoex structure pointed to by the *pHints* parameter is not supported.
WSAHOST_NOT_FOUND	No such host is known. This error is returned if the name does not resolve for the supplied parameters or the *pName* and *pServiceName* parameters were not provided.
WSANO_DATA	The requested name is valid, but no data of the requested type was found.
WSANO_RECOVERY	A nonrecoverable error occurred during a database lookup. This error is returned if nonrecoverable error in name resolution occurred.
WSANOTINITIALISED	A successful WSAStartup call must occur before using this function.
WSASERVICE_NOT_FOUND	No such service is known. The service cannot be found in the specified name space. This error is returned if the *pName* or *pServiceName* parameter is not found for the namespace specified in the *dwNameSpace* parameter or the namespace specified in the *dwNameSpace* parameter is not installed.
WSATRY_AGAIN	This is usually a temporary error during hostname resolution and means that the local server did not receive a response from an authoritative server. This error is returned when a temporary failure in name resolution occurred.
WSATYPE_NOT_FOUND	The specified class was not found. The *pServiceName* parameter is not supported for the specified **ai_socktype** member of the addrinfoex structure pointed to by the *pHints* parameter.

Remarks

The **GetAddrInfoEx** function provides protocol-independent translation from host name to address and from service name to port number. The **GetAddrInfoEx** function is an enhanced version of the getaddrinfo and GetAddrInfoW functions. The **GetAddrInfoEx** function allows specifying the namespace provider to resolve the query.

The **GetAddrInfoEx** function aggregates and returns results from multiple namespace providers, unless a specific namespace provider is specified. For use with the IPv6 and IPv4 protocol, name resolution can be by the Domain Name System (DNS), a local *hosts* file, an email provider (the **NS_EMAIL** namespace), or by other naming mechanisms.

When UNICODE or _UNICODE is defined, **GetAddrInfoEx** is defined to **GetAddrInfoExW**, the Unicode version of this function. The string parameters are defined to the **PWSTR** data type and the **ADDRINFOEXW** structure is used. On Windows 8 and Windows Server 2012, the *timeout*, *lpOverlapped*, *lpCompletionRoutine*, and *lpNameHandle* parameters may be used to call the **GetAddrInfoEx** function so that it can complete asynchronously.

When UNICODE or _UNICODE is not defined, **GetAddrInfoEx** is defined to **GetAddrInfoExA**, the ANSI version of this function. The string parameters are of the **PCSTR** data type and the **ADDRINFOEXA** structure is used. The *timeout*, *lpOverlapped*, *lpCompletionRoutine*, and *lpNameHandle* parameters must be set to **NULL**.

One or both of the *pName* or *pServiceName* parameters must point to a **NULL**-terminated string. Generally both are provided.

Upon success, a linked list of addrinfoex structures is returned in the *ppResult* parameter. The list can be processed by following the pointer provided in the **ai_next** member of each returned **addrinfoex** structure until a **NULL** pointer is encountered. In each returned **addrinfoex** structure, the **ai_family**, **ai_socktype**, and **ai_protocol** members correspond to respective arguments in a socket or WSASocket function call. Also, the **ai_addr** member in each returned **addrinfoex** structure points to a filled-in socket address structure, the length of which is specified in its **ai_addrlen** member.

If the *pName* parameter points to a computer name, all permanent addresses for the computer that can be used as a source address are returned. On Windows Vista and later, these addresses would include all unicast IP addresses returned by the GetUnicastIpAddressTable or GetUnicastIpAddressEntry functions in which the **SkipAsSource** member is set to false in the MIB_UNICASTIPADDRESS_ROW structure.

If the *pName* parameter points to a string equal to "localhost", all loopback addresses on the local computer are returned.

If the *pName* parameter contains an empty string, all registered addresses on the local computer are returned.

On Windows Server 2003 and later if the *pName* parameter points to a string equal to "..localmachine", all registered addresses on the local computer are returned.

If the *pName* parameter refers to a cluster virtual server name, only virtual server addresses are returned. On Windows Vista and later, these addresses would include all unicast IP addresses returned by the GetUnicastIpAddressTable or GetUnicastIpAddressEntry functions in which the **SkipAsSource** member is set to true in the MIB_UNICASTIPADDRESS_ROW structure. See Windows Clustering for more information about clustering.

Windows 7 with Service Pack 1 (SP1) and Windows Server 2008 R2 with Service Pack 1 (SP1) add support to Netsh.exe for setting the SkipAsSource attribute on an IP address. This also changes the behavior such that if the **SkipAsSource** member in the MIB_UNICASTIPADDRESS_ROW structure is set to false, the IP address will be registered in DNS. If the **SkipAsSource** member is set to true, the IP address is not registered in DNS.

A hotfix is available for Windows 7 and Windows Server 2008 R2 that adds support to Netsh.exe for setting the

SkipAsSource attribute on an IP address. This hotfix also changes behavior such that if the **SkipAsSource** member in the MIB_UNICASTIPADDRESS_ROW structure is set to false, the IP address will be registered in DNS. If the **SkipAsSource** member is set to true, the IP address is not registered in DNS. For more information, see Knowledge Base (KB) 2386184.

A similar hotfix is also available for Windows Vista with Service Pack 2 (SP2) and Windows Server 2008 with Service Pack 2 (SP2) that adds support to Netsh.exe for setting the SkipAsSource attribute on an IP address. This hotfix also changes behavior such that if the **SkipAsSource** member in the MIB_UNICASTIPADDRESS_ROW structure is set to false, the IP address will be registered in DNS. If the **SkipAsSource** member is set to true, the IP address is not registered in DNS.

Callers of the **GetAddrInfoEx** function can provide hints about the type of socket supported through an addrinfoex structure pointed to by the *pHints* parameter. When the *pHints* parameter is used, the following rules apply to its associated **addrinfoex** structure:

- A value of **AF_UNSPEC** for **ai_family** indicates the caller will accept only the **AF_INET** and **AF_INET6** address families. Note that **AF_UNSPEC** and **PF_UNSPEC** are the same.
- A value of zero for **ai_socktype** indicates the caller will accept any socket type.
- A value of zero for **ai_protocol** indicates the caller will accept any protocol.
- The **ai_addrlen** member must be set to zero.
- The **ai_canonname** member must be set to **NULL**.
- The **ai_addr** member must be set to **NULL**.
- The **ai_next** member must be set to **NULL**.

Other values in the addrinfoex structure provided in the *pHints* parameter indicate specific requirements. For example, if the caller handles only IPv4 and does not handle IPv6, the **ai_family** member should be set to **AF_INET**. For another example, if the caller handles only TCP and does not handle UDP, the **ai_socktype** member should be set to **SOCK_STREAM**.

If the *pHints* parameter is a **NULL** pointer, the **GetAddrInfoEx** function treats it as if the addrinfoex structure in *pHints* were initialized with its **ai_family** member set to **AF_UNSPEC** and all other members set to **NULL** or zero.

When **GetAddrInfoEx** is called from a service, if the operation is the result of a user process calling the service, the service should impersonate the user. This is to allow security to be properly enforced.

The **GetAddrInfoEx** function can be used to convert a text string representation of an IP address to an addrinfoex structure that contains a sockaddr structure for the IP address and other information. To be used in this way, the string pointed to by the *pName* parameter must contain a text representation of an IP address and the **addrinfoex** structure pointed to by the *pHints* parameter must have the AI_NUMERICHOST flag set in the **ai_flags** member. The string pointed to by the *pName* parameter may contain a text representation of either an IPv4 or an IPv6 address. The text IP address is converted to an **addrinfoex** structure pointed to by the *ppResult* parameter. The returned **addrinfoex** structure contains a **sockaddr** structure for the IP address along with additional information about the IP address.

Multiple namespace providers may be installed on a local computer for the same namespace. For example, the base Windows TCP/IP networking software registers for the NS_DNS namespace. The Microsoft Forefront Threat Management Gateway (TMG) and the older Microsoft Internet Security and Acceleration (ISA) Server include Firewall Client software that also registers for the NS_DNS namespace. When the *dwNameSpace* parameter is set to a value (NS_DNS, for example) and the *lpNspId* parameter is **NULL**, the results returned by the **GetAddrInfoEx** function are the merged results from all namespace providers that register for the specified namespace with duplicate results eliminated. The *lpNspId* parameter should be set to the GUID of the specific namespace provider if only a single namespace provider is to be queried.

If the *pNameSpace* parameter is set to NS_ALL, then the results from querying all namespace providers is

merged and returned. In this case, duplicate responses may be returned in the results pointed to by the *ppResult* parameter if multiple namespace providers return the same information.

On Windows 8 and Windows Server 2012, if the **GetAddrInfoEx** function will complete asynchronously, the pointer returned in the *lpNameHandle* parameter may be used with the **GetAddrInfoExCancel** function. The handle returned is valid when **GetAddrInfoEx** returns until the completion routine is called, the event is triggered, or GetAddrInfoExCancel function is called with this handle.

Freeing Address Information from Dynamic Allocation

All information returned by the **GetAddrInfoEx** function pointed to by the *ppResult* parameter is dynamically allocated, including all addrinfoex structures, socket address structures, and canonical host name strings pointed to by **addrinfoex** structures. Memory allocated by a successful call to this function must be released with a subsequent call to FreeAddrInfoEx.

Example Code

The following example demonstrates the use of the **GetAddrInfoEx** function.

```
#ifndef UNICODE
#define UNICODE
#endif

#include <winsock2.h>
#include <ws2tcpip.h>
#include <objbase.h>
#include <stdio.h>

// Need to link with Ws2_32.lib
#pragma comment(lib, "ws2_32.lib")

// Need to link with Ole32.lib to print GUID
#pragma comment(lib, "ole32.lib")

int __cdecl wmain(int argc, wchar_t ** argv)
{

    //-----------------------------------------
    // Declare and initialize variables
    WSADATA wsaData;
    int iResult;

    DWORD dwRetval;

    int i = 1;

    DWORD dwNamespace = NS_ALL;
    LPGUID lpNspid = NULL;

    ADDRINFOEX *result = NULL;
    ADDRINFOEX *ptr = NULL;
    ADDRINFOEX hints;

    // LPSOCKADDR sockaddr_ip;
    struct sockaddr_in *sockaddr_ipv4;
    struct sockaddr_in6 *sockaddr_ipv6;

    // DWORD ipbufferlength = 46;
    wchar_t ipstringbuffer[46];

    // variables needed to print namespace provider GUID
    int iRet = 0;
    WCHAR GuidString[40] = { 0 };

    // Validate the parameters
    if (argc != 4) {
```

```
        wprintf(L"usage: %ws <hostname> <servicename> <namespace>\n", argv[0]);
        wprintf(L"getaddrinfoex provides protocol-independent translation\n");
        wprintf(L"   from a host name to an IP address\n");
        wprintf(L"%ws example usage\n", argv[0]);
        wprintf(L"   %ws www.contoso.com 0 12\n", argv[0]);
        wprintf(L"   looks up the www.contoso.com in the NS_DNS namespace\n",
                argv[0]);
        return 1;
    }
    // Initialize Winsock
    iResult = WSAStartup(MAKEWORD(2, 2), &wsaData);
    if (iResult != 0) {
        wprintf(L"WSAStartup failed: %d\n", iResult);
        return 1;
    }
    //--------------------------------
    // Setup the hints address info structure
    // which is passed to the getaddrinfo() function
    ZeroMemory(&hints, sizeof (hints));
    hints.ai_family = AF_UNSPEC;
    hints.ai_socktype = SOCK_STREAM;
    hints.ai_protocol = IPPROTO_TCP;

    dwNamespace = (DWORD) _wtoi(argv[3]);

    wprintf(L"Calling GetAddrInfoEx with following parameters:\n");
    wprintf(L"\tName = %ws\n", argv[1]);
    wprintf(L"\tServiceName (or port) = %ws\n", argv[2]);
    wprintf(L"\tNamespace = %s (", argv[3]);
    switch (dwNamespace) {
    case NS_ALL:
        wprintf(L"(NS_ALL)\n");
        break;
    case NS_DNS:
        wprintf(L"(NS_DNS)\n");
        break;
    case NS_NETBT:
        wprintf(L"NS_NETBT");
        break;
    case NS_WINS:
        wprintf(L"NS_WINS");
        break;
    case NS_NLA:
        wprintf(L"NS_NLA");
        break;
    case NS_BTH:
        wprintf(L"NS_BTH");
        break;
    case NS_NTDS:
        wprintf(L"NS_NTDS");
        break;
    case NS_EMAIL:
        wprintf(L"NS_EMAIL");
        break;
    case NS_PNRPNAME:
        wprintf(L"NS_PNRPNAME");
        break;
    case NS_PNRPCLOUD:
        wprintf(L"NS_PNRPCLOUD");
        break;
    default:
        wprintf(L"Other");
        break;
    }
    wprintf(L")\n\n");

//--------------------------------
// Call getaddrinfoex(). If the call succeeds,
// the result variable will hold a linked list
```

```
// of addrinfo structures containing response
// information
    dwRetval =
        GetAddrInfoEx(argv[1], argv[2], dwNamespace, lpNspid, &hints, &result,
                      NULL, NULL, NULL, NULL);
    if (dwRetval != 0) {
        wprintf(L"GetAddrInfoEx failed with error: %d\n", dwRetval);
        WSACleanup();
        return 1;
    }

    wprintf(L"GetAddrInfoEx returned success\n");

    // Retrieve each address and print out the hex bytes
    for (ptr = result; ptr != NULL; ptr = ptr->ai_next) {

        wprintf(L"GetAddrInfoEx response %d\n", i++);
        wprintf(L"\tFlags: 0x%x\n", ptr->ai_flags);
        wprintf(L"\tFamily: ");
        switch (ptr->ai_family) {
        case AF_UNSPEC:
            wprintf(L"Unspecified\n");
            break;
        case AF_INET:
            wprintf(L"AF_INET (IPv4)\n");
            // the InetNtop function is available on Windows Vista and later
            sockaddr_ipv4 = (struct sockaddr_in *) ptr->ai_addr;
            wprintf(L"\tIPv4 address %ws\n",
                    InetNtop(AF_INET, &sockaddr_ipv4->sin_addr, ipstringbuffer,
                             46));

            // We could also use the WSAAddressToString function
            // sockaddr_ip = (LPSOCKADDR) ptr->ai_addr;
            // The buffer length is changed by each call to WSAAddresstoString
            // So we need to set it for each iteration through the loop for safety
            // ipbufferlength = 46;
            // iRetval = WSAAddressToString(sockaddr_ip, (DWORD) ptr->ai_addrlen, NULL,
            //    ipstringbuffer, &ipbufferlength );
            // if (iRetval)
            //    wprintf(L"WSAAddressToString failed with %u\n", WSAGetLastError() );
            // else
            //    wprintf(L"\tIPv4 address %ws\n", ipstringbuffer);
            break;
        case AF_INET6:
            wprintf(L"AF_INET6 (IPv6)\n");
            // the InetNtop function is available on Windows Vista and later
            sockaddr_ipv6 = (struct sockaddr_in6 *) ptr->ai_addr;
            wprintf(L"\tIPv6 address %ws\n",
                    InetNtop(AF_INET6, &sockaddr_ipv6->sin6_addr,
                             ipstringbuffer, 46));

            // We could also use WSAAddressToString which also returns the scope ID
            // sockaddr_ip = (LPSOCKADDR) ptr->ai_addr;
            // The buffer length is changed by each call to WSAAddresstoString
            // So we need to set it for each iteration through the loop for safety
            // ipbufferlength = 46;
            //iRetval = WSAAddressToString(sockaddr_ip, (DWORD) ptr->ai_addrlen, NULL,
            //    ipstringbuffer, &ipbufferlength );
            //if (iRetval)
            //    wprintf(L"WSAAddressToString failed with %u\n", WSAGetLastError() );
            //else
            //    wprintf(L"\tIPv6 address %ws\n", ipstringbuffer);
            break;
        default:
            wprintf(L"Other %ld\n", ptr->ai_family);
            break;
        }
        wprintf(L"\tSocket type: ");
        switch (ptr->ai_socktype) {
```

```
        case 0:
            wprintf(L"Unspecified\n");
            break;
        case SOCK_STREAM:
            wprintf(L"SOCK_STREAM (stream)\n");
            break;
        case SOCK_DGRAM:
            wprintf(L"SOCK_DGRAM (datagram) \n");
            break;
        case SOCK_RAW:
            wprintf(L"SOCK_RAW (raw) \n");
            break;
        case SOCK_RDM:
            wprintf(L"SOCK_RDM (reliable message datagram)\n");
            break;
        case SOCK_SEQPACKET:
            wprintf(L"SOCK_SEQPACKET (pseudo-stream packet)\n");
            break;
        default:
            wprintf(L"Other %ld\n", ptr->ai_socktype);
            break;
        }
        wprintf(L"\tProtocol: ");
        switch (ptr->ai_protocol) {
        case 0:
            wprintf(L"Unspecified\n");
            break;
        case IPPROTO_TCP:
            wprintf(L"IPPROTO_TCP (TCP)\n");
            break;
        case IPPROTO_UDP:
            wprintf(L"IPPROTO_UDP (UDP) \n");
            break;
        default:
            wprintf(L"Other %ld\n", ptr->ai_protocol);
            break;
        }
        wprintf(L"\tLength of this sockaddr: %d\n", ptr->ai_addrlen);
        wprintf(L"\tCanonical name: %s\n", ptr->ai_canonname);

        if (ptr->ai_blob == NULL)
            wprintf(L"\tBlob: (null)\n");
        else
            wprintf(L"\tLength of the blob: %u\n",
                    (DWORD) ptr->ai_bloblen);

        if (ptr->ai_provider == NULL)
            wprintf(L"\tNamespace provider GUID: (null)\n");
        else {
            iRet =
                StringFromGUID2(*(ptr->ai_provider), (LPOLESTR) & GuidString,
                                39);
            // For c rather than C++ source code, the above line needs to be
            // iRet = StringFromGUID2(&ptr.ai_provider, (LPOLESTR) &GuidString, 39);
            if (iRet == 0)
                wprintf(L"StringFromGUID2 failed\n");
            else {
                wprintf(L"\tNamespace provider: %ws\n", GuidString);
            }
        }
    }

    FreeAddrInfoEx(result);
    WSACleanup();

    return 0;
}
```

The following example demonstrates how to use the **GetAddrInfoEx** function asynchronous to resolve a name to an IP address..

```
//
//    This sample demonstrates how to use asynchronous GetAddrInfoEx to
//    resolve a name to an IP address.
//
//    ResolveName <QueryName>
//

#ifndef UNICODE
#define UNICODE
#endif

#include <winsock2.h>
#include <ws2tcpip.h>
#include <stdio.h>
#include <stdlib.h>

// Need to link with Ws2_32.lib
#pragma comment(lib, "ws2_32.lib")

#define MAX_ADDRESS_STRING_LENGTH   64

//
//  Asynchronous query context structure.
//

typedef struct _QueryContext
{
    OVERLAPPED      QueryOverlapped;
    PADDRINFOEX     QueryResults;
    HANDLE          CompleteEvent;
}QUERY_CONTEXT, *PQUERY_CONTEXT;

VOID
WINAPI
QueryCompleteCallback(
    _In_ DWORD Error,
    _In_ DWORD Bytes,
    _In_ LPOVERLAPPED Overlapped
    );

int
__cdecl
wmain(
    _In_ int Argc, PWCHAR Argv[]
    )
{
    INT                 Error = ERROR_SUCCESS;
    WSADATA             wsaData;
    BOOL                IsWSAStartupCalled = FALSE;
    ADDRINFOEX          Hints;
    QUERY_CONTEXT       QueryContext;
    HANDLE              CancelHandle = NULL;
    DWORD               QueryTimeout = 5 * 1000; // 5 seconds

    ZeroMemory(&QueryContext, sizeof(QueryContext));

    //
    //  Validate the parameters
    //

    if (Argc != 2)
    {
        wprintf(L"Usage: ResolveName <QueryName>\n");
```

```
        wprintf(L"Usage: ResolveName <QueryName>\n");
        goto exit;
    }

    //
    //  All Winsock functions require WSAStartup() to be called first
    //

    Error = WSAStartup(MAKEWORD(2, 2), &wsaData);
    if (Error != 0)
    {
        wprintf(L"WSAStartup failed with %d\n", Error);
        goto exit;
    }

    IsWSAStartupCalled = TRUE;

    ZeroMemory(&Hints, sizeof(Hints));
    Hints.ai_family = AF_UNSPEC;

    //
    //  Note that this is a simple sample that waits/cancels a single
    //  asynchronous query. The reader may extend this to support
    //  multiple asynchronous queries.
    //

    QueryContext.CompleteEvent = CreateEvent(NULL, TRUE, FALSE, NULL);

    if (QueryContext.CompleteEvent == NULL)
    {
        Error = GetLastError();
        wprintf(L"Failed to create completion event: Error %d\n",  Error);
        goto exit;
    }

    //
    //  Initiate asynchronous GetAddrInfoExW.
    //
    //  Note GetAddrInfoEx can also be invoked asynchronously using an event
    //  in the overlapped object (Just set hEvent in the Overlapped object
    //  and set NULL as completion callback.)
    //
    //  This sample uses the completion callback method.
    //

    Error = GetAddrInfoExW(Argv[1],
                           NULL,
                           NS_DNS,
                           NULL,
                           &Hints,
                           &QueryContext.QueryResults,
                           NULL,
                           &QueryContext.QueryOverlapped,
                           QueryCompleteCallback,
                           &CancelHandle);

    //
    //  If GetAddrInfoExW() returns  WSA_IO_PENDING, GetAddrInfoExW will invoke
    //  the completion routine. If GetAddrInfoExW returned anything else we must
    //  invoke the completion directly.
    //

    if (Error != WSA_IO_PENDING)
    {
        QueryCompleteCallback(Error, 0, &QueryContext.QueryOverlapped);
        goto exit;
    }

    //
    //  Wait for query completion for 5 seconds and cancel the query if it has
```

```
    //  wait for query completion for 5 seconds and cancel the query if it has
    //  not yet completed.
    //

    if (WaitForSingleObject(QueryContext.CompleteEvent,
                            QueryTimeout)  == WAIT_TIMEOUT )
    {

        //
        //  Cancel the query: Note that the GetAddrInfoExCancelcancel call does
        //  not block, so we must wait for the completion routine to be invoked.
        //  If we fail to wait, WSACleanup() could be called while an
        //  asynchronous query is still in progress, possibly causing a crash.
        //

        wprintf(L"The query took longer than %d seconds to complete; "
                L"cancelling the query...\n", QueryTimeout/1000);

        GetAddrInfoExCancel(&CancelHandle);

        WaitForSingleObject(QueryContext.CompleteEvent,
                            INFINITE);
    }

exit:

    if (IsWSAStartupCalled)
    {
        WSACleanup();
    }

    if (QueryContext.CompleteEvent)
    {
        CloseHandle(QueryContext.CompleteEvent);
    }

    return Error;
}

//
// Callback function called by Winsock as part of asynchronous query complete
//

VOID
WINAPI
QueryCompleteCallback(
    _In_ DWORD Error,
    _In_ DWORD Bytes,
    _In_ LPOVERLAPPED Overlapped
    )
{
    PQUERY_CONTEXT  QueryContext = NULL;
    PADDRINFOEX     QueryResults = NULL;
    WCHAR           AddrString[MAX_ADDRESS_STRING_LENGTH];
    DWORD           AddressStringLength;

    UNREFERENCED_PARAMETER(Bytes);

    QueryContext = CONTAINING_RECORD(Overlapped,
                                     QUERY_CONTEXT,
                                     QueryOverlapped);

    if (Error != ERROR_SUCCESS)
    {
        wprintf(L"ResolveName failed with %d\n", Error);
        goto exit;
    }

    wprintf(L"ResolveName succeeded. Query Results:\n");
```

```
    QueryResults = QueryContext->QueryResults;

    while(QueryResults)
    {
        AddressStringLength = MAX_ADDRESS_STRING_LENGTH;

        WSAAddressToString(QueryResults->ai_addr,
                           (DWORD)QueryResults->ai_addrlen,
                           NULL,
                           AddrString,
                           &AddressStringLength);

        wprintf(L"Ip Address: %s\n", AddrString);
        QueryResults = QueryResults->ai_next;
    }

exit:

    if (QueryContext->QueryResults)
    {
        FreeAddrInfoEx(QueryContext->QueryResults);
    }

    //
    //  Notify caller that the query completed
    //

    SetEvent(QueryContext->CompleteEvent);
    return;
}
```

Note Ensure that the development environment targets the newest version of *Ws2tcpip.h* which includes structure and function definitions for addrinfoex and **GetAddrInfoEx**, respectively.

Internationalized Domain Names

Internet host names typically consist of a very restricted set of characters:

- Upper and lower case ASCII letters from the English alphabet.
- Digits from 0 to 9.
- ASCII hyphen characters.

With the growth of the Internet, there is a growing need to identify Internet host names for other languages not represented by the ASCII character set. Identifiers which facilitate this need and allow non-ASCII characters (Unicode) to be represented as special ASCII character strings are known as Internationalized Domain Names (IDNs). A mechanism called Internationalizing Domain Names in Applications (IDNA) is used to handle IDNs in a standard fashion. The specifications for IDNs and IDNA are documented in RFC 3490, RTF 5890, and RFC 6365 published by the Internet Engineering Task Force (IETF).

On Windows 8 and Windows Server 2012, the **GetAddrInfoEx** function provides support for Internationalized Domain Name (IDN) parsing applied to the name passed in the *pName* parameter. Winsock performs Punycode/IDN encoding and conversion. This behavior can be disabled using the **AI_DISABLE_IDN_ENCODING** flag discussed below.

On Windows 7 and Windows Server 2008 R2 or earlier, the **GetAddrInfoEx** function does not currently provide support for IDN parsing applied to the name passed in the *pName* parameter. The wide character version of the **GetAddrInfoEx** function does not use Punycode to convert an IDN Punycode format as per RFC 3490. The wide character version of the **GetAddrInfoEx** function when querying DNS encodes the Unicode name in UTF-8

format, the format used by Microsoft DNS servers in an enterprise environment.

Several functions on Windows Vista and later support conversion between Unicode labels in an IDN to their ASCII equivalents. The resulting representation of each Unicode label contains only ASCII characters and starts with the xn-- prefix if the Unicode label contained any non-ASCII characters. The reason for this is to support existing DNS servers on the Internet, since some DNS tools and servers only support ASCII characters (see RFC 3490).

The IdnToAscii function uses Punycode to convert an IDN to the ASCII representation of the original Unicode string using the standard algorithm defined in RFC 3490. The IdnToUnicode function converts the ASCII form of an IDN to the normal Unicode UTF-16 encoding syntax. For more information and links to related draft standards, see Handling Internationalized Domain Names (IDNs).

The IdnToAscii function can be used to convert an IDN name to an ASCII form that then can be passed in the *pName* parameter to the **GetAddrInfoEx** function when the ASCII version of this function is used (when UNICODE and _UNICODE are not defined). To pass this IDN name to the **GetAddrInfoEx** function when the wide character version of this function is used (when UNICODE or _UNICODE is defined), you can use the MultiByteToWideChar function to convert the **CHAR** string into a **WCHAR** string.

Use of ai_flags in the hints parameter

Flags in the **ai_flags** member of the optional addrinfoex structure provided in the *hints* parameter modify the behavior of the function.

These flag bits are defined in the *Ws2def.h* header file on the Microsoft Windows Software Development Kit (SDK) for Windows 7. These flag bits are defined in the *Ws2tcpip.h* header file on the Windows SDK for Windows Server 2008 and Windows Vista. These flag bits are defined in the *Ws2tcpip.h* header file on the Platform Software Development Kit (SDK) for Windows Server 2003, and Windows XP.

The flag bits can be a combination of the following:

FLAG BITS	DESCRIPTION
AI_PASSIVE	Setting the **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 *pName* is a **NULL** pointer, the IP address portion of the socket address structure is set to **INADDR_ANY** for IPv4 addresses and **IN6ADDR_ANY_INIT** for IPv6 addresses. When the **AI_PASSIVE** flag is not set, the returned socket address structure is ready for a call to the connect function for a connection-oriented protocol, or ready for a call to either the **connect**, sendto, or send functions for a connectionless protocol. If the *pName* parameter is a **NULL** pointer in this case, the IP address portion of the socket address structure is set to the loopback address.

AI_CANONNAME	If neither **AI_CANONNAME** nor **AI_NUMERICHOST** is used, the **GetAddrInfoEx** function attempts resolution. If a literal string is passed **GetAddrInfoEx** attempts to convert the string, and if a host name is passed the **GetAddrInfoEx** function attempts to resolve the name to an address or multiple addresses. When the **AI_CANONNAME** bit is set, the *pName* parameter cannot be **NULL**. Otherwise the **GetAddrInfoEx** function will fail with WSANO_RECOVERY. When the **AI_CANONNAME** bit is set and the **GetAddrInfoEx** function returns success, the **ai_canonname** member in the *ppResult* parameter points to a **NULL**-terminated string that contains the canonical name of the specified node. **Note** The **GetAddrInfoEx** function can return success when the **AI_CANONNAME** flag is set, yet the **ai_canonname** member in the associated addrinfo structure is **NULL**. Therefore, the recommended use of the **AI_CANONNAME** flag includes testing whether the **ai_canonname** member in the associated addrinfoex structure is **NULL**.
AI_NUMERICHOST	When the **AI_NUMERICHOST** bit is set, the *pName* parameter must contain a non-**NULL** numeric host address string, otherwise the **EAI_NONAME** error is returned. This flag prevents a name resolution service from being called.
AI_NUMERICSERV	When the **AI_NUMERICSERV** bit is set, the *pServiceName* parameter must contain a non-**NULL** numeric port number, otherwise the **EAI_NONAME** error is returned. This flag prevents a name resolution service from being called. The **AI_NUMERICSERV** flag is defined on Windows SDK for Windows Vista and later. The **AI_NUMERICSERV** flag is not supported by Microsoft providers.
AI_ALL	If the **AI_ALL** bit is set, a request is made for IPv6 addresses and IPv4 addresses with **AI_V4MAPPED**. The **AI_ALL** flag is defined on the Windows SDK for Windows Vista and later. The **AI_ALL** flag is supported on Windows Vista and later.
AI_ADDRCONFIG	If the **AI_ADDRCONFIG** bit is set, **GetAddrInfoEx** will resolve only if a global address is configured. If **AI_ADDRCONFIG** flag is specified, IPv4 addresses shall be returned only if an IPv4 address is configured on the local system, and IPv6 addresses shall be returned only if an IPv6 address is configured on the local system. The IPv4 or IPv6 loopback address is not considered a valid global address. The **AI_ADDRCONFIG** flag is defined on the Windows SDK for Windows Vista and later. The **AI_ADDRCONFIG** flag is supported on Windows Vista and later.

AI_V4MAPPED	If the **AI_V4MAPPED** bit is set and a request for IPv6 addresses fails, a name service request is made for IPv4 addresses and these addresses are converted to IPv4-mapped IPv6 address format. The **AI_V4MAPPED** flag is defined on the Windows SDK for Windows Vista and later. The **AI_V4MAPPED** flag is supported on Windows Vista and later.
AI_NON_AUTHORITATIVE	If the **AI_NON_AUTHORITATIVE** bit is set, the **NS_EMAIL** namespace provider returns both authoritative and non-authoritative results. If the **AI_NON_AUTHORITATIVE** bit is not set, the **NS_EMAIL** namespace provider returns only authoritative results. The **AI_NON_AUTHORITATIVE** flag is defined on the Windows SDK for Windows Vista and later. The **AI_NON_AUTHORITATIVE** flag is supported on Windows Vista and later and applies only to the **NS_EMAIL** namespace.
AI_SECURE	If the **AI_SECURE** bit is set, the **NS_EMAIL** namespace provider will return results that were obtained with enhanced security to minimize possible spoofing. The **AI_SECURE** flag is defined on the Windows SDK for Windows Vista and later. The **AI_SECURE** flag is supported on Windows Vista and later and applies only to the **NS_EMAIL** namespace.
AI_RETURN_PREFERRED_NAMES	If the **AI_RETURN_PREFERRED_NAMES** is set, then no name should be provided in the *pName* parameter. The **NS_EMAIL** namespace provider will return preferred names for publication. The **AI_RETURN_PREFERRED_NAMES** flag is defined on the Windows SDK for Windows Vista and later. The **AI_RETURN_PREFERRED_NAMES** flag is supported on Windows Vista and later and applies only to the **NS_EMAIL** namespace.

AI_FQDN	If the **AI_FQDN** is set and a flat name (single label) is specified, **GetAddrInfoEx** will return the fully qualified domain name that the name eventually resolved to. The fully qualified domain name is returned in the **ai_canonname** member in the associated addrinfoex structure. This is different than **AI_CANONNAME** bit flag that returns the canonical name registered in DNS which may be different than the fully qualified domain name that the flat name resolved to. When the **AI_FQDN** bit is set, the *pName* parameter cannot be **NULL**. Otherwise the **GetAddrInfoEx** function will fail with WSANO_RECOVERY. On Windows 8 and Windows Server 2012, both the **AI_FQDN** and **AI_CANONNAME** bits can be set. If the **GetAddrInfoEx** function is called with both the **AI_FQDN** and **AI_CANONNAME** bits, the *ppResult* parameter returns a pointer to an addrinfoex2 structure, not an addrinfoex structure. On Windows 7 and Windows Server 2008 R2, only one of the **AI_FQDN** and **AI_CANONNAME** bits can be set. The **GetAddrInfoEx** function will fail if both flags are present with **EAI_BADFLAGS**. **Windows 7:** The **AI_FQDN** flag is defined on the Windows SDK for Windows 7 and later. The **AI_FQDN** flag is supported on Windows 7 and later.
AI_FILESERVER	If the **AI_FILESERVER** is set, this is a hint to the namespace provider that the hostname being queried is being used in file share scenario. The namespace provider may ignore this hint. **Windows 7:** The **AI_FILESERVER** flag is defined on the Windows SDK for Windows 7 and later. The **AI_FILESERVER** flag is supported on Windows 7 and later.
AI_DISABLE_IDN_ENCODING	If the **AI_DISABLE_IDN_ENCODING** is set, this disables the automatic International Domain Name encoding using Punycode in the name resolution functions called by the **GetAddrInfoEx** function. **Windows 8:** The **AI_DISABLE_IDN_ENCODING** flag is defined on the Windows SDK for Windows 8 and later. The **AI_DISABLE_IDN_ENCODING** flag is supported on Windows 8 and later.

NOTE

The ws2tcpip.h header defines GetAddrInfoEx as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see Conventions for Function Prototypes.

Requirements

Minimum supported client	Windows Vista [desktop apps | UWP apps]
Minimum supported server	Windows Server 2008 [desktop apps | UWP apps]
Target Platform	Windows
Header	ws2tcpip.h
Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

FreeAddrInfoEx

GetAddrInfoExCancel

GetAddrInfoExOverlappedResult

GetAddrInfoW

IdnToAscii

IdnToUnicode

WSAEnumNameSpaceProviders

WSAGetLastError

Windows Sockets Error Codes

addrinfoex

addrinfoex2

gai_strerror

getaddrinfo

GetAddrInfoW function (ws2tcpip.h)

10/22/2022 • 20 minutes to read • Edit Online

The **GetAddrInfoW** function provides protocol-independent translation from a Unicode host name to an address.

Syntax

```
INT WSAAPI GetAddrInfoW(
  [in, optional] PCWSTR          pNodeName,
  [in, optional] PCWSTR          pServiceName,
  [in, optional] const ADDRINFOW *pHints,
  [out]          PADDRINFOW      *ppResult
);
```

Parameters

`[in, optional] pNodeName`

A pointer to a **NULL**-terminated Unicode string that contains a host (node) name or a numeric host address string. For the Internet protocol, the numeric host address string is a dotted-decimal IPv4 address or an IPv6 hex address.

`[in, optional] pServiceName`

A pointer to a **NULL**-terminated Unicode string that contains either a service name or port number represented as a string.

A service name is a string alias for a port number. For example, "http" is an alias for port 80 defined by the Internet Engineering Task Force (IETF) as the default port used by web servers for the HTTP protocol. Possible values for the *pServiceName* parameter when a port number is not specified are listed in the following file:

`%WINDIR%\system32\drivers\etc\services`

`[in, optional] pHints`

A pointer to an addrinfoW structure that provides hints about the type of socket the caller supports.

The **ai_addrlen**, **ai_canonname**, **ai_addr**, and **ai_next** members of the addrinfoW structure pointed to by the *pHints* parameter must be zero or **NULL**. Otherwise the GetAddrInfoEx function will fail with WSANO_RECOVERY.

See the Remarks for more details.

`[out] ppResult`

A pointer to a linked list of one or more addrinfoW structures that contains response information about the host.

Return value

Success returns zero. Failure returns a nonzero Windows Sockets error code, as found in the Windows Sockets Error Codes.

Most nonzero error codes returned by the **GetAddrInfoW** function map to the set of errors outlined by Internet Engineering Task Force (IETF) recommendations. The following table lists these error codes and their WSA equivalents. It is recommended that the WSA error codes be used, as they offer familiar and comprehensive error information for Winsock programmers.

ERROR VALUE	WSA EQUIVALENT	DESCRIPTION
EAI_AGAIN	WSATRY_AGAIN	A temporary failure in name resolution occurred.
EAI_BADFLAGS	WSAEINVAL	An invalid value was provided for the **ai_flags** member of the *pHints* parameter.
EAI_FAIL	WSANO_RECOVERY	A nonrecoverable failure in name resolution occurred.
EAI_FAMILY	WSAEAFNOSUPPORT	The **ai_family** member of the *pHints* parameter is not supported.
EAI_MEMORY	WSA_NOT_ENOUGH_MEMORY	A memory allocation failure occurred.
EAI_NONAME	WSAHOST_NOT_FOUND	The name does not resolve for the supplied parameters or the *pNodeName* and *pServiceName* parameters were not provided.
EAI_SERVICE	WSATYPE_NOT_FOUND	The *pServiceName* parameter is not supported for the specified **ai_socktype** member of the *pHints* parameter.
EAI_SOCKTYPE	WSAESOCKTNOSUPPORT	The **ai_socktype** member of the *pHints* parameter is not supported.

Use the gai_strerror function to print error messages based on the EAI_* codes returned by the **GetAddrInfoW** function. The **gai_strerror** function is provided for compliance with IETF recommendations, but it is not thread safe. Therefore, use of a traditional Windows Sockets function, such as WSAGetLastError, is recommended.

ERROR CODE	MEANING
WSA_NOT_ENOUGH_MEMORY	There was insufficient memory to perform the operation.
WSAEAFNOSUPPORT	An address incompatible with the requested protocol was used. This error is returned if the **ai_family** member of the addrinfoW structure pointed to by the *hints* parameter is not supported.
WSAEINVAL	An invalid argument was supplied. This error is returned if an invalid value was provided for the **ai_flags** member of the addrinfoW structure pointed to by the *hints* parameter.

WSAESOCKTNOSUPPORT	The support for the specified socket type does not exist in this address family. This error is returned if the **ai_socktype** member of the addrinfoW structure pointed to by the *hints* parameter is not supported.
WSAHOST_NOT_FOUND	No such host is known. This error is returned if the name does not resolve for the supplied parameters or the *pNodename* and *pServicename* parameters were not provided.
WSANO_DATA	The requested name is valid, but no data of the requested type was found.
WSANO_RECOVERY	A nonrecoverable error occurred during a database lookup. This error is returned if nonrecoverable error in name resolution occurred.
WSANOTINITIALISED	A successful WSAStartup call must occur before using this function.
WSATRY_AGAIN	This is usually a temporary error during hostname resolution and means that the local server did not receive a response from an authoritative server. This error is returned when a temporary failure in name resolution occurred.
WSATYPE_NOT_FOUND	The specified class was not found. The *pServiceName* parameter is not supported for the specified **ai_socktype** member of the addrinfoW structure pointed to by the *hints* parameter.

Remarks

The **GetAddrInfoW** function is the Unicode version of a function that provides protocol-independent translation from host name to address. The ANSI version of this function is getaddrinfo.

The **GetAddrInfoW** function returns results for the **NS_DNS** namespace. The **GetAddrInfoW** function aggregates all responses if more than one namespace provider returns information. For use with the IPv6 and IPv4 protocol, name resolution can be by the Domain Name System (DNS), a local *hosts* file, or by other naming mechanisms for the **NS_DNS** namespace.

Macros in the Winsock header file define a mixed-case function name of **GetAddrInfo** and a **ADDRINFOT** structure. This **GetAddrInfo** function should be called with the *pNodeName* and *pServiceName* parameters of a pointer of type **TCHAR** and the *pHints* and *ppResult* parameters of a pointer of type **ADDRINFOT**. When UNICODE or _UNICODE is defined, **GetAddrInfo** is defined to **GetAddrInfoW**, the Unicode version of the function, and **ADDRINFOT** is defined to the addrinfoW structure. When UNICODE or _UNICODE is not defined, **GetAddrInfo** is defined to getaddrinfo, the ANSI version of the function, and **ADDRINFOT** is defined to the addrinfo structure.

One or both of the *pNodeName* or *pServiceName* parameters must point to a **NULL**-terminated Unicode string; generally both are provided.

Upon success, a linked list of addrinfoW structures is returned in the *ppResult* parameter. The list can be processed by following the pointer provided in the **ai_next** member of each returned **addrinfoW** structure until a **NULL** pointer is encountered. In each returned **addrinfoW** structure, the **ai_family**, **ai_socktype**, and **ai_protocol** members correspond to respective arguments in a socket or WSASocket function call. Also, the **ai_addr** member in each returned **addrinfoW** structure points to a filled-in socket address structure, the length

of which is specified in its **ai_addrlen** member.

If the *pNodeName* parameter points to a computer name, all permanent addresses for the computer that can be used as a source address are returned. On Windows Vista and later, these addresses would include all unicast IP addresses returned by the GetUnicastIpAddressTable or GetUnicastIpAddressEntry functions in which the **SkipAsSource** member is set to false in the MIB_UNICASTIPADDRESS_ROW structure.

If the *pNodeName* parameter points to a string equal to "localhost", all loopback addresses on the local computer are returned.

If the *pNodeName* parameter contains an empty string, all registered addresses on the local computer are returned.

On Windows Server 2003 and later if the *pNodeName* parameter points to a string equal to "..localmachine", all registered addresses on the local computer are returned.

If the *pNodeName* parameter refers to a cluster virtual server name, only virtual server addresses are returned. On Windows Vista and later, these addresses would include all unicast IP addresses returned by the GetUnicastIpAddressTable or GetUnicastIpAddressEntry functions in which the **SkipAsSource** member is set to true in the MIB_UNICASTIPADDRESS_ROW structure. See Windows Clustering for more information about clustering.

Windows 7 with Service Pack 1 (SP1) and Windows Server 2008 R2 with Service Pack 1 (SP1) add support to Netsh.exe for setting the SkipAsSource attribute on an IP address. This also changes the behavior such that if the **SkipAsSource** member in the MIB_UNICASTIPADDRESS_ROW structure is set to false, the IP address will be registered in DNS. If the **SkipAsSource** member is set to true, the IP address is not registered in DNS.

A hotfix is available for Windows 7 and Windows Server 2008 R2 that adds support to Netsh.exe for setting the SkipAsSource attribute on an IP address. This hotfix also changes behavior such that if the **SkipAsSource** member in the MIB_UNICASTIPADDRESS_ROW structure is set to false, the IP address will be registered in DNS. If the **SkipAsSource** member is set to true, the IP address is not registered in DNS. For more information, see Knowledge Base (KB) 2386184.

A similar hotfix is also available for Windows Vista with Service Pack 2 (SP2) and Windows Server 2008 with Service Pack 2 (SP2) that adds support to Netsh.exe for setting the SkipAsSource attribute on an IP address. This hotfix also changes behavior such that if the **SkipAsSource** member in the MIB_UNICASTIPADDRESS_ROW structure is set to false, the IP address will be registered in DNS. If the **SkipAsSource** member is set to true, the IP address is not registered in DNS.

Callers of the **GetAddrInfoW** function can provide hints about the type of socket supported through an addrinfoW structure pointed to by the *pHints* parameter. When the *pHints* parameter is used, the following rules apply to its associated **addrinfoW** structure:

- A value of **AF_UNSPEC** for **ai_family** indicates the caller will accept only the **AF_INET** and **AF_INET6** address families. Note that **AF_UNSPEC** and **PF_UNSPEC** are the same.
- A value of zero for **ai_socktype** indicates the caller will accept any socket type.
- A value of zero for **ai_protocol** indicates the caller will accept any protocol.
- The **ai_addrlen** member must be set to zero.
- The **ai_canonname** member must be set to **NULL**.
- The **ai_addr** member must be set to **NULL**.
- The **ai_next** member must be set to **NULL**.

Other values in the addrinfoW structure provided in the *pHints* parameter indicate specific requirements. For example, if the caller handles only IPv4 and does not handle IPv6, the **ai_family** member should be set to **AF_INET**. For another example, if the caller handles only TCP and does not handle UDP, the **ai_socktype** member should be set to **SOCK_STREAM**.

If the *pHints* parameter is a **NULL** pointer, the **GetAddrInfoW** function handles it as if the addrinfoW structure in *pHints* were initialized with its **ai_family** member set to **AF_UNSPEC** and all other members set to zero.

On Windows Vista and later when **GetAddrInfoW** is called from a service, if the operation is the result of a user process calling the service, then the service should impersonate the user. This is to allow security to be properly enforced.

The **GetAddrInfoW** function can be used to convert a text string representation of an IP address to an addrinfoW structure that contains a sockaddr structure for the IP address and other information. To be used in this way, the string pointed to by the *pNodeName* parameter must contain a text representation of an IP address and the **addrinfoW** structure pointed to by the *pHints* parameter must have the **AI_NUMERICHOST** flag set in the **ai_flags** member. The string pointed to by the *pNodeName* parameter may contain a text representation of either an IPv4 or an IPv6 address. The text IP address is converted to an **addrinfoW** structure pointed to by the *ppResult* parameter. The returned **addrinfoW** structure contains a **sockaddr** structure for the IP address along with additional information about the IP address. For this method to work with an IPv6 address string on Windows Server 2003 and Windows XP, the IPv6 protocol must be installed on the local computer. Otherwise, the WSAHOST_NOT_FOUND error is returned.

Freeing Address Information from Dynamic Allocation

All information returned by the **GetAddrInfoW** function pointed to by the *ppResult* parameter is dynamically allocated, including all addrinfoW structures, socket address structures, and canonical host name strings pointed to by **addrinfoW** structures. Memory allocated by a successful call to this function must be released with a subsequent call to FreeAddrInfoW.

Example Code

The following code example shows how to use the **GetAddrInfoW** function.

```
#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")

// set APPVER=5.02 for WinXP SP2 and later

int __cdecl wmain(int argc, wchar_t **argv)
{

    //-----------------------------------------
    // Declare and initialize variables
    WSADATA wsaData;
    int iResult;
    INT iRetval;

    DWORD dwRetval;

    int i = 1;

    ADDRINFOW *result = NULL;
    ADDRINFOW *ptr = NULL;
    ADDRINFOW hints;

//    struct sockaddr_in6 *sockaddr_ipv6;
    LPSOCKADDR sockaddr_ip;
```

```
    wchar_t ipstringbuffer[46];
    DWORD ipbufferlength = 46;

    // Validate the parameters
    if (argc != 3) {
        wprintf(L"usage: %ws <hostname> <servicename>\n", argv[0]);
        wprintf(L"getaddrinfow provides protocol-independent translation\n");
        wprintf(L"   from an Unicode host name to an IP address\n");
        wprintf(L"%ws example usage\n", argv[0]);
        wprintf(L"   %ws www.contoso.com 0\n", argv[0]);
        return 1;
    }


    // Initialize Winsock
    iResult = WSAStartup(MAKEWORD(2, 2), &wsaData);
    if (iResult != 0) {
        wprintf(L"WSAStartup failed: %d\n", iResult);
        return 1;
    }

    //--------------------------------
    // Setup the hints address info structure
    // which is passed to the getaddrinfo() function
    ZeroMemory( &hints, sizeof(hints) );
    hints.ai_family = AF_UNSPEC;
    hints.ai_socktype = SOCK_STREAM;
    hints.ai_protocol = IPPROTO_TCP;

    wprintf(L"Calling getaddrinfow with following parameters:\n");
    wprintf(L"\tnodename = %ws\n", argv[1]);
    wprintf(L"\tservname (or port) = %ws\n\n", argv[2]);


//--------------------------------
// Call GetAddrinfoW(). If the call succeeds,
// the result variable will hold a linked list
// of addrinfow structures containing response
// information
    dwRetval = GetAddrInfoW(argv[1], argv[2], &hints, &result);
    if ( dwRetval != 0 ) {
        wprintf(L"GetAddrInfoW failed with error: %d\n", dwRetval);
        WSACleanup();
        return 1;
    }

    wprintf(L"GetAddrInfoW returned success\n");

    // Retrieve each address and print out the hex bytes
    for(ptr=result; ptr != NULL ;ptr=ptr->ai_next) {

        wprintf(L"GetAddrInfoW response %d\n", i++);
        wprintf(L"\tFlags: 0x%x\n", ptr->ai_flags);
        wprintf(L"\tFamily: ");
        switch (ptr->ai_family) {
            case AF_UNSPEC:
                wprintf(L"Unspecified\n");
                break;
            case AF_INET:
                wprintf(L"AF_INET (IPv4)\n");
                sockaddr_ip = (LPSOCKADDR) ptr->ai_addr;
                // The buffer length is changed by each call to WSAAddresstoString
                // So we need to set it for each iteration through the loop for safety
                ipbufferlength = 46;
                iRetval = WSAAddressToString(sockaddr_ip, (DWORD) ptr->ai_addrlen, NULL,
                    ipstringbuffer, &ipbufferlength );
                if (iRetval)
                    wprintf(L"WSAAddressToString failed with %u\n", WSAGetLastError() );
                else
```

```
            else
                wprintf(L"\tIPv4 address %ws\n", ipstringbuffer);
            break;
        case AF_INET6:
            wprintf(L"AF_INET6 (IPv6)\n");
            // the InetNtop function is available on Windows Vista and later
            // sockaddr_ipv6 = (struct sockaddr_in6 *) ptr->ai_addr;
            // printf("\tIPv6 address %s\n",
            //    InetNtop(AF_INET6, &sockaddr_ipv6->sin6_addr, ipstringbuffer, 46) );

            // We use WSAAddressToString since it is supported on Windows XP and later
            sockaddr_ip = (LPSOCKADDR) ptr->ai_addr;
            // The buffer length is changed by each call to WSAAddresstoString
            // So we need to set it for each iteration through the loop for safety
            ipbufferlength = 46;
            iRetval = WSAAddressToString(sockaddr_ip, (DWORD) ptr->ai_addrlen, NULL,
                ipstringbuffer, &ipbufferlength );
            if (iRetval)
                wprintf(L"WSAAddressToString failed with %u\n", WSAGetLastError() );
            else
                wprintf(L"\tIPv6 address %ws\n", ipstringbuffer);
            break;
        default:
            wprintf(L"Other %ld\n", ptr->ai_family);
            break;
    }
    wprintf(L"\tSocket type: ");
    switch (ptr->ai_socktype) {
        case 0:
            wprintf(L"Unspecified\n");
            break;
        case SOCK_STREAM:
            wprintf(L"SOCK_STREAM (stream)\n");
            break;
        case SOCK_DGRAM:
            wprintf(L"SOCK_DGRAM (datagram) \n");
            break;
        case SOCK_RAW:
            wprintf(L"SOCK_RAW (raw) \n");
            break;
        case SOCK_RDM:
            wprintf(L"SOCK_RDM (reliable message datagram)\n");
            break;
        case SOCK_SEQPACKET:
            wprintf(L"SOCK_SEQPACKET (pseudo-stream packet)\n");
            break;
        default:
            wprintf(L"Other %ld\n", ptr->ai_socktype);
            break;
    }
    wprintf(L"\tProtocol: ");
    switch (ptr->ai_protocol) {
        case 0:
            wprintf(L"Unspecified\n");
            break;
        case IPPROTO_TCP:
            wprintf(L"IPPROTO_TCP (TCP)\n");
            break;
        case IPPROTO_UDP:
            wprintf(L"IPPROTO_UDP (UDP) \n");
            break;
        default:
            wprintf(L"Other %ld\n", ptr->ai_protocol);
            break;
    }
    wprintf(L"\tLength of this sockaddr: %d\n", ptr->ai_addrlen);
    wprintf(L"\tCanonical name: %s\n", ptr->ai_canonname);
}

FreeAddrInfoW(result);
```

```
    FreeAddrInfoW(result);
    WSACleanup();

    return 0;
}
```

> **Note** Ensure that the development environment targets the newest version of *Ws2tcpip.h* which includes structure and function definitions for addrinfoW and **GetAddrInfoW**, respectively.

Internationalized Domain Names

Internet host names typically consist of a very restricted set of characters:

- Upper and lower case ASCII letters from the English alphabet.
- Digits from 0 to 9.
- ASCII hyphen characters.

With the growth of the Internet, there is a growing need to identify Internet host names for other languages not represented by the ASCII character set. Identifiers which facilitate this need and allow non-ASCII characters (Unicode) to be represented as special ASCII character strings are known as Internationalized Domain Names (IDNs). A mechanism called Internationalizing Domain Names in Applications (IDNA) is used to handle IDNs in a standard fashion. The specifications for IDNs and IDNA are documented in RFC 3490, RTF 5890, and RFC 6365 published by the Internet Engineering Task Force (IETF).

On Windows 8 and Windows Server 2012, the **GetAddrInfoW** function provides support for Internationalized Domain Name (IDN) parsing applied to the name passed in the *pNodeName* parameter. Winsock performs Punycode/IDN encoding and conversion. This behavior can be disabled using the **AI_DISABLE_IDN_ENCODING** flag discussed below.

On Windows 7 and Windows Server 2008 R2 or earlier, the **GetAddrInfoW** function does not currently provide support for IDN parsing applied to the name passed in the *pNodeName* parameter. Winsock does not perform any Punycode/IDN conversion. The **GetAddrInfoW** function does not use Punycode to convert an IDN as per RFC 3490. The **GetAddrInfoW** function when querying DNS encodes the Unicode name in UTF-8 format, the format used by Microsoft DNS servers in an enterprise environment.

Several functions on Windows Vista and later support conversion between Unicode labels in an IDN to their ASCII equivalents. The resulting representation of each Unicode label contains only ASCII characters and starts with the xn-- prefix if the Unicode label contained any non-ASCII characters. The reason for this is to support existing DNS servers on the Internet, since some DNS tools and servers only support ASCII characters (see RFC 3490).

The IdnToAscii function use Punycode to convert an IDN to the ASCII representation of the original Unicode string using the standard algorithm defined in RFC 3490. The IdnToUnicode function converts the ASCII form of an IDN to the normal Unicode UTF-16 encoding syntax. For more information and links to related draft standards, see Handling Internationalized Domain Names (IDNs).

The IdnToAscii function can be used to convert an IDN name to the ASCII form. To pass this ASCII form to the **GetAddrInfoW** function, you can use the MultiByteToWideChar function to convert the **CHAR** string into a **WCHAR** string that then can be passed in the *pNodeName* parameter to the **GetAddrInfoW** function.

Use of ai_flags in the hints parameter

Flags in the **ai_flags** member of the optional addrinfoW structure provided in the *pHints* parameter modify the behavior of the function.

These flag bits are defined in the *Ws2def.h* header file on the Microsoft Windows Software Development Kit

(SDK) for Windows 7. These flag bits are defined in the *Ws2tcpip.h* header file on the Windows SDK for Windows Server 2008 and Windows Vista. These flag bits are defined in the *Ws2tcpip.h* header file on the Platform Software Development Kit (SDK) for Windows Server 2003, and Windows XP.

The flag bits can be a combination of the following:

FLAG BITS	DESCRIPTION
AI_PASSIVE	Setting the **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 *pNodeName* is a **NULL** pointer, the IP address portion of the socket address structure is set to **INADDR_ANY** for IPv4 addresses and **IN6ADDR_ANY_INIT** for IPv6 addresses. When the **AI_PASSIVE** flag is not set, the returned socket address structure is ready for a call to the connect function for a connection-oriented protocol, or ready for a call to either the **connect**, sendto, or send functions for a connectionless protocol. If the *pNodeName* parameter is a **NULL** pointer in this case, the IP address portion of the socket address structure is set to the loopback address.
AI_CANONNAME	If neither **AI_CANONNAME** nor **AI_NUMERICHOST** is used, the **GetAddrInfoW** function attempts resolution. If a literal string is passed **GetAddrInfoW** attempts to convert the string, and if a host name is passed the **GetAddrInfoW** function attempts to resolve the name to an address or multiple addresses. When the **AI_CANONNAME** bit is set, the *pNodeName* parameter cannot be **NULL**. Otherwise the GetAddrInfoEx function will fail with WSANO_RECOVERY. When the **AI_CANONNAME** bit is set and the **GetAddrInfoW** function returns success, the **ai_canonname** member in the *ppResult* parameter points to a **NULL**-terminated string that contains the canonical name of the specified node. **Note** The **GetAddrInfoW** function can return success when the **AI_CANONNAME** flag is set, yet the **ai_canonname** member in the associated addrinfoW structure is **NULL**. Therefore, the recommended use of the **AI_CANONNAME** flag includes testing whether the **ai_canonname** member in the associated **addrinfoW** structure is **NULL**.
AI_NUMERICHOST	When the **AI_NUMERICHOST** bit is set, the *pNodeName* parameter must contain a non-**NULL** numeric host address string, otherwise the **EAI_NONAME** error is returned. This flag prevents a name resolution service from being called.

AI_NUMERICSERV	When the **AI_NUMERICSERV** bit is set, the *pServiceName* parameter must contain a non-**NULL** numeric port number, otherwise the **EAI_NONAME** error is returned. This flag prevents a name resolution service from being called. The **AI_NUMERICSERV** flag is defined on Windows SDK for Windows Vista and later. The **AI_NUMERICSERV** flag is not supported by Microsoft providers.
AI_ALL	If the **AI_ALL** bit is set, a request is made for IPv6 addresses and IPv4 addresses with **AI_V4MAPPED**. The **AI_ALL** flag is defined on the Windows SDK for Windows Vista and later. The **AI_ALL** flag is supported on Windows Vista and later.
AI_ADDRCONFIG	If the **AI_ADDRCONFIG** bit is set, **GetAddrInfoW** will resolve only if a global address is configured. If **AI_ADDRCONFIG** flag is specified, IPv4 addresses shall be returned only if an IPv4 address is configured on the local system, and IPv6 addresses shall be returned only if an IPv6 address is configured on the local system. The IPv4 or IPv6 loopback address is not considered a valid global address. The **AI_ADDRCONFIG** flag is defined on the Windows SDK for Windows Vista and later. The **AI_ADDRCONFIG** flag is supported on Windows Vista and later.
AI_V4MAPPED	If the **AI_V4MAPPED** bit is set and a request for IPv6 addresses fails, a name service request is made for IPv4 addresses and these addresses are converted to IPv4-mapped IPv6 address format. The **AI_V4MAPPED** flag is defined on the Windows SDK for Windows Vista and later. The **AI_V4MAPPED** flag is supported on Windows Vista and later.
AI_NON_AUTHORITATIVE	If the **AI_NON_AUTHORITATIVE** bit is set, the **NS_EMAIL** namespace provider returns both authoritative and non-authoritative results. If the **AI_NON_AUTHORITATIVE** bit is not set, the **NS_EMAIL** namespace provider returns only authoritative results. The **AI_NON_AUTHORITATIVE** flag is defined on the Windows SDK for Windows Vista and later. The **AI_NON_AUTHORITATIVE** flag is supported on Windows Vista and later and applies only to the **NS_EMAIL** namespace.
AI_SECURE	If the **AI_SECURE** bit is set, the **NS_EMAIL** namespace provider will return results that were obtained with enhanced security to minimize possible spoofing. The **AI_SECURE** flag is defined on the Windows SDK for Windows Vista and later. The **AI_SECURE** flag is supported on Windows Vista and later and applies only to the **NS_EMAIL** namespace.

AI_RETURN_PREFERRED_NAMES	If the **AI_RETURN_PREFERRED_NAMES** is set, then no name should be provided in the *pNodeName* parameter. The **NS_EMAIL** namespace provider will return preferred names for publication. The **AI_RETURN_PREFERRED_NAMES** flag is defined on the Windows SDK for Windows Vista and later. The **AI_RETURN_PREFERRED_NAMES** flag is supported on Windows Vista and later and applies only to the **NS_EMAIL** namespace.
AI_FQDN	If the **AI_FQDN** is set and a flat name (single label) is specified, **GetAddrInfoW** will return the fully qualified domain name that the name eventually resolved to. The fully qualified domain name is returned in the **ai_canonname** member in the associated addrinfoW structure. This is different than **AI_CANONNAME** bit flag that returns the canonical name registered in DNS which may be different than the fully qualified domain name that the flat name resolved to. Only one of the **AI_FQDN** and **AI_CANONNAME** bits can be set. The **GetAddrInfoW** function will fail if both flags are present with **EAI_BADFLAGS**. When the **AI_FQDN** bit is set, the *pNodeName* parameter cannot be **NULL**. Otherwise the GetAddrInfoEx function will fail with WSANO_RECOVERY. **Windows 7:** The **AI_FQDN** flag is defined on the Windows SDK for Windows 7 and later. The **AI_FQDN** flag is supported on Windows 7 and later.
AI_FILESERVER	If the **AI_FILESERVER** is set, this is a hint to the namespace provider that the hostname being queried is being used in file share scenario. The namespace provider may ignore this hint. **Windows 7:** The **AI_FILESERVER** flag is defined on the Windows SDK for Windows 7 and later. The **AI_FILESERVER** flag is supported on Windows 7 and later.
AI_DISABLE_IDN_ENCODING	If the **AI_DISABLE_IDN_ENCODING** is set, this disables the automatic International Domain Name encoding using Punycode in the name resolution functions called by the **GetAddrInfoW** function. **Windows 8:** The **AI_DISABLE_IDN_ENCODING** flag is defined on the Windows SDK for Windows 8 and later. The **AI_DISABLE_IDN_ENCODING** flag is supported on Windows 8 and later.

Windows 8.1 and **Windows Server 2012 R2**: This function is supported for Windows Store apps on Windows 8.1, Windows Server 2012 R2, and later.

> **NOTE**
> The ws2tcpip.h header defines GetAddrInfo as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see Conventions for Function Prototypes.

Requirements

Minimum supported client	Windows 8.1, Windows Vista [desktop apps | UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps | UWP apps]
Target Platform	Windows
Header	ws2tcpip.h
Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

FreeAddrInfoW

GetAddrInfoEx

GetHostNameW

IdnToAscii

IdnToUnicode

WSAGetLastError

WSASocket

Winsock Functions

Winsock Reference

addrinfo

addrinfoW

addrinfoex

addrinfoex2

bind

connect

gai_strerror

getaddrinfo

send

sendto

socket

getipv4sourcefilter function (ws2tcpip.h)

10/22/2022 • 2 minutes to read • Edit Online

The **getipv4sourcefilter** inline function retrieves the multicast filter state for an IPv4 socket.

Syntax

```
int getipv4sourcefilter(
  [in]      SOCKET              Socket,
  [in]      IN_ADDR             Interface,
  [in]      IN_ADDR             Group,
  [out]     MULTICAST_MODE_TYPE *FilterMode,
  [in, out] ULONG               *SourceCount,
  [out]     IN_ADDR             *SourceList
);
```

Parameters

`[in] Socket`

A descriptor that identifies a multicast socket.

`[in] Interface`

The local IPv4 address of the interface or the interface index on which the multicast group should be joined or dropped.

This value is in network byte order. If this member specifies an IPv4 address of 0.0.0.0, the default IPv4 multicast interface is used.

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).

To use an interface index of 1 would be the same as an IP address of 0.0.0.1.

`[in] Group`

The IPv4 address of the multicast group.

`[out] FilterMode`

A pointer to a value to receive the multicast filter mode for multicast group address when the function returns.

`[in, out] SourceCount`

On input, a pointer to a value that indicates the maximum number of source addresses that will fit in the buffer pointed to by the *SourceList* parameter.

On output, a pointer to a value that indicates the total number of source addresses associated with the multicast filter.

`[out] SourceList`

A pointer to a buffer to receive the list of IP addresses associated with the multicast filter.

If *SourceCount* is zero on input, a **NULL** pointer may be supplied.

Return value

On success, **getipv4sourcefilter** returns NO_ERROR (0). Any nonzero return value indicates failure and a specific error code can be retrieved by calling WSAGetLastError.

ERROR CODE	MEANING
WSAENOBUFS	Insufficient buffer space is available.
WSAENOTSOCK	The descriptor is not a socket.

Remarks

The **getipv4sourcefilter** inline function is used to retrieve the multicast filter state for an IPv4 socket.

If the app does not know the size of the source list beforehand, it can make a guess (zero, for example). If upon completion, the *SourceCount* parameter holds a larger value, the operation can be repeated with a large enough buffer.

On return, the *SourceCount* parameter is always updated to be the total number of sources in the filter, while the buffer pointed to by the *SourceList* parameter will hold as many source addresses as fit, up to the minimum of the array size passed in as the original *SourceCount* value and the total number of sources in the filter.

This function is part of socket interface extensions for multicast source filters defined in RFC 3678. An app can use these functions to retrieve and set the multicast source address filters associated with a socket.

Windows Phone 8: This function is supported for Windows Phone Store apps on Windows Phone 8 and later.

Windows 8.1 and **Windows Server 2012 R2**: This function is supported for Windows Store apps on Windows 8.1, Windows Server 2012 R2, and later.

Requirements

Minimum supported client	Windows 8.1, Windows Vista [desktop apps | UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps | UWP apps]
Target Platform	Windows
Header	ws2tcpip.h
Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

MULTICAST_MODE_TYPE

getsourcefilter

in_addr

setipv4sourcefilter

setsourcefilter

getnameinfo function (ws2tcpip.h)

10/22/2022 • 8 minutes to read • Edit Online

The **getnameinfo** function provides protocol-independent name resolution from an address to an ANSI host name and from a port number to the ANSI service name.

Syntax

```
INT WSAAPI getnameinfo(
  [in]  const SOCKADDR *pSockaddr,
  [in]  socklen_t      SockaddrLength,
  [out] PCHAR          pNodeBuffer,
  [in]  DWORD          NodeBufferSize,
  [out] PCHAR          pServiceBuffer,
  [in]  DWORD          ServiceBufferSize,
  [in]  INT            Flags
);
```

Parameters

`[in] pSockaddr`

A pointer to a socket address structure that contains the address and port number of the socket. For IPv4, the *sa* parameter points to a sockaddr_in structure. For IPv6, the *sa* parameter points to a **sockaddr_in6** structure.

`[in] SockaddrLength`

The length, in bytes, of the structure pointed to by the *sa* parameter.

`[out] pNodeBuffer`

A pointer to an ANSI string used to hold the host name. On success, the host name is returned as a Fully Qualified Domain Name (FQDN) by default. If the *host* parameter is **NULL**, this indicates the caller does not want to receive a host name string.

`[in] NodeBufferSize`

The length, in bytes, of the buffer pointed to by the *host* parameter. The caller must provide a buffer large enough to hold the host name, including the terminating **NULL** character.

`[out] pServiceBuffer`

A pointer to an ANSI string to hold the service name. On success, an ANSI string that represents the service name associated with the port number is returned. If the *serv* parameter is **NULL**, this indicates the caller does not want to receive a service name string.

`[in] ServiceBufferSize`

The length, in bytes, of the buffer pointed to by the *serv* parameter. The caller must provide a buffer large enough to hold the service name, including the terminating **NULL** character.

`[in] Flags`

A value used to customize processing of the **getnameinfo** function. See the Remarks section.

Return value

On success, **getnameinfo** returns zero. Any nonzero return value indicates failure and a specific error code can be retrieved by calling WSAGetLastError.

Nonzero error codes returned by the **getnameinfo** function also map to the set of errors outlined by Internet Engineering Task Force (IETF) recommendations. The following table lists these error codes and their WSA equivalents. It is recommended that the WSA error codes be used, as they offer familiar and comprehensive error information for Winsock programmers.

ERROR VALUE	WSA EQUIVALENT	DESCRIPTION
EAI_AGAIN	WSATRY_AGAIN	A temporary failure in name resolution occurred.
EAI_BADFLAGS	WSAEINVAL	One or more invalid parameters was passed to the **getnameinfo** function. This error is returned if a host name was requested but the *hostlen* parameter was zero or if a service name was requested, but the *servlen* parameter was zero.
EAI_FAIL	WSANO_RECOVERY	A nonrecoverable failure in name resolution occurred.
EAI_FAMILY	WSAEAFNOSUPPORT	The **sa_family** member of socket address structure pointed to by the *sa* parameter is not supported.
EAI_MEMORY	WSA_NOT_ENOUGH_MEMORY	A memory allocation failure occurred.
EAI_NONAME	WSAHOST_NOT_FOUND	A service name was requested, but no port number was found in the structure pointed to by the *sa* parameter or no service name matching the port number was found. NI_NAMEREQD is set and the host name cannot be located, or both the *host* and *serv* parameters were **NULL**.

Use the gai_strerror function to print error messages based on the EAI codes returned by the **getnameinfo** function. The **gai_strerror** function is provided for compliance with IETF recommendations, but it is not thread safe. Therefore, use of traditional Windows Sockets functions such as WSAGetLastError is recommended.

In addition, the following error codes can be returned.

ERROR CODE	MEANING
WSAEFAULT	This error is returned if the *sa* parameter is **NULL** or the *salen* parameter is less than the length required for the size of sockaddr_in structure for IPv4 or the **sockaddr_in6** structure for IPv6.

Remarks

The **getnameinfo** function is the ANSI version of a function that provides protocol-independent name

resolution. The **getnameinfo** function is used to translate the contents of a socket address structure to a node name and/or a service name.

For IPv6 and IPv4 protocols, Name resolution can be by the Domain Name System (DNS), a local *hosts* file, or by other naming mechanisms. This function can be used to determine the host name for an IPv4 or IPv6 address, a reverse DNS lookup, or determine the service name for a port number. The **getnameinfo** function can also be used to convert an IP address or a port number in a **sockaddr** structure to an ANSI string. This function can also be used to determine the IP address for a host name.

Another name that can be used for the **getnameinfo** function is **GetNameInfoA**. Macros in the *Ws2tcpip.h* header file define **GetNameInfoA** to **getnameinfo**.

The Unicode version of this function available on Windows XP with Service Pack 2 (SP2) and later is GetNameInfoW.

Macros in the Winsock header file define a mixed-case function name of **GetNameInfo** that can be used when the application is targeted for Windows XP with SP2 and later (_WIN32_WINNT >= 0x0502). This **GetNameInfo** function should be called with the *host* and *serv* parameters of a pointer of type **TCHAR**. When UNICODE or _UNICODE is not defined, **GetNameInfo** is defined to the ANSI version and **getnameinfo** is called with the *host* and *serv* parameters of a pointer of type **char**. When UNICODE or _UNICODE is defined, **GetNameInfo** is defined to the Unicode version and GetNameInfoW is called with the *pNodeBuffer* and *pServiceBuffer* parameters of a pointer of type **PWCHAR**.

To simplify determining buffer requirements for the *host* and *serv* parameters, the following values for maximum host name length and maximum service name are defined in the *Ws2tcpip.h* header file.

```
#define NI_MAXSERV    32
#define NI_MAXHOST  1025
```

The *flags* parameter can be used to customize processing of the **getnameinfo** function. The following flags are available:

- NI_NOFQDN
- NI_NUMERICHOST
- NI_NAMEREQD
- NI_NUMERICSERV
- NI_DGRAM

When the **NI_NAMEREQD** flag is set, a host name that cannot be resolved by DNS results in an error.

Setting the **NI_NOFQDN** flag results in local hosts having only their Relative Distinguished Name (RDN) returned in the *host* parameter.

Setting the **NI_NUMERICHOST** flag returns the numeric form of the host name instead of its name. The numeric form of the host name is also returned if the host name cannot be resolved by DNS.

Setting the **NI_NUMERICSERV** flag returns the port number of the service instead of its name. Also, if a host name is not found for an IP address (127.0.0.2, for example), the hostname is returned as the IP address.

On Windows Vista and later, if **NI_NUMERICSERV** is not specified in the *flags* parameter and the port number contained in sockaddr structure pointed to by the *sa* parameter does not resolve to a well known service, the **getnameinfo** function returns the numeric form of the service address (the port number) as a numeric string. When **NI_NUMERICSERV** is specified, the port number is returned as a numeric string. This behavior is specified in section 6.2 of RFC 3493. For more information, see www.ietf.org/rfc3493.txt

On Windows Server 2003 and earlier, if **NI_NUMERICSERV** is not specified in the *flags* parameter, and the port

number contained in the **sockaddr** structure pointed to by the *sa* parameter does not resolve to a well known service, the **getnameinfo** function fails. When **NI_NUMERICSERV** is specified, the port number is returned as a numeric string.

Setting the **NI_DGRAM** flag indicates that the service is a datagram service. This flag is necessary for the few services that provide different port numbers for UDP and TCP service.

> **Note** The ability to perform reverse DNS lookups using the **getnameinfo** function is convenient, but such lookups are considered inherently unreliable, and should be used only as a hint.

> **Note** The **getnameinfo** function cannot be used to resolve alias names.

Example Code

The following code example shows how to use the **getnameinfo** function.

```
#include <winsock2.h>
#include <ws2tcpip.h>
#include <stdio.h>

// link with ws2_32.lib
#pragma comment(lib, "Ws2_32.lib")

int __cdecl main(int argc, char **argv)
{

    //-----------------------------------------
    // Declare and initialize variables
    WSADATA wsaData = {0};
    int iResult = 0;

    DWORD dwRetval;

    struct sockaddr_in saGNI;
    char hostname[NI_MAXHOST];
    char servInfo[NI_MAXSERV];
    u_short port = 27015;

    // Validate the parameters
    if (argc != 2) {
        printf("usage: %s IPv4 address\n", argv[0]);
        printf("  to return hostname\n");
        printf("       %s 127.0.0.1\n", argv[0]);
        return 1;
    }
    // Initialize Winsock
    iResult = WSAStartup(MAKEWORD(2, 2), &wsaData);
    if (iResult != 0) {
        printf("WSAStartup failed: %d\n", iResult);
        return 1;
    }
    //-----------------------------------------
    // Set up sockaddr_in structure which is passed
    // to the getnameinfo function
    saGNI.sin_family = AF_INET;
    saGNI.sin_addr.s_addr = inet_addr(argv[1]);
    saGNI.sin_port = htons(port);

    //-----------------------------------------
    // Call getnameinfo
    dwRetval = getnameinfo((struct sockaddr *) &saGNI,
                           sizeof (struct sockaddr),
                           hostname,
                           NI_MAXHOST, servInfo, NI_MAXSERV, NI_NUMERICSERV);

    if (dwRetval != 0) {
        printf("getnameinfo failed with error # %ld\n", WSAGetLastError());
        return 1;
    } else {
        printf("getnameinfo returned hostname = %s\n", hostname);
        return 0;
    }
}
```

Support for getnameinfo on older versions of Windows

The **getnameinfo** function was added to the *Ws2_32.dll* on Windows XP and later. If you want to execute an application using this function on earlier versions of Windows (Windows 2000, Windows NT, and Windows Me/98/95), then you need to include the *Ws2tcpip.h* file and also include the *Wspiapi.h* file. When the *Wspiapi.h* include file is added, the **getnameinfo** function is defined to the WspiapiGetNameInfo inline function in

the *Wspiapi.h* file. At runtime, the WspiapiGetNameInfo function is implemented in such a way that if the *Ws2_32.dll* or the *Wship6.dll* (the file containing **getnameinfo** in the IPv6 Technology Preview for Windows 2000) does not include **getnameinfo**, then a version of **getnameinfo** is implemented inline based on code in the *Wspiapi.h* header file. This inline code will be used on older Windows platforms that do not natively support the **getnameinfo** function.

The IPv6 protocol is supported on Windows 2000 when the IPv6 Technology Preview for Windows 2000 is installed. Otherwise **getnameinfo** support on versions of Windows earlier than Windows XP is limited to handling IPv4 name resolution.

The GetNameInfoW function is the Unicode version of **getnameinfo**. The **GetNameInfoW** function was added to the *Ws2_32.dll* in Windows XP with SP2. The **GetNameInfoW** function cannot be used on versions of Windows earlier than Windows XP with SP2.

Windows Phone 8: This function is supported for Windows Phone Store apps on Windows Phone 8 and later.

Windows 8.1 and **Windows Server 2012 R2**: This function is supported for Windows Store apps on Windows 8.1, Windows Server 2012 R2, and later.

Requirements

Minimum supported client	Windows 8.1, Windows Vista [desktop apps | UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps | UWP apps]
Target Platform	Windows
Header	ws2tcpip.h
Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

GetNameInfoW

WSAGetLastError

Winsock Functions

Winsock Reference

gai_strerror

getaddrinfo

sockaddr

GetNameInfoW function (ws2tcpip.h)

10/22/2022 • 7 minutes to read • Edit Online

The **GetNameInfoW** function provides protocol-independent name resolution from an address to a Unicode host name and from a port number to the Unicode service name.

Syntax

```
INT WSAAPI GetNameInfoW(
  [in]  const SOCKADDR *pSockaddr,
  [in]  socklen_t      SockaddrLength,
  [out] PWCHAR         pNodeBuffer,
  [in]  DWORD          NodeBufferSize,
  [out] PWCHAR         pServiceBuffer,
  [in]  DWORD          ServiceBufferSize,
  [in]  INT            Flags
);
```

Parameters

`[in] pSockaddr`

A pointer to a socket address structure containing the IP address and port number of the socket. For IPv4, the *pSockaddr* parameter points to a sockaddr_in structure. For IPv6, the *pSockaddr* parameter points to a **sockaddr_in6** structure.

`[in] SockaddrLength`

The length, in bytes, of the structure pointed to by the *pSockaddr* parameter.

`[out] pNodeBuffer`

A pointer to a Unicode string to hold the host name. On success, a pointer to the Unicode host name is returned as a Fully Qualified Domain Name (FQDN) by default. If the *pNodeBuffer* parameter is **NULL**, this indicates the caller does not want to receive a host name string.

`[in] NodeBufferSize`

The number of **WCHAR** characters in the buffer pointed to by the *pNodeBuffer* parameter. The caller must provide a buffer large enough to hold the Unicode host name, including the terminating **NULL** character.

`[out] pServiceBuffer`

A pointer to a Unicode string to hold the service name. On success, a pointer is returned to a Unicode string representing the service name associated with the port number. If the *pServiceBuffer* parameter is **NULL**, this indicates the caller does not want to receive a service name string.

`[in] ServiceBufferSize`

The number of **WCHAR** characters in the buffer pointed to by the *pServiceBuffer* parameter. The caller must provide a buffer large enough to hold the Unicode service name, including the terminating **NULL** character.

`[in] Flags`

A value used to customize processing of the **GetNameInfoW** function. See the Remarks section.

Return value

On success, **GetNameInfoW** returns zero. Any nonzero return value indicates failure and a specific error code can be retrieved by calling WSAGetLastError.

Nonzero error codes returned by the **GetNameInfoW** function also map to the set of errors outlined by Internet Engineering Task Force (IETF) recommendations. The following table shows these error codes and their WSA equivalents. It is recommended that the WSA error codes be used, as they offer familiar and comprehensive error information for Winsock programmers.

ERROR VALUE	WSA EQUIVALENT	DESCRIPTION
EAI_AGAIN	WSATRY_AGAIN	A temporary failure in name resolution occurred.
EAI_BADFLAGS	WSAEINVAL	One or more invalid parameters was passed to the **GetNameInfoW** function. This error is returned if a host name was requested but the *NodeBufferSize* parameter was zero or if a service name was requested but the *ServiceBufferSize* parameter was zero.
EAI_FAIL	WSANO_RECOVERY	A nonrecoverable failure in name resolution occurred.
EAI_FAMILY	WSAEAFNOSUPPORT	The **sa_family** member of socket address structure pointed to by the *pSockaddr* parameter is not supported.
EAI_MEMORY	WSA_NOT_ENOUGH_MEMORY	A memory allocation failure occurred.
EAI_NONAME	WSAHOST_NOT_FOUND	A service name was requested, but no port number was found in the structure pointed to by the *pSockaddr* parameter or no service name matching the port number was found. NI_NAMEREQD is set and the host's name cannot be located, or both the *pNodeBuffer* and *pServiceBuffer* parameters were **NULL**.

You can use the gai_strerror function to print error messages based on the EAI codes returned by the **GetNameInfoW** function. The **gai_strerror** function is provided for compliance with IETF recommendations, but it is not thread safe. Therefore, use of traditional Windows Sockets functions such as WSAGetLastError is recommended.

In addition, the following error codes can be returned.

ERROR CODE	MEANING
WSAEFAULT	This error is returned if the *pSockaddr* parameter is **NULL** or the *SockaddrLength* parameter is less than the length needed for the size of sockaddr_in structure for IPv4 or the **sockaddr_in6** structure for IPv6.

Remarks

The **GetNameInfoW** function is the Unicode version of a function that provides protocol-independent name resolution. The **GetNameInfoW** function is used to translate the contents of a socket address structure to a node name and/or a service name.

For the IPv6 and IPv4 protocols, name resolution can be by the Domain Name System (DNS), a local *hosts* file, or by other naming mechanisms. This function can be used to determine the host name for an IPv4 or IPv6 address, a reverse DNS lookup, or determine the service name for a port number. The **GetNameInfoW** function can also be used to convert an IP address or a port number in a **SOCKADDR** structure to an Unicode string. This function can also be used to determine the IP address for a host name.

The ANSI version of this function is getnameinfo.

Macros in the Winsock header file define a mixed-case function name of **GetNameInfo** that can be used when the application is targeted for Windows XP with Service Pack 2 (SP2) and later (_WIN32_WINNT >= 0x0502). This **GetNameInfo** function should be called with the *pNodeBuffer* and *pServiceBuffer* parameters of a pointer of type **TCHAR**. When UNICODE or _UNICODE is defined, **GetNameInfo** is defined to the Unicode version and **GetNameInfoW** is called with the *host* and *serv* parameters of a pointer of type **char**. When UNICODE or _UNICODE is not defined, **GetNameInfo** is defined to the ANSI version and getnameinfo is called with the *pNodeBuffer* and *pServiceBuffer* parameters of a pointer of type **PWCHAR**.

To simplify determining buffer requirements for the *pNodeBuffer* and *pServiceBuffer* parameters, the following values for maximum host name length and maximum service name are defined in the *Ws2tcpip.h* header file:

```
#include <windows.h>

#define NI_MAXSERV    32
#define NI_MAXHOST  1025
```

The *Flags* parameter can be used to customize processing of the **GetNameInfoW** function. The following flags are available:

- NI_NOFQDN
- NI_NUMERICHOST
- NI_NAMEREQD
- NI_NUMERICSERV
- NI_DGRAM

When the **NI_NAMEREQD** flag is set, a host name that cannot be resolved by the DNS results in an error.

Setting the **NI_NOFQDN** flag results in local hosts having only their Relative Distinguished Name (RDN) returned in the *pNodeBuffer* parameter.

Setting the **NI_NUMERICHOST** flag returns the numeric form of the host name instead of its name. The numeric form of the host name is also returned if the host name cannot be resolved by DNS.

Setting the **NI_NUMERICSERV** flag returns the port number of the service instead of its name. Also, if a host name is not found for an IP address (127.0.0.2, for example), the hostname is returned as the IP address.

On Windows Vista and later, if **NI_NUMERICSERV** is not specified in the *flags* parameter, and the port number contained in **sockaddr** structure pointed to by the *sa* parameter does not resolve to a well known service, the **GetNameInfoW** function returns the numeric form of the service address (the port number) as a numeric string. When **NI_NUMERICSERV** is specified, the port number is returned as a numeric string. This behavior is specified in section 6.2 of RFC 3493. For more information, see www.ietf.org/rfc/rfc3493.txt

On Windows Server 2003 and earlier, if **NI_NUMERICSERV** is not specified in the *flags* parameter and the port number contained in sockaddr structure pointed to by the *sa* parameter does not resolve to a well known service, the **GetNameInfoW** function fails. When **NI_NUMERICSERV** is specified, the port number is returned as a numeric string.

Setting the **NI_DGRAM** flag indicates that the service is a datagram service. This flag is necessary for the few services that provide different port numbers for UDP and TCP service.

> **Note** The capability to perform reverse DNS lookups using the **GetNameInfoW** function is convenient, but such lookups are considered inherently unreliable, and should be used only as a hint.

> **Note** **GetNameInfoW** cannot be used to resolve alias names.

Windows Phone 8: This function is supported for Windows Phone Store apps on Windows Phone 8 and later.

Windows 8.1 and **Windows Server 2012 R2**: This function is supported for Windows Store apps on Windows 8.1, Windows Server 2012 R2, and later.

Example Code

The following example demonstrates the use of the **GetNameInfoW** function.

```
#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 __cdecl main(int argc, char **argv)
{

    //-----------------------------------------
    // Declare and initialize variables
    WSADATA wsaData;
    int iResult;

    DWORD dwRetval;

    struct sockaddr_in saGNI;
    WCHAR hostname[NI_MAXHOST];
    WCHAR servInfo[NI_MAXSERV];
    u_short port = 27015;

    // Validate the parameters
    if (argc != 2) {
        wprintf(L"usage: %s IPv4 address\n", argv[0]);
        wprintf(L"  to return hostname\n");
        wprintf(L"       %s 127.0.0.1\n", argv[0]);
        return 1;
    }
    // Initialize Winsock
    iResult = WSAStartup(MAKEWORD(2, 2), &wsaData);
    if (iResult != 0) {
        wprintf(L"WSAStartup failed: %d\n", iResult);
        return 1;
    }
    //-----------------------------------------
    // Set up sockaddr_in structure which is passed
    // to the getnameinfo function
    saGNI.sin_family = AF_INET;
    saGNI.sin_addr.s_addr = inet_addr(argv[1]);
    saGNI.sin_port = htons(port);

    //-----------------------------------------
    // Call GetNameInfoW
    dwRetval = GetNameInfoW((struct sockaddr *) &saGNI,
                            sizeof (struct sockaddr),
                            hostname,
                            NI_MAXHOST, servInfo, NI_MAXSERV, NI_NUMERICSERV);

    if (dwRetval != 0) {
        wprintf(L"GetNameInfoW failed with error # %ld\n", WSAGetLastError());
        return 1;
    } else {
        wprintf(L"GetNameInfoW returned hostname = %ws\n", hostname);
        return 0;
    }
}
```

> **NOTE**
> The ws2tcpip.h header defines GetNameInfo as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see Conventions for Function Prototypes.

Requirements

Minimum supported client	Windows 8.1, Windows Vista [desktop apps | UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps | UWP apps]
Target Platform	Windows
Header	ws2tcpip.h
Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

GetAddrInfoW

WSAGetLastError

Winsock Functions

Winsock Reference

gai_strerror

getaddrinfo

getnameinfo

sockaddr

getsourcefilter function (ws2tcpip.h)

10/22/2022 • 2 minutes to read • Edit Online

The **getsourcefilter** inline function retrieves the multicast filter state for an IPv4 or IPv6 socket.

Syntax

```
int getsourcefilter(
  [in]      SOCKET              Socket,
  [in]      ULONG               Interface,
  [in]      const SOCKADDR      *Group,
  [in]      int                 GroupLength,
  [out]     MULTICAST_MODE_TYPE *FilterMode,
  [in, out] ULONG               *SourceCount,
  [out]     SOCKADDR_STORAGE    *SourceList
);
```

Parameters

`[in] Socket`

A descriptor that identifies a multicast socket.

`[in] Interface`

The interface index of the multicast interface.

`[in] Group`

A pointer to the socket address of the multicast group.

`[in] GroupLength`

The length, in bytes, of the socket address pointed to by the *Group* parameter.

`[out] FilterMode`

A pointer to a value to receive the multicast filter mode for the multicast group address when the function returns.

`[in, out] SourceCount`

On input, a pointer to a value that indicates the maximum number of source addresses that will fit in the buffer pointed to by the *SourceList* parameter.

On output, a pointer to a value that indicates the total number of source addresses associated with the multicast filter.

`[out] SourceList`

A pointer to a buffer to receive the list of IP addresses associated with the multicast filter.

If *SourceCount* is zero on input, a **NULL** pointer may be supplied.

Return value

On success, **getsourcefilter** returns NO_ERROR (0). Any nonzero return value indicates failure and a specific error code can be retrieved by calling WSAGetLastError.

ERROR CODE	MEANING
WSAENOBUFS	Insufficient buffer space is available.
WSAENOTSOCK	The descriptor is not a socket.

Remarks

The **getsourcefilter** inline function is used to retrieve the multicast filter state for an IPv4 or IPv6 socket.

If the app does not know the size of the source list beforehand, it can make a guess (zero, for example). If upon completion, the *SourceCount* parameter holds a larger value, the operation can be repeated with a large enough buffer.

On return, the *SourceCount* parameter is always updated to be the total number of sources in the filter, while the buffer pointed to by the *SourceList* parameter will hold as many source addresses as fit, up to the minimum of the array size passed in as the original *SourceCount* value and the total number of sources in the filter.

This function is part of socket interface extensions for multicast source filters defined in RFC 3678. An app can use these functions to retrieve and set the multicast source address filters associated with a socket.

Windows Phone 8: This function is supported for Windows Phone Store apps on Windows Phone 8 and later.

Windows 8.1 and **Windows Server 2012 R2**: This function is supported for Windows Store apps on Windows 8.1, Windows Server 2012 R2, and later.

Requirements

Minimum supported client	Windows 8.1, Windows Vista [desktop apps | UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps | UWP apps]
Target Platform	Windows
Header	ws2tcpip.h
Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

MULTICAST_MODE_TYPE

SOCKADDR_STORAGE

getipv4sourcefilter

setipv4sourcefilter

setsourcefilter

inet_ntop function (ws2tcpip.h)

10/22/2022 • 4 minutes to read • Edit Online

See the inet_xtoy sample.

The **InetNtop** function converts an IPv4 or IPv6 Internet network address into a string in Internet standard format. The ANSI version of this function is **inet_ntop**.

Syntax

```
PCSTR WSAAPI inet_ntop(
  [in]  INT        Family,
  [in]  const VOID *pAddr,
  [out] PSTR       pStringBuf,
  [in]  size_t     StringBufSize
);
```

Parameters

`[in] Family`

The address family.

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. Note that the values for the AF_ address family and PF_ protocol family constants are identical (for example, **AF_INET** and **PF_INET**), so either constant can be used.

The values currently supported are **AF_INET** and **AF_INET6**.

VALUE	MEANING
AF_INET 2	The Internet Protocol version 4 (IPv4) address family. When this parameter is specified, this function returns an IPv4 address string.
AF_INET6 23	The Internet Protocol version 6 (IPv6) address family. When this parameter is specified, this function returns an IPv6 address string.

`[in] pAddr`

A pointer to the IP address in network byte to convert to a string.

When the *Family* parameter is **AF_INET**, then the *pAddr* parameter must point to an IN_ADDR structure with the IPv4 address to convert.

When the *Family* parameter is **AF_INET6**, then the *pAddr* parameter must point to an IN6_ADDR structure with the IPv6 address to convert.

`[out] pStringBuf`

A pointer to a buffer in which to store the **NULL**-terminated string representation of the IP address.

For an IPv4 address, this buffer should be large enough to hold at least 16 characters.

For an IPv6 address, this buffer should be large enough to hold at least 46 characters.

`[in] StringBufSize`

On input, the length, in characters, of the buffer pointed to by the *pStringBuf* parameter.

Return value

If no error occurs, **InetNtop** function returns a pointer to a buffer containing the string representation of IP address in standard format.

Otherwise, a value of **NULL** is returned, and a specific error code can be retrieved by calling the WSAGetLastError for extended error information.

If the function fails, the extended error code returned by WSAGetLastError can be one of the following values.

ERROR CODE	MEANING
WSAEAFNOSUPPORT	The address family specified in the *Family* parameter is not supported. This error is returned if the *Family* parameter specified was not **AF_INET** or **AF_INET6**.
ERROR_INVALID_PARAMETER	An invalid parameter was passed to the function. This error is returned if a **NULL** pointer is passed in the *pStringBuf* or the *StringBufSize* parameter is zero. This error is also returned if the length of the buffer pointed to by the *pStringBuf* parameter is not large enough to receive the string representation of the IP address.

Remarks

The **InetNtop** function is supported on Windows Vista and later.

The **InetNtop** function provides a protocol-independent address-to-string translation. The **InetNtop** function takes an Internet address structure specified by the *pAddr* parameter and returns a **NULL**-terminated string that represents the IP address. While the inet_ntoa function works only with IPv4 addresses, the **InetNtop** function works with either IPv4 or IPv6 addresses.

The ANSI version of this function is **inet_ntop** as defined in RFC 2553. For more information, see RFC 2553 available at the IETF website.

The **InetNtop** function does not require that the Windows Sockets DLL be loaded to perform IP address to string conversion.

If the *Family* parameter specified is **AF_INET**, then the *pAddr* parameter must point to an IN_ADDR structure with the IPv4 address to convert. The address string returned in the buffer pointed to by the *pStringBuf* parameter is in dotted-decimal notation as in "192.168.16.0", an example of an IPv4 address in dotted-decimal notation.

If the *Family* parameter specified is **AF_INET6**, then the *pAddr* parameter must point to an IN6_ADDR structure with the IPv6 address to convert. The address string returned in the buffer pointed to by the *pStringBuf* parameter is in Internet standard format. The basic string representation consists of 8 hexadecimal numbers separated by colons. A string of consecutive zero numbers is replaced with a double-colon. There can only be one double-colon in the string representation of the IPv6 address. The last 32 bits are represented in IPv4-style dotted-octet notation if the address is a IPv4-compatible address.

If the length of the buffer pointed to by the *pStringBuf* parameter is not large enough to receive the string representation of the IP address, **InetNtop** returns ERROR_INVALID_PARAMETER.

When UNICODE or _UNICODE is defined, **InetNtop** is defined to **InetNtopW**, the Unicode version of this function. The *pStringBuf* parameter is defined to the **PSTR** data type.

When UNICODE or _UNICODE is not defined, **InetNtop** is defined to **InetNtopA**, the ANSI version of this function. The ANSI version of this function is always defined as **inet_ntop**. The *pStringBuf* parameter is defined to the **PWSTR** data type.

The IN_ADDR structure is defined in the *Inaddr.h* header file.

The IN6_ADDR structure is defined in the *In6addr.h* header file.

On Windows Vista and later, the RtlIpv4AddressToString and RtlIpv4AddressToStringEx functions can be used to convert an IPv4 address represented as an IN_ADDR structure to a string representation of an IPv4 address in Internet standard dotted-decimal notation. On Windows Vista and later, the RtlIpv6AddressToString and RtlIpv6AddressToStringEx functions can be used to convert an IPv6 address represented as an IN6_ADDR structure to a string representation of an IPv6 address. The **RtlIpv6AddressToStringEx** function is more flexible since it also converts an IPv6 address, scope ID, and port to a IPv6 string in standard format.

Windows 8.1 and **Windows Server 2012 R2**: The **InetNtopW** function is supported for Windows Store apps on Windows 8.1, Windows Server 2012 R2, and later.

Requirements

Minimum supported client	Windows 8.1, Windows Vista [desktop apps | UWP apps]
Minimum supported server	Windows Server 2008 [desktop apps | UWP apps]
Target Platform	Windows
Header	ws2tcpip.h
Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

IN6_ADDR

IN_ADDR

InetPton

RtlIpv4AddressToString

RtlIpv4AddressToStringEx

RtlIpv4StringToAddress

RtlIpv4StringToAddressEx

RtlIpv6AddressToString

RtlIpv6AddressToStringEx

RtlIpv6StringToAddress

RtlIpv6StringToAddressEx

inet_addr

inet_ntoa

inet_xtoy sample

inet_pton function (ws2tcpip.h)

10/22/2022 • 4 minutes to read • Edit Online

The **InetPton** function 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 **inet_pton**.

Syntax

```
INT WSAAPI inet_pton(
  [in]  INT   Family,
  [in]  PCSTR pszAddrString,
  [out] PVOID pAddrBuf
);
```

Parameters

`[in] Family`

The address family.

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. Note that the values for the AF_ address family and PF_ protocol family constants are identical (for example, **AF_INET** and **PF_INET**), so either constant can be used.

The values currently supported are **AF_INET** and **AF_INET6**.

VALUE	MEANING
AF_INET 2	The Internet Protocol version 4 (IPv4) address family. When this parameter is specified, the *pszAddrString* parameter must point to a text representation of an IPv4 address and the *pAddrBuf* parameter returns a pointer to an IN_ADDR structure that represents the IPv4 address.
AF_INET6 23	The Internet Protocol version 6 (IPv6) address family. When this parameter is specified, the *pszAddrString* parameter must point to a text representation of an IPv6 address and the *pAddrBuf* parameter returns a pointer to an IN6_ADDR structure that represents the IPv6 address.

`[in] pszAddrString`

A pointer to the **NULL**-terminated string that contains the text representation of the IP address to convert to numeric binary form.

When the *Family* parameter is **AF_INET**, then the *pszAddrString* parameter must point to a text representation of an IPv4 address in standard dotted-decimal notation.

When the *Family* parameter is **AF_INET6**, then the *pszAddrString* parameter must point to a text representation of an IPv6 address in standard notation.

`[out] pAddrBuf`

A pointer to a buffer in which to store the numeric binary representation of the IP address. The IP address is returned in network byte order.

When the *Family* parameter is **AF_INET**, this buffer should be large enough to hold an IN_ADDR structure.

When the *Family* parameter is **AF_INET6**, this buffer should be large enough to hold an IN6_ADDR structure.

Return value

If no error occurs, the **InetPton** function returns a value of 1 and the buffer pointed to by the *pAddrBuf* parameter contains the binary numeric IP address in network byte order.

The **InetPton** function returns a value of 0 if the *pAddrBuf* parameter points to a string that is not a valid IPv4 dotted-decimal string or a valid IPv6 address string. Otherwise, a value of -1 is returned, and a specific error code can be retrieved by calling the WSAGetLastError for extended error information.

If the function has an error, the extended error code returned by WSAGetLastError can be one of the following values.

ERROR CODE	MEANING
WSAEAFNOSUPPORT	The address family specified in the *Family* parameter is not supported. This error is returned if the *Family* parameter specified was not **AF_INET** or **AF_INET6**.
WSAEFAULT	The *pszAddrString* or *pAddrBuf* parameters are **NULL** or are not part of the user address space.

Remarks

The **InetPton** function is supported on Windows Vista and later.

The **InetPton** function provides a protocol-independent conversion of an Internet network address in its standard text presentation form into its numeric binary form. The **InetPton** function takes a text representation of an Internet address pointed to by the *pszAddrString* parameter and returns a pointer to the numeric binary IP address in the *pAddrBuf* parameter. While the inet_addr function works only with IPv4 address strings, the **InetPton** function works with either IPv4 or IPv6 address strings.

The ANSI version of this function is **inet_pton** as defined in RFC 2553. For more information, see RFC 2553 available at the IETF website.

The **InetPton** function does not require that the Windows Sockets DLL be loaded to perform conversion of a text string that represents an IP address to a numeric binary IP address.

If the *Family* parameter specified is **AF_INET**, then the *pszAddrString* parameter must point a text string of an IPv4 address in dotted-decimal notation as in "192.168.16.0", an example of an IPv4 address in dotted-decimal notation.

If the *Family* parameter specified is **AF_INET6**, then the *pszAddrString* parameter must point a text string of an IPv6 address in Internet standard format. The basic string representation consists of 8 hexadecimal numbers separated by colons. A string of consecutive zero numbers may be replaced with a double-colon. There can only be one double-colon in the string representation of the IPv6 address. The last 32 bits may be represented in IPv4-style dotted-octet notation if the address is a IPv4-compatible address.

When UNICODE or _UNICODE is defined, **InetPton** is defined to **InetPtonW**, the Unicode version of this function. The *pszAddrString* parameter is defined to the **PCWSTR** data type.

When UNICODE or _UNICODE is not defined, **InetPton** is defined to **InetPtonA**, the ANSI version of this function. The ANSI version of this function is always defined as inet_pton. The *pszAddrString* parameter is defined to the **PCSTR** data type.

The IN_ADDR structure is defined in the *Inaddr.h* header file.

The IN6_ADDR structure is defined in the *In6addr.h* header file.

On Windows Vista and later, the RtlIpv4StringToAddress and RtlIpv4StringToAddressEx functions can be used to convert a text representation of an IPv4 address in Internet standard dotted-decimal notation to a numeric binary address represented as an IN_ADDR structure. On Windows Vista and later, the RtlIpv6StringToAddress and RtlIpv6StringToAddressEx functions can be used to convert a string representation of an IPv6 address to a numeric binary IPv6 address represented as an IN6_ADDR structure. The **RtlIpv6StringToAddressEx** function is more flexible since it also converts a string representation of an IPv6 address that can include a scope ID and port in standard notation to a numeric binary form.

Windows 8.1 and **Windows Server 2012 R2**: The **InetPtonW** function is supported for Windows Store apps on Windows 8.1, Windows Server 2012 R2, and later.

Requirements

Minimum supported client	Windows 8.1, Windows Vista [desktop apps | UWP apps]
Minimum supported server	Windows Server 2008 [desktop apps | UWP apps]
Target Platform	Windows
Header	ws2tcpip.h
Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

IN6_ADDR

IN_ADDR

InetNtop

RtlIpv4AddressToString

RtlIpv4AddressToStringEx

RtlIpv4StringToAddress

RtlIpv4StringToAddressEx

RtlIpv6AddressToString

RtlIpv6AddressToStringEx

RtlIpv6StringToAddress

RtlIpv6StringToAddressEx

inet_addr

inet_ntoa

InetNtopW function (ws2tcpip.h)

10/22/2022 • 4 minutes to read • Edit Online

The **InetNtop** function converts an IPv4 or IPv6 Internet network address into a string in Internet standard format. The ANSI version of this function is **inet_ntop**.

Syntax

```
PCWSTR WSAAPI InetNtopW(
  [in]  INT        Family,
  [in]  const VOID *pAddr,
  [out] PWSTR      pStringBuf,
  [in]  size_t     StringBufSize
);
```

Parameters

`[in] Family`

The address family.

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. Note that the values for the AF_ address family and PF_ protocol family constants are identical (for example, **AF_INET** and **PF_INET**), so either constant can be used.

The values currently supported are **AF_INET** and **AF_INET6**.

VALUE	MEANING
AF_INET 2	The Internet Protocol version 4 (IPv4) address family. When this parameter is specified, this function returns an IPv4 address string.
AF_INET6 23	The Internet Protocol version 6 (IPv6) address family. When this parameter is specified, this function returns an IPv6 address string.

`[in] pAddr`

A pointer to the IP address in network byte to convert to a string.

When the *Family* parameter is **AF_INET**, then the *pAddr* parameter must point to an IN_ADDR structure with the IPv4 address to convert.

When the *Family* parameter is **AF_INET6**, then the *pAddr* parameter must point to an IN6_ADDR structure with the IPv6 address to convert.

`[out] pStringBuf`

A pointer to a buffer in which to store the **NULL**-terminated string representation of the IP address.

For an IPv4 address, this buffer should be large enough to hold at least 16 characters.

For an IPv6 address, this buffer should be large enough to hold at least 46 characters.

`[in] StringBufSize`

On input, the length, in characters, of the buffer pointed to by the *pStringBuf* parameter.

Return value

If no error occurs, **InetNtop** function returns a pointer to a buffer containing the string representation of IP address in standard format.

Otherwise, a value of **NULL** is returned, and a specific error code can be retrieved by calling the WSAGetLastError for extended error information.

If the function fails, the extended error code returned by WSAGetLastError can be one of the following values.

ERROR CODE	MEANING
WSAEAFNOSUPPORT	The address family specified in the *Family* parameter is not supported. This error is returned if the *Family* parameter specified was not **AF_INET** or **AF_INET6**.
ERROR_INVALID_PARAMETER	An invalid parameter was passed to the function. This error is returned if a **NULL** pointer is passed in the *pStringBuf* or the *StringBufSize* parameter is zero. This error is also returned if the length of the buffer pointed to by the *pStringBuf* parameter is not large enough to receive the string representation of the IP address.

Remarks

The **InetNtop** function is supported on Windows Vista and later.

The **InetNtop** function provides a protocol-independent address-to-string translation. The **InetNtop** function takes an Internet address structure specified by the *pAddr* parameter and returns a **NULL**-terminated string that represents the IP address. While the inet_ntoa function works only with IPv4 addresses, the **InetNtop** function works with either IPv4 or IPv6 addresses.

The ANSI version of this function is **inet_ntop** as defined in RFC 2553. For more information, see RFC 2553 available at the IETF website.

The **InetNtop** function does not require that the Windows Sockets DLL be loaded to perform IP address to string conversion.

If the *Family* parameter specified is **AF_INET**, then the *pAddr* parameter must point to an IN_ADDR structure with the IPv4 address to convert. The address string returned in the buffer pointed to by the *pStringBuf* parameter is in dotted-decimal notation as in "192.168.16.0", an example of an IPv4 address in dotted-decimal notation.

If the *Family* parameter specified is **AF_INET6**, then the *pAddr* parameter must point to an IN6_ADDR structure with the IPv6 address to convert. The address string returned in the buffer pointed to by the *pStringBuf* parameter is in Internet standard format. The basic string representation consists of 8 hexadecimal numbers separated by colons. A string of consecutive zero numbers is replaced with a double-colon. There can only be one double-colon in the string representation of the IPv6 address. The last 32 bits are represented in IPv4-style dotted-octet notation if the address is a IPv4-compatible address.

If the length of the buffer pointed to by the *pStringBuf* parameter is not large enough to receive the string representation of the IP address, **InetNtop** returns ERROR_INVALID_PARAMETER.

When UNICODE or _UNICODE is defined, **InetNtop** is defined to **InetNtopW**, the Unicode version of this function. The *pStringBuf* parameter is defined to the **PSTR** data type.

When UNICODE or _UNICODE is not defined, **InetNtop** is defined to **InetNtopA**, the ANSI version of this function. The ANSI version of this function is always defined as **inet_ntop**. The *pStringBuf* parameter is defined to the **PWSTR** data type.

The IN_ADDR structure is defined in the *Inaddr.h* header file.

The IN6_ADDR structure is defined in the *In6addr.h* header file.

On Windows Vista and later, the RtlIpv4AddressToString and RtlIpv4AddressToStringEx functions can be used to convert an IPv4 address represented as an IN_ADDR structure to a string representation of an IPv4 address in Internet standard dotted-decimal notation. On Windows Vista and later, the RtlIpv6AddressToString and RtlIpv6AddressToStringEx functions can be used to convert an IPv6 address represented as an IN6_ADDR structure to a string representation of an IPv6 address. The **RtlIpv6AddressToStringEx** function is more flexible since it also converts an IPv6 address, scope ID, and port to a IPv6 string in standard format.

Windows 8.1 and **Windows Server 2012 R2**: The **InetNtopW** function is supported for Windows Store apps on Windows 8.1, Windows Server 2012 R2, and later.

Requirements

Minimum supported client	Windows 8.1, Windows Vista [desktop apps | UWP apps]
Minimum supported server	Windows Server 2008 [desktop apps | UWP apps]
Target Platform	Windows
Header	ws2tcpip.h
Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

IN6_ADDR

IN_ADDR

InetPton

RtlIpv4AddressToString

RtlIpv4AddressToStringEx

RtlIpv4StringToAddress

RtlIpv4StringToAddressEx

RtlIpv6AddressToString

RtlIpv6AddressToStringEx

RtlIpv6StringToAddress

RtlIpv6StringToAddressEx

inet_addr

inet_ntoa

InetPtonW function (ws2tcpip.h)

10/22/2022 • 4 minutes to read • Edit Online

The **InetPton** function 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 **inet_pton**.

Syntax

```
INT WSAAPI InetPtonW(
  [in]  INT    Family,
  [in]  PCWSTR pszAddrString,
  [out] PVOID  pAddrBuf
);
```

Parameters

`[in] Family`

The address family.

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. Note that the values for the AF_ address family and PF_ protocol family constants are identical (for example, **AF_INET** and **PF_INET**), so either constant can be used.

The values currently supported are **AF_INET** and **AF_INET6**.

VALUE	MEANING
AF_INET 2	The Internet Protocol version 4 (IPv4) address family. When this parameter is specified, the *pszAddrString* parameter must point to a text representation of an IPv4 address and the *pAddrBuf* parameter returns a pointer to an IN_ADDR structure that represents the IPv4 address.
AF_INET6 23	The Internet Protocol version 6 (IPv6) address family. When this parameter is specified, the *pszAddrString* parameter must point to a text representation of an IPv6 address and the *pAddrBuf* parameter returns a pointer to an IN6_ADDR structure that represents the IPv6 address.

`[in] pszAddrString`

A pointer to the **NULL**-terminated string that contains the text representation of the IP address to convert to numeric binary form.

When the *Family* parameter is **AF_INET**, then the *pszAddrString* parameter must point to a text representation of an IPv4 address in standard dotted-decimal notation.

When the *Family* parameter is **AF_INET6**, then the *pszAddrString* parameter must point to a text representation of an IPv6 address in standard notation.

`[out] pAddrBuf`

A pointer to a buffer in which to store the numeric binary representation of the IP address. The IP address is returned in network byte order.

When the *Family* parameter is **AF_INET**, this buffer should be large enough to hold an IN_ADDR structure.

When the *Family* parameter is **AF_INET6**, this buffer should be large enough to hold an IN6_ADDR structure.

Return value

If no error occurs, the **InetPton** function returns a value of 1 and the buffer pointed to by the *pAddrBuf* parameter contains the binary numeric IP address in network byte order.

The **InetPton** function returns a value of 0 if the *pAddrBuf* parameter points to a string that is not a valid IPv4 dotted-decimal string or a valid IPv6 address string. Otherwise, a value of -1 is returned, and a specific error code can be retrieved by calling the WSAGetLastError for extended error information.

If the function has an error, the extended error code returned by WSAGetLastError can be one of the following values.

ERROR CODE	MEANING
WSAEAFNOSUPPORT	The address family specified in the *Family* parameter is not supported. This error is returned if the *Family* parameter specified was not **AF_INET** or **AF_INET6**.
WSAEFAULT	The *pszAddrString* or *pAddrBuf* parameters are **NULL** or are not part of the user address space.

Remarks

The **InetPton** function is supported on Windows Vista and later.

The **InetPton** function provides a protocol-independent conversion of an Internet network address in its standard text presentation form into its numeric binary form. The **InetPton** function takes a text representation of an Internet address pointed to by the *pszAddrString* parameter and returns a pointer to the numeric binary IP address in the *pAddrBuf* parameter. While the inet_addr function works only with IPv4 address strings, the **InetPton** function works with either IPv4 or IPv6 address strings.

The ANSI version of this function is **inet_pton** as defined in RFC 2553. For more information, see RFC 2553 available at the IETF website.

The **InetPton** function does not require that the Windows Sockets DLL be loaded to perform conversion of a text string that represents an IP address to a numeric binary IP address.

If the *Family* parameter specified is **AF_INET**, then the *pszAddrString* parameter must point a text string of an IPv4 address in dotted-decimal notation as in "192.168.16.0", an example of an IPv4 address in dotted-decimal notation.

If the *Family* parameter specified is **AF_INET6**, then the *pszAddrString* parameter must point a text string of an IPv6 address in Internet standard format. The basic string representation consists of 8 hexadecimal numbers separated by colons. A string of consecutive zero numbers may be replaced with a double-colon. There can only be one double-colon in the string representation of the IPv6 address. The last 32 bits may be represented in IPv4-style dotted-octet notation if the address is a IPv4-compatible address.

When UNICODE or _UNICODE is defined, **InetPton** is defined to **InetPtonW**, the Unicode version of this function. The *pszAddrString* parameter is defined to the **PCWSTR** data type.

When UNICODE or _UNICODE is not defined, **InetPton** is defined to **InetPtonA**, the ANSI version of this function. The ANSI version of this function is always defined as inet_pton. The *pszAddrString* parameter is defined to the **PCSTR** data type.

The IN_ADDR structure is defined in the *Inaddr.h* header file.

The IN6_ADDR structure is defined in the *In6addr.h* header file.

On Windows Vista and later, the RtlIpv4StringToAddress and RtlIpv4StringToAddressEx functions can be used to convert a text representation of an IPv4 address in Internet standard dotted-decimal notation to a numeric binary address represented as an IN_ADDR structure. On Windows Vista and later, the RtlIpv6StringToAddress and RtlIpv6StringToAddressEx functions can be used to convert a string representation of an IPv6 address to a numeric binary IPv6 address represented as an IN6_ADDR structure. The **RtlIpv6StringToAddressEx** function is more flexible since it also converts a string representation of an IPv6 address that can include a scope ID and port in standard notation to a numeric binary form.

Windows 8.1 and **Windows Server 2012 R2**: The **InetPtonW** function is supported for Windows Store apps on Windows 8.1, Windows Server 2012 R2, and later.

Requirements

Minimum supported client	Windows 8.1, Windows Vista [desktop apps | UWP apps]
Minimum supported server	Windows Server 2008 [desktop apps | UWP apps]
Target Platform	Windows
Header	ws2tcpip.h
Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

IN6_ADDR

IN_ADDR

InetNtop

RtlIpv4AddressToString

RtlIpv4AddressToStringEx

RtlIpv4StringToAddress

RtlIpv4StringToAddressEx

RtlIpv6AddressToString

RtlIpv6AddressToStringEx

RtlIpv6StringToAddress

RtlIpv6StringToAddressEx

inet_addr

inet_ntoa

SetAddrInfoExA function (ws2tcpip.h)

10/22/2022 • 5 minutes to read • Edit Online

The **SetAddrInfoEx** function registers or deregisters a name, a service name, and associated addresses with a specific namespace provider.

Syntax

```
INT WSAAPI SetAddrInfoExA(
  [in]            PCSTR                              pName,
  [in]            PCSTR                              pServiceName,
  [in, out]       SOCKET_ADDRESS                     *pAddresses,
  [in]            DWORD                              dwAddressCount,
  [in, optional]  LPBLOB                             lpBlob,
  [in]            DWORD                              dwFlags,
  [in]            DWORD                              dwNameSpace,
  [in, optional]  LPGUID                             lpNspId,
  [in, optional]  timeval                            *timeout,
  [in, optional]  LPOVERLAPPED                       lpOverlapped,
  [in, optional]  LPLOOKUPSERVICE_COMPLETION_ROUTINE lpCompletionRoutine,
  [out, optional] LPHANDLE                           lpNameHandle
);
```

Parameters

`[in] pName`

A pointer to a **NULL**-terminated string containing a name under which addresses are to be registered or deregistered. The interpretation of this parameter specific to the namespace provider.

`[in] pServiceName`

A pointer to an optional **NULL**-terminated string that contains the service name associated with the name being registered. The interpretation of this parameter is specific to the namespace provider.

`[in, out] pAddresses`

A pointer to an optional list of addresses to register with the namespace provider.

`[in] dwAddressCount`

The number of addresses passed in *pAddresses* parameter. If this parameter is zero, the *pName* parameter is deregistered from the namespace provider.

`[in, optional] lpBlob`

An optional pointer to data that is used to set provider-specific namespace information that is associated with the *pName* parameter beyond a list of addresses. Any information that cannot be passed in the *pAddresses* parameter can be passed in the *lpBlob* parameter. The format of this information is specific to the namespace provider.

`[in] dwFlags`

A set of flags controlling how the *pName* and *pServiceName* parameters are to be registered with the namespace provider. The interpretation of this information is specific to the namespace provider.

`[in] dwNameSpace`

A namespace identifier that determines which namespace provider to register this information with. Passing a specific namespace identifier will result in registering this information only with the namespace providers that support the specified namespace. Specifying NS_ALL will result in registering the information with all installed and active namespace providers.

Options for the *dwNameSpace* parameter are listed in the *Winsock2.h* include file. Several namespace providers are included with Windows Vista and later. Other namespace providers can be installed, so the following possible values are only those commonly available. Many others are possible.

VALUE	MEANING
NS_ALL	All installed and active namespaces.
NS_BTH	The Bluetooth namespace. This namespace identifier is supported on Windows Vista and later.
NS_DNS	The domain name system (DNS) namespace.
NS_EMAIL	The email namespace. This namespace identifier is supported on Windows Vista and later.
NS_NLA	The network location awareness (NLA) namespace. This namespace identifier is supported on Windows XP and later.
NS_PNRPNAME	The peer-to-peer namespace for a specific peer name. This namespace identifier is supported on Windows Vista and later.
NS_PNRPCLOUD	The peer-to-peer namespace for a collection of peer names. This namespace identifier is supported on Windows Vista and later.

`[in, optional] lpNspId`

A pointer to an optional GUID of a specific namespace provider to register this information with in the case where multiple namespace providers are registered under a single namespace such as NS_DNS. Passing the GUID for a specific namespace provider will result in the information being registered with only the specified namespace provider. The WSAEnumNameSpaceProviders function can be called to retrieve the GUID for a namespace provider.

`[in, optional] timeout`

An optional parameter indicating the time, in milliseconds, to wait for a response from the namespace provider before aborting the call. This parameter is currently reserved and must be set to **NULL** since a *timeout* option is not supported.

`[in, optional] lpOverlapped`

An optional pointer to an overlapped structure used for asynchronous operation. This parameter is currently reserved and must be set to **NULL** since asynchronous operations are not supported.

`[in, optional] lpCompletionRoutine`

An optional pointer to a function to be invoked upon successful completion for asynchronous operations. This parameter is currently reserved and must be set to **NULL** since asynchronous operations are not supported.

`[out, optional] lpNameHandle`

An optional pointer used only for asynchronous operations. This parameter is currently reserved and must be set to **NULL** since asynchronous operations are not supported.

Return value

On success, **SetAddrInfoEx** returns NO_ERROR (0). Failure returns a nonzero Windows Sockets error code, as found in the Windows Sockets Error Codes.

ERROR CODE	MEANING
WSANOTINITIALISED	A successful WSAStartup call must occur before using this function.
WSATRY_AGAIN	A temporary failure in name resolution occurred.
WSAEINVAL	An invalid parameter was provided. This error is returned if any of the reserved parameters are not **NULL**.
WSAENOBUFS	Insufficient buffer space is available.
WSANO_RECOVERY	A nonrecoverable failure in name resolution occurred.
WSA_NOT_ENOUGH_MEMORY	A memory allocation failure occurred.

Remarks

The **SetAddrInfoEx** function provides a protocol-independent method to register or deregister a name and one or more addresses with a namespace provider. The NS_EMAIL namespace provider in Windows Vista and later supports registration and deregistration of addresses. The default NS_DNS, NS_PNRPNAME, and NS_PNRPNAME namespace providers do not currently support name registration.

If the **SetAddrInfoEx** function is called with NS_ALL set as the *dwNameSpace* parameter and the *lpNspId* parameter unspecified, then **SetAddrInfoEx** will attempt to register or deregister the name and associated addresses with all installed and active namespaces. The **SetAddrInfoEx** function will return success if any of the namespace providers successfully registered or deregistered the name, but there will not be any indication of which namespace provider succeeded or which ones failed the request.

When **UNICODE** or **_UNICODE** is defined, **SetAddrInfoEx** is defined to SetAddrInfoExW, the Unicode version of this function. The string parameters are defined to the **PWSTR** data type.

When **UNICODE** or **_UNICODE** is not defined, **SetAddrInfoEx** is defined to SetAddrInfoExA, the ANSI version of this function. The string parameters are of the **PCSTR** data type.

Information that is registered with a namespace provider can be returned by calling the GetAddrInfoEx, getaddrinfo, or GetAddrInfoW functions. The **GetAddrInfoEx** function is an enhanced version of the

getaddrinfo and **GetAddrInfoW** functions.

On Windows Vista and later, when **SetAddrInfoEx** is called from a service, if the operation is the result of a user process calling the service, then the service should impersonate the user. This is to allow security and routing compartments to be properly enforced.

Windows 8.1 and **Windows Server 2012 R2**: The **SetAddrInfoExW** function is supported for Windows Store apps on Windows 8.1, Windows Server 2012 R2, and later.

> **NOTE**
> The ws2tcpip.h header defines SetAddrInfoEx as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see Conventions for Function Prototypes.

Requirements

Minimum supported client	Windows 8.1, Windows Vista [desktop apps | UWP apps]
Minimum supported server	Windows Server 2008 [desktop apps | UWP apps]
Target Platform	Windows
Header	ws2tcpip.h
Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

GetAddrInfoEx

GetAddrInfoW

WSAEnumNameSpaceProviders

WSAGetLastError

Windows Sockets Error Codes

getaddrinfo

SetAddrInfoExW function (ws2tcpip.h)

10/22/2022 • 5 minutes to read • Edit Online

The **SetAddrInfoEx** function registers or deregisters a name, a service name, and associated addresses with a specific namespace provider.

Syntax

```
INT WSAAPI SetAddrInfoExW(
  [in]            PCWSTR                            pName,
  [in]            PCWSTR                            pServiceName,
  [in, out]       SOCKET_ADDRESS                    *pAddresses,
  [in]            DWORD                             dwAddressCount,
  [in, optional]  LPBLOB                            lpBlob,
  [in]            DWORD                             dwFlags,
  [in]            DWORD                             dwNameSpace,
  [in, optional]  LPGUID                            lpNspId,
  [in, optional]  timeval                           *timeout,
  [in, optional]  LPOVERLAPPED                      lpOverlapped,
  [in, optional]  LPLOOKUPSERVICE_COMPLETION_ROUTINE lpCompletionRoutine,
  [out, optional] LPHANDLE                          lpNameHandle
);
```

Parameters

`[in] pName`

A pointer to a **NULL**-terminated string containing a name under which addresses are to be registered or deregistered. The interpretation of this parameter specific to the namespace provider.

`[in] pServiceName`

A pointer to an optional **NULL**-terminated string that contains the service name associated with the name being registered. The interpretation of this parameter is specific to the namespace provider.

`[in, out] pAddresses`

A pointer to an optional list of addresses to register with the namespace provider.

`[in] dwAddressCount`

The number of addresses passed in *pAddresses* parameter. If this parameter is zero, the *pName* parameter is deregistered from the namespace provider.

`[in, optional] lpBlob`

An optional pointer to data that is used to set provider-specific namespace information that is associated with the *pName* parameter beyond a list of addresses. Any information that cannot be passed in the *pAddresses* parameter can be passed in the *lpBlob* parameter. The format of this information is specific to the namespace provider.

`[in] dwFlags`

A set of flags controlling how the *pName* and *pServiceName* parameters are to be registered with the namespace provider. The interpretation of this information is specific to the namespace provider.

`[in] dwNameSpace`

A namespace identifier that determines which namespace provider to register this information with. Passing a specific namespace identifier will result in registering this information only with the namespace providers that support the specified namespace. Specifying NS_ALL will result in registering the information with all installed and active namespace providers.

Options for the *dwNameSpace* parameter are listed in the *Winsock2.h* include file. Several namespace providers are included with Windows Vista and later. Other namespace providers can be installed, so the following possible values are only those commonly available. Many others are possible.

VALUE	MEANING
NS_ALL	All installed and active namespaces.
NS_BTH	The Bluetooth namespace. This namespace identifier is supported on Windows Vista and later.
NS_DNS	The domain name system (DNS) namespace.
NS_EMAIL	The email namespace. This namespace identifier is supported on Windows Vista and later.
NS_NLA	The network location awareness (NLA) namespace. This namespace identifier is supported on Windows XP and later.
NS_PNRPNAME	The peer-to-peer namespace for a specific peer name. This namespace identifier is supported on Windows Vista and later.
NS_PNRPCLOUD	The peer-to-peer namespace for a collection of peer names. This namespace identifier is supported on Windows Vista and later.

`[in, optional] lpNspId`

A pointer to an optional GUID of a specific namespace provider to register this information with in the case where multiple namespace providers are registered under a single namespace such as NS_DNS. Passing the GUID for a specific namespace provider will result in the information being registered with only the specified namespace provider. The WSAEnumNameSpaceProviders function can be called to retrieve the GUID for a namespace provider.

`[in, optional] timeout`

An optional parameter indicating the time, in milliseconds, to wait for a response from the namespace provider before aborting the call. This parameter is currently reserved and must be set to **NULL** since a *timeout* option is not supported.

`[in, optional] lpOverlapped`

An optional pointer to an overlapped structure used for asynchronous operation. This parameter is currently reserved and must be set to **NULL** since asynchronous operations are not supported.

`[in, optional] lpCompletionRoutine`

An optional pointer to a function to be invoked upon successful completion for asynchronous operations. This parameter is currently reserved and must be set to **NULL** since asynchronous operations are not supported.

`[out, optional] lpNameHandle`

An optional pointer used only for asynchronous operations. This parameter is currently reserved and must be set to **NULL** since asynchronous operations are not supported.

Return value

On success, **SetAddrInfoEx** returns NO_ERROR (0). Failure returns a nonzero Windows Sockets error code, as found in the Windows Sockets Error Codes.

ERROR CODE	MEANING
WSANOTINITIALISED	A successful WSAStartup call must occur before using this function.
WSATRY_AGAIN	A temporary failure in name resolution occurred.
WSAEINVAL	An invalid parameter was provided. This error is returned if any of the reserved parameters are not **NULL**.
WSAENOBUFS	Insufficient buffer space is available.
WSANO_RECOVERY	A nonrecoverable failure in name resolution occurred.
WSA_NOT_ENOUGH_MEMORY	A memory allocation failure occurred.

Remarks

The **SetAddrInfoEx** function provides a protocol-independent method to register or deregister a name and one or more addresses with a namespace provider. The NS_EMAIL namespace provider in Windows Vista and later supports registration and deregistration of addresses. The default NS_DNS, NS_PNRPNAME, and NS_PNRPNAME namespace providers do not currently support name registration.

If the **SetAddrInfoEx** function is called with NS_ALL set as the *dwNameSpace* parameter and the *lpNspId* parameter unspecified, then **SetAddrInfoEx** will attempt to register or deregister the name and associated addresses with all installed and active namespaces. The **SetAddrInfoEx** function will return success if any of the namespace providers successfully registered or deregistered the name, but there will not be any indication of which namespace provider succeeded or which ones failed the request.

When **UNICODE** or **_UNICODE** is defined, **SetAddrInfoEx** is defined to SetAddrInfoExW, the Unicode version of this function. The string parameters are defined to the **PWSTR** data type.

When **UNICODE** or **_UNICODE** is not defined, **SetAddrInfoEx** is defined to SetAddrInfoExA, the ANSI version of this function. The string parameters are of the **PCSTR** data type.

Information that is registered with a namespace provider can be returned by calling the GetAddrInfoEx, getaddrinfo, or GetAddrInfoW functions. The **GetAddrInfoEx** function is an enhanced version of the

getaddrinfo and **GetAddrInfoW** functions.

On Windows Vista and later, when **SetAddrInfoEx** is called from a service, if the operation is the result of a user process calling the service, then the service should impersonate the user. This is to allow security and routing compartments to be properly enforced.

Windows 8.1 and **Windows Server 2012 R2**: The **SetAddrInfoExW** function is supported for Windows Store apps on Windows 8.1, Windows Server 2012 R2, and later.

> **NOTE**
> The ws2tcpip.h header defines SetAddrInfoEx as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see Conventions for Function Prototypes.

Requirements

Minimum supported client	Windows 8.1, Windows Vista [desktop apps | UWP apps]
Minimum supported server	Windows Server 2008 [desktop apps | UWP apps]
Target Platform	Windows
Header	ws2tcpip.h
Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

GetAddrInfoEx

GetAddrInfoW

WSAEnumNameSpaceProviders

WSAGetLastError

Windows Sockets Error Codes

getaddrinfo

setipv4sourcefilter function (ws2tcpip.h)

10/22/2022 • 2 minutes to read • Edit Online

The **setipv4sourcefilter** inline function sets the multicast filter state for an IPv4 socket.

Syntax

```
int setipv4sourcefilter(
  [in] SOCKET              Socket,
  [in] IN_ADDR             Interface,
  [in] IN_ADDR             Group,
  [in] MULTICAST_MODE_TYPE FilterMode,
  [in] ULONG               SourceCount,
  [in] const IN_ADDR       *SourceList
);
```

Parameters

`[in] Socket`

A descriptor that identifies a multicast socket.

`[in] Interface`

The local IPv4 address of the interface or the interface index on which the multicast group should be joined or dropped.

This value is in network byte order. If this member specifies an IPv4 address of 0.0.0.0, the default IPv4 multicast interface is used.

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).

To use an interface index of 1 would be the same as an IP address of 0.0.0.1.

`[in] Group`

The IPv4 address of the multicast group.

`[in] FilterMode`

The multicast filter mode for multicast group address.

`[in] SourceCount`

The number of source addresses in the buffer pointed to by the *SourceList* parameter.

`[in] SourceList`

A pointer to a buffer with the IP addresses to associate with the multicast filter.

Return value

On success, **setipv4sourcefilter** returns NO_ERROR (0). Any nonzero return value indicates failure and a specific error code can be retrieved by calling WSAGetLastError.

ERROR CODE	MEANING
WSAENOBUFS	Insufficient buffer space is available.
WSAENOTSOCK	The descriptor is not a socket.

Remarks

The **setipv4sourcefilter** inline function is used to set the multicast filter state for an IPv4 socket.

This function is part of socket interface extensions for multicast source filters defined in RFC 3678. An app can use these functions to retrieve and set the multicast source address filters associated with a socket.

Windows Phone 8: This function is supported for Windows Phone Store apps on Windows Phone 8 and later.

Windows 8.1 and **Windows Server 2012 R2**: This function is supported for Windows Store apps on Windows 8.1, Windows Server 2012 R2, and later.

Requirements

Minimum supported client	Windows 8.1, Windows Vista [desktop apps | UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps | UWP apps]
Target Platform	Windows
Header	ws2tcpip.h
Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

MULTICAST_MODE_TYPE

getipv4sourcefilter

getsourcefilter

in_addr

setsourcefilter

setsourcefilter function (ws2tcpip.h)

10/22/2022 • 2 minutes to read • Edit Online

The **setsourcefilter** inline function sets the multicast filter state for an IPv4 or IPv6 socket.

Syntax

```
int setsourcefilter(
  [in] SOCKET                 Socket,
  [in] ULONG                  Interface,
  [in] const SOCKADDR         *Group,
  [in] int                    GroupLength,
  [in] MULTICAST_MODE_TYPE    FilterMode,
  [in] ULONG                  SourceCount,
  [in] const SOCKADDR_STORAGE *SourceList
);
```

Parameters

`[in] Socket`

A descriptor that identifies a multicast socket.

`[in] Interface`

The interface index of the multicast interface.

`[in] Group`

A pointer to the socket address of the multicast group.

`[in] GroupLength`

The length, in bytes, of the socket address pointed to by the *Group* parameter.

`[in] FilterMode`

The multicast filter mode for the multicast group address.

`[in] SourceCount`

The number of source addresses in the buffer pointed to by the *SourceList* parameter.

`[in] SourceList`

A pointer to a buffer with the IP addresses to associate with the multicast filter.

Return value

On success, **setsourcefilter** returns NO_ERROR (0). Any nonzero return value indicates failure and a specific error code can be retrieved by calling WSAGetLastError.

ERROR CODE	MEANING

WSAENOBUFS	Insufficient buffer space is available.
WSAENOTSOCK	The descriptor is not a socket.

Remarks

The **setsourcefilter** inline function is used to set the multicast filter state for an IPv4 or IPv6 socket.

This function is part of socket interface extensions for multicast source filters defined in RFC 3678. An app can use these functions to retrieve and set the multicast source address filters associated with a socket.

Windows Phone 8: This function is supported for Windows Phone Store apps on Windows Phone 8 and later.

Windows 8.1 and **Windows Server 2012 R2**: This function is supported for Windows Store apps on Windows 8.1, Windows Server 2012 R2, and later.

Requirements

Minimum supported client	Windows 8.1, Windows Vista [desktop apps | UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps | UWP apps]
Target Platform	Windows
Header	ws2tcpip.h
Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

MULTICAST_MODE_TYPE

SOCKADDR_STORAGE

getipv4sourcefilter

getsourcefilter

setipv4sourcefilter

WSADeleteSocketPeerTargetName function (ws2tcpip.h)

10/22/2022 • 2 minutes to read • Edit Online

The **WSADeleteSocketPeerTargetName** function removes the association between a peer target name and an IP address for a socket. After a successful return, there will be no future association between the IP address and the target name.

Syntax

```
INT WSAAPI WSADeleteSocketPeerTargetName(
  [in]           SOCKET                             Socket,
  [in]           const sockaddr                     *PeerAddr,
  [in]           ULONG                              PeerAddrLen,
  [in, optional] LPWSAOVERLAPPED                    Overlapped,
  [in, optional] LPWSAOVERLAPPED_COMPLETION_ROUTINE CompletionRoutine
);
```

Parameters

`[in] Socket`

A descriptor identifying a socket on which the peer target name is being deleted.

`[in] PeerAddr`

The IP address of the peer for which the target name is being deleted.

`[in] PeerAddrLen`

The size, in bytes, of the *PeerAddr* parameter.

`[in, optional] Overlapped`

A pointer to a WSAOVERLAPPED structure. This parameter is ignored for non-overlapped sockets.

`[in, optional] CompletionRoutine`

A pointer to the completion routine called when the operation has been completed. This parameter is ignored for non-overlapped sockets.

Return value

If the function succeeds, the return value is 0. Otherwise, a value of **SOCKET_ERROR** is returned, and a specific error code can be retrieved by calling WSAGetLastError.

Some possible error codes are listed below.

ERROR CODE	MEANING
WSAEAFNOSUPPORT	The specified address family is not supported.

WSAEFAULT	The system detected an invalid address pointer in attempting to use a pointer argument of a call. This error is returned if the *PeerAddr* parameter was a **NULL** pointer.
WSAEINVAL	An invalid parameter was passed. This error is returned if the socket passed in the *Socket* parameter was not created with an address family of the **AF_INET** or **AF_INET6** and a socket type of **SOCK_DGRAM** or **SOCK_STREAM**.
WSAEMSGSIZE	A buffer passed was too small.
WSAENOTSOCK	The descriptor passed in the *Socket* parameter is not a valid socket.

Remarks

The **WSADeleteSocketPeerTargetName** function provides a method to remove the association between a peer target name and an IP address for a socket. This function is used to delete a peer target name that was previously set with the WSASetSocketPeerTargetName function. After the **WSADeleteSocketPeerTargetName** function returns, no future authentication to the IP address will use the previously specified target name. This function is primarily designed to be used by connectionless clients (for example, a socket created with the type set to SOCK_DGRAM or the protocol set to IPPROTO_UDP) after they have terminated the connection with the IP address associated with the peer target name. For connection oriented clients (for example, a socket created with the type set to SOCK_STREAM or protocol set to IPPROTO_TCP), this function should not be called.

The **WSADeleteSocketPeerTargetName** function simplifies having to call the WSAIoctl function with a *dwIoControlCode* parameter set to **SIO_DELETE_PEER_TARGET_NAME**.

An error will be returned if the following conditions are not met.

- The address family of the *Socket* parameter must be either AF_INET or AF_INET6.
- The socket type must be either SOCK_STREAM or SOCK_DGRAM.

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows
Header	ws2tcpip.h
Library	Fwpuclnt.lib
DLL	Fwpuclnt.dll

See also

Using Secure Socket Extensions

WSAImpersonateSocketPeer

WSAQuerySocketSecurity

WSARevertImpersonation

WSASetSocketPeerTargetName

WSASetSocketSecurity

Windows Filtering Platform

Windows Filtering Platform API Functions

Winsock Secure Socket Extensions

WSAGetFailConnectOnIcmpError function (ws2tcpip.h)

10/22/2022 • 2 minutes to read • Edit Online

Queries the state of the TCP_FAIL_CONNECT_ON_ICMP_ERROR socket option.

Syntax

```
INT WSAGetFailConnectOnIcmpError(
  [in]  SOCKET Socket,
  [out] DWORD  *Enabled
);
```

Parameters

`[in] Socket`

A descriptor that identifies a TCP socket.

`[out] Enabled`

Type: DWORD*

A pointer to a **DWORD**. On success, the function sets the DWORD to a non-zero value if TCP_FAIL_CONNECT_ON_ICMP_ERROR is enabled, otherwise zero.

Return value

On success, the function returns 0. Otherwise, a value of SOCKET_ERROR is returned, and you can retrieve a specific error code by calling WSAGetLastError.

Remarks

This functionality is supported through the TCP_FAIL_CONNECT_ON_ICMP_ERROR socket option. **WSAGetFailConnectOnIcmpError** is a type-safe wrapper for getting this socket option, and we recommend it over getsockopt.

Requirements

Minimum supported client	Windows 10, version 2004 (10.0; Build 19041)
Minimum supported server	Windows Server, version 2004 (10.0; Build 19041)
Target Platform	Windows
Header	ws2tcpip.h

Library	Ws2_32.lib
DLL	Ws2_32.dll

WSAGetIcmpErrorInfo function (ws2tcpip.h)

10/22/2022 • 2 minutes to read • Edit Online

Retrieves information about an ICMP error received on a TCP socket during connection setup.

Syntax

```
INT WSAGetIcmpErrorInfo(
  [in]  SOCKET          Socket,
  [out] ICMP_ERROR_INFO *Info
);
```

Parameters

`[in] Socket`

A descriptor that identifies a TCP socket.

`[out] Info`

Type: **DWORD***

A pointer to an ICMP_ERROR_INFO structure. On success, the function initializes the structure.

Return value

On success, the function returns 0. Otherwise, a value of SOCKET_ERROR is returned, and you can retrieve a specific error code by calling WSAGetLastError.

Remarks

If no ICMP error has been received since the last connect call, then **WSANO_DATA** is returned. This functionality is supported through the **TCP_ICMP_ERROR_INFO** socket option. **WSAGetIcmpErrorInfo** is a type-safe wrapper for getting this socket option, and we recommend it over getsockopt.

Requirements

Minimum supported client	Windows 10, version 2004 (10.0; Build 19041)
Minimum supported server	Windows Server, version 2004 (10.0; Build 19041)
Target Platform	Windows
Header	ws2tcpip.h
Library	Ws2_32.lib
DLL	Ws2_32.dll

WSAGetIPUserMtu function (ws2tcpip.h)

10/22/2022 • 2 minutes to read • Edit Online

Retrieves the user-defined IP layer MTU for a socket.

Syntax

```
INT WSAGetIPUserMtu(
  [in]  SOCKET Socket,
  [out] DWORD  *Mtu
);
```

Parameters

`[in] Socket`

A descriptor that identifies a socket.

`[out] Mtu`

Type: **DWORD***

A pointer to a **DWORD**. On success, the function sets the DWORD to the user-defined IP layer MTU set on the socket.

Return value

On success, the function returns 0. Otherwise, a value of SOCKET_ERROR is returned, and you can retrieve a specific error code by calling WSAGetLastError.

Remarks

This functionality is supported through the **IP_USER_MTU** socket option. **WSAGetIPUserMtu** is a type-safe wrapper for getting this socket option, and we recommend it over getsockopt.

Requirements

Minimum supported client	Windows 10, version 2004 (10.0; Build 19041)
Minimum supported server	Windows Server, version 2004 (10.0; Build 19041)
Target Platform	Windows
Header	ws2tcpip.h
Library	Ws2_32.lib
DLL	Ws2_32.dll

WSAGetUdpRecvMaxCoalescedSize function (ws2tcpip.h)

10/22/2022 • 2 minutes to read • Edit Online

Retrieves the maximum size of a received, coalesced message for a UDP socket.

Syntax

```
INT WSAGetUdpRecvMaxCoalescedSize(
  [in]  SOCKET Socket,
  [out] DWORD  *MaxCoalescedMsgSize
);
```

Parameters

`[in] Socket`

A descriptor that identifies a UDP socket.

`[out] MaxCoalescedMsgSize`

Type: **DWORD***

A pointer to a **DWORD**. On success, the function sets the DWORD to the maximum coalesced message size used on the socket for UDP receive coalescing.

Return value

On success, the function returns 0. Otherwise, a value of SOCKET_ERROR is returned, and you can retrieve a specific error code by calling WSAGetLastError.

Remarks

UDP receive coalescing is supported through the UDP_RECV_MAX_COALESCED_SIZE socket option. **WSAGetUdpRecvMaxCoalescedSize** is a type-safe wrapper for getting this socket option, and we recommend it over getsockopt.

Requirements

Minimum supported client	Windows 10, version 2004 (10.0; Build 19041)
Minimum supported server	Windows Server, version 2004 (10.0; Build 19041)
Target Platform	Windows
Header	ws2tcpip.h

Library	Ws2_32.lib
DLL	Ws2_32.dll

WSAGetUdpSendMessageSize function (ws2tcpip.h)

10/22/2022 • 2 minutes to read • Edit Online

Retrieves the segmentation message size for a UDP socket.

Syntax

```
INT WSAGetUdpSendMessageSize(
  [in]  SOCKET Socket,
  [out] DWORD  *MsgSize
);
```

Parameters

`[in] Socket`

A descriptor that identifies a UDP socket.

`[out] MsgSize`

Type: **DWORD***

A pointer to a **DWORD**. On success, the function sets the DWORD to the message size used on the socket for UDP segmentation.

Return value

On success, the function returns 0. Otherwise, a value of SOCKET_ERROR is returned, and you can retrieve a specific error code by calling WSAGetLastError.

Remarks

UDP send segmentation is supported through the UDP_SEND_MSG_SIZE socket option. **WSAGetUdpSendMessageSize** is a type-safe wrapper for getting this socket option, and we recommend it over getsockopt.

Requirements

Minimum supported client	Windows 10, version 2004 (10.0; Build 19041)
Minimum supported server	Windows Server, version 2004 (10.0; Build 19041)
Target Platform	Windows
Header	ws2tcpip.h
Library	Ws2_32.lib

DLL	Ws2_32.dll

WSAImpersonateSocketPeer function (ws2tcpip.h)

10/22/2022 • 2 minutes to read • Edit Online

The **WSAImpersonateSocketPeer** function is used to impersonate the security principal corresponding to a socket peer in order to perform application-level authorization.

Syntax

```
INT WSAAPI WSAImpersonateSocketPeer(
  [in]           SOCKET         Socket,
  [in, optional] const sockaddr *PeerAddr,
  [in]           ULONG          PeerAddrLen
);
```

Parameters

`[in] Socket`

Identifies the application socket.

`[in, optional] PeerAddr`

The IP address of the peer to be impersonated. For connection-oriented sockets, the connected socket uniquely identifies a peer. In this case, this parameter is ignored.

`[in] PeerAddrLen`

The size, in bytes, of the *PeerAddress* parameter.

Return value

If the function succeeds, the return value is 0. Otherwise, a value of **SOCKET_ERROR** is returned, and a specific error code can be retrieved by calling WSAGetLastError.

Some possible error codes are listed below.

ERROR CODE	MEANING
WSAEFAULT	The system detected an invalid address pointer in attempting to use a pointer argument of a call. This error is returned if the *PeerAddr* parameter was a **NULL** pointer.
WSAEAFNOSUPPORT	The specified address family is not supported.
WSAEMSGSIZE	A buffer passed was too small.
WSAENOTSOCK	The descriptor passed in the *Socket* parameter is not a valid socket.

Remarks

The **WSAImpersonateSocketPeer** function provides an application the ability to impersonate the security principal corresponding to a socket peer in order to perform application-level authorization. If peer user (impersonation) token is available then it will be used for impersonation, otherwise the peer computer token will be used. The **WSAImpersonateSocketPeer** function can be called only for blocking, non-overlapped sockets. After performing any authorization checks, an application must call the WSARevertImpersonation function to terminate the impersonation.

For connection-oriented sockets, the **WSAImpersonateSocketPeer** function should be called after a connection is established. For a server application using connection-oriented sockets, the **WSAImpersonateSocketPeer** should be called after the accept, AcceptEx, or WSAAccept function returns.

For connectionless sockets, the application should call the **WSAImpersonateSocketPeer** function immediately after the recv, recvfrom, WSARecv, WSARecvEx, WSARecvFrom, or LPFN_WSARECVMSG (WSARecvMsg) function returns for a new peer address.

The **WSAImpersonateSocketPeer** function can be called multiple times for a single socket.

An error will be returned if the following conditions are not met.

- The address family of the *Socket* parameter must be either AF_INET or AF_INET6.
- The socket type must be either SOCK_STREAM or SOCK_DGRAM.

The WSARevertImpersonation function must be called to end the impersonation.

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows
Header	ws2tcpip.h
Library	Fwpuclnt.lib
DLL	Fwpuclnt.dll

See also

AcceptEx

Using Secure Socket Extensions

WSAAccept

WSADeleteSocketPeerTargetName

WSAQuerySocketSecurity

WSARecv

WSARecvEx

WSARecvFrom

LPFN_WSARECVMSG (WSARecvMsg)

WSARevertImpersonation

WSASetSocketPeerTargetName

WSASetSocketSecurity

Windows Filtering Platform

Windows Filtering Platform API Functions

Winsock Secure Socket Extensions

accept

recv

recvfrom

WSAQuerySocketSecurity function (ws2tcpip.h)

10/22/2022 • 4 minutes to read • Edit Online

The **WSAQuerySocketSecurity** function queries information about the security applied to a connection on a socket.

Syntax

```
INT WSAAPI WSAQuerySocketSecurity(
  [in]            SOCKET                                Socket,
  [in, optional]  const SOCKET_SECURITY_QUERY_TEMPLATE *SecurityQueryTemplate,
  [in]            ULONG                                 SecurityQueryTemplateLen,
  [out, optional] SOCKET_SECURITY_QUERY_INFO            *SecurityQueryInfo,
  [in, out]       ULONG                                 *SecurityQueryInfoLen,
  [in, optional]  LPWSAOVERLAPPED                       Overlapped,
  [in, optional]  LPWSAOVERLAPPED_COMPLETION_ROUTINE    CompletionRoutine
);
```

Parameters

`[in] Socket`

A descriptor identifying a socket for which security information is being queried.

`[in, optional] SecurityQueryTemplate`

A pointer to a SOCKET_SECURITY_QUERY_TEMPLATE structure that specifies the type of query information to return.

A SOCKET_SECURITY_QUERY_TEMPLATE structure pointed to by this parameter may contain zeroes for all members to request default security information. On successful return, only the **Flags** member in the SOCKET_SECURITY_QUERY_INFO will be set in the returned *SecurityQueryInfo* parameter.

This parameter may be a **NULL** pointer if the *Socket* parameter was created with a protocol of **IPPROTO_TCP**. In this case, the information returned is the same as if a SOCKET_SECURITY_QUERY_TEMPLATE structure with all values set to zero was passed. This parameter should be specified for a socket with protocol of **IPPROTO_TCP** if more than the default security information is required.

If the SOCKET_SECURITY_QUERY_TEMPLATE structure is specified with the **PeerTokenAccessMask** member not specified (set to zero), then the **WSAQuerySocketSecurity** function will not return the **PeerApplicationAccessTokenHandle** and **PeerMachineAccessTokenHandle** members in the SOCKET_SECURITY_QUERY_INFO structure.

If a *Socket* parameter was created with a protocol not equal to **IPPROTO_TCP**, the *SecurityQueryTemplate* parameter must be specified. In these cases, the **PeerAddress** member of the SOCKET_SECURITY_QUERY_TEMPLATE structure must specify an address family of AF_INET or AF_INET6 along with peer IP address and port number.

`[in] SecurityQueryTemplateLen`

The size, in bytes, of the *SecurityQueryTemplate* parameter.

This parameter may be a zero if the *Socket* parameter was created with a protocol of **IPPROTO_TCP**. Otherwise, this parameter must be the size of a SOCKET_SECURITY_QUERY_TEMPLATE structure.

`[out, optional] SecurityQueryInfo`

A pointer to a buffer that will receive a SOCKET_SECURITY_QUERY_INFO structure containing the information queried. This value can be set to **NULL** to query the size of the output buffer.

`[in, out] SecurityQueryInfoLen`

On input, a pointer to the size, in bytes, of the *SecurityQueryInfo* parameter. If the buffer is too small to receive the queried information, the call will return SOCKET_ERROR, and the number of bytes needed to return the queried information will be set in the value pointed to by this parameter. On a successful call, the number of bytes copied is returned.

`[in, optional] Overlapped`

A pointer to a WSAOVERLAPPED structure. This parameter is ignored for non-overlapped sockets.

`[in, optional] CompletionRoutine`

A pointer to the completion routine called when the operation has been completed. This parameter is ignored for non-overlapped sockets.

Return value

If the function succeeds, the return value is zero. Otherwise, a value of **SOCKET_ERROR** is returned, and a specific error code can be retrieved by calling WSAGetLastError.

Some possible error codes are listed below.

ERROR CODE	MEANING
WSAEAFNOSUPPORT	The specified address family is not supported.
WSAECONNRESET	For a stream socket, the virtual circuit was reset by the remote side. The application should close the socket as it is no longer usable. For a UDP datagram socket, this error would indicate that a previous send operation resulted in an ICMP "Port Unreachable" message.
WSAEFAULT	The system detected an invalid pointer address in attempting to use a parameter. This error is returned if the *SecurityQueryInfoLen* parameter was a **NULL** pointer.
WSAEINVAL	An invalid parameter was passed. This error is returned if the socket passed in the *Socket* parameter was not created with an address family of the **AF_INET** or **AF_INET6** and a socket type of **SOCK_DGRAM** or **SOCK_STREAM**.
WSAEMSGSIZE	A buffer passed was too small. This error is returned for a *Socket* parameter when the protocol was not **IPPROTO_TCP** if the *SecurityQueryInfo* parameter is a **NULL** pointer or the *SecurityQueryTemplateLen* parameter is less than the size of a SOCKET_SECURITY_QUERY_TEMPLATE structure.
WSAENOTSOCK	The descriptor passed in the *Socket* parameter is not a valid socket.

Remarks

The **WSAQuerySocketSecurity** function provides a method to query the current security settings on a socket. After a connection is established, the **WSAQuerySocketSecurity** function allows an application to query the security properties of the connection, which can include information on peer access tokens.

For connection-oriented sockets, it is preferred to call the **WSAQuerySocketSecurity** function immediately after a connection is established. For connectionless sockets, it is preferred to call the **WSAQuerySocketSecurity** function immediately after data is sent to a new peer address or received from a new peer address. The **WSAQuerySocketSecurity** function can be called multiple times on a single socket.

This function simplifies having to call the WSAIoctl function with a *dwIoControlCode* parameter set to **SIO_QUERY_SECURITY**.

The **WSAQuerySocketSecurity** function may be called on a *Socket* parameter created with an address family of **AF_INET** or **AF_INET6**.

If the *Socket* parameter was created with a protocol of **IPPROTO_TCP**, the *SecurityQueryTemplate* parameter may be **NULL** and the *SecurityQueryTemplateLen* parameter may be zero. Otherwise, the *SecurityQueryTemplate* parameter must point to a SOCKET_SECURITY_QUERY_TEMPLATE structure.

For a client application using connection-oriented sockets (socket created with a protocol of **IPPROTO_TCP**), the **WSAQuerySocketSecurity** function should be called after the connect, ConnectEx, or WSAConnect function returns. For a server application using connection-oriented sockets (protocol of **IPPROTO_TCP**), the **WSAQuerySocketSecurity** function should be called after the accept, AcceptEx, or WSAAccept function returns.

For connectionless sockets (socket created with a protocol of **IPPROTO_UDP**), the application should call the **WSAQuerySocketSecurity** function immediately after WSASendTo or WSARecvFrom call returns for a new peer address.

An error will be returned if the following conditions are not met.

- The address family of the *Socket* parameter must be either AF_INET or AF_INET6.
- The socket type must be either SOCK_STREAM or SOCK_DGRAM.

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows
Header	ws2tcpip.h
Library	Fwpuclnt.lib
DLL	Fwpuclnt.dll

See also

SOCKET_SECURITY_QUERY_INFO

SOCKET_SECURITY_QUERY_TEMPLATE

Using Secure Socket Extensions

WSADeleteSocketPeerTargetName

WSAImpersonateSocketPeer

WSARevertImpersonation

WSASetSocketPeerTargetName

WSASetSocketSecurity

Windows Filtering Platform

Windows Filtering Platform API Functions

Winsock Secure Socket Extensions

WSARevertImpersonation function (ws2tcpip.h)

10/22/2022 • 2 minutes to read • Edit Online

The **WSARevertImpersonation** function terminates the impersonation of a socket peer. This must be called after calling WSAImpersonateSocketPeer and finishing any access checks.

Syntax

```
INT WSAAPI WSARevertImpersonation();
```

Return value

If the function succeeds, the return value is zero. Otherwise, a value of **SOCKET_ERROR** is returned, and a specific error code can be retrieved by calling WSAGetLastError.

Some possible error codes are listed below.

ERROR CODE	MEANING
WSASYSCALLFAILURE	A system call that should never fail has failed.

Remarks

The **WSARevertImpersonation** function causes the calling thread to discontinue the impersonation of a socket peer. If the thread is not currently impersonating a socket peer, no action is taken.

The **WSARevertImpersonation** function should be called after calling WSAImpersonateSocketPeer and all access checks are finished.

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows
Header	ws2tcpip.h
Library	Fwpuclnt.lib
DLL	Fwpuclnt.dll

See also

Using Secure Socket Extensions

WSADeleteSocketPeerTargetName

WSAImpersonateSocketPeer

WSAQuerySocketSecurity

WSASetSocketPeerTargetName

WSASetSocketSecurity

Windows Filtering Platform

Windows Filtering Platform API Functions

Winsock Secure Socket Extensions

WSASetFailConnectOnIcmpError function (ws2tcpip.h)

10/22/2022 • 2 minutes to read • Edit Online

Sets the state of the TCP_FAIL_CONNECT_ON_ICMP_ERROR socket option.

Syntax

```
INT WSASetFailConnectOnIcmpError(
  [in] SOCKET Socket,
  [in] DWORD  Enabled
);
```

Parameters

`[in] Socket`

A descriptor that identifies a TCP socket.

`[in] Enabled`

Type: DWORD

A non-zero value if TCP_FAIL_CONNECT_ON_ICMP_ERROR should be enabled, otherwise zero.

Return value

On success, the function returns 0. Otherwise, a value of SOCKET_ERROR is returned, and you can retrieve a specific error code by calling WSAGetLastError.

Remarks

This functionality is supported through the TCP_FAIL_CONNECT_ON_ICMP_ERROR socket option. **WSASetFailConnectOnIcmpError** is a type-safe wrapper for setting this socket option, and we recommend it over setsockopt.

Requirements

Minimum supported client	Windows 10, version 2004 (10.0; Build 19041)
Minimum supported server	Windows Server, version 2004 (10.0; Build 19041)
Target Platform	Windows
Header	ws2tcpip.h
Library	Ws2_32.lib

DLL	Ws2_32.dll

WSASetIPUserMtu function (ws2tcpip.h)

10/22/2022 • 2 minutes to read • Edit Online

Sets the user-defined IP layer MTU on a socket.

Syntax

```
INT WSASetIPUserMtu(
  [in] SOCKET Socket,
  [in] DWORD  Mtu
);
```

Parameters

`[in] Socket`

A descriptor that identifies a socket.

`[in] Mtu`

Type: **DWORD**

The user-defined IP layer MTU to be set on the socket.

Return value

On success, the function returns 0. Otherwise, a value of SOCKET_ERROR is returned, and you can retrieve a specific error code by calling WSAGetLastError.

Remarks

This functionality is supported through the **IP_USER_MTU** socket option. **WSASetIPUserMtu** is a type-safe wrapper for setting this socket option, and we recommend it over setsockopt.

Requirements

Minimum supported client	Windows 10, version 2004 (10.0; Build 19041)
Minimum supported server	Windows Server, version 2004 (10.0; Build 19041)
Target Platform	Windows
Header	ws2tcpip.h
Library	Ws2_32.lib
DLL	Ws2_32.dll

WSASetSocketPeerTargetName function (ws2tcpip.h)

10/22/2022 • 2 minutes to read • Edit Online

The **WSASetSocketPeerTargetName** function is 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.

Syntax

```
INT WSAAPI WSASetSocketPeerTargetName(
  [in]           SOCKET                              Socket,
  [in]           const SOCKET_PEER_TARGET_NAME       *PeerTargetName,
  [in]           ULONG                               PeerTargetNameLen,
  [in, optional] LPWSAOVERLAPPED                     Overlapped,
  [in, optional] LPWSAOVERLAPPED_COMPLETION_ROUTINE  CompletionRoutine
);
```

Parameters

`[in] Socket`

A descriptor identifying a socket on which the peer target name is being assigned.

`[in] PeerTargetName`

A pointer to a SOCKET_PEER_TARGET_NAME structure that defines the peer target name.

`[in] PeerTargetNameLen`

The size, in bytes, of the *PeerTargetName* parameter.

`[in, optional] Overlapped`

A pointer to a WSAOVERLAPPED structure. This parameter is ignored for non-overlapped sockets.

`[in, optional] CompletionRoutine`

A pointer to the completion routine called when the operation has been completed. This parameter is ignored for non-overlapped sockets.

Return value

If the function succeeds, the return value is zero. Otherwise, a value of **SOCKET_ERROR** is returned, and a specific error code can be retrieved by calling WSAGetLastError.

Some possible error codes are listed below.

ERROR CODE	MEANING
WSAEAFNOSUPPORT	The specified address family is not supported.

WSAEFAULT	The system detected an invalid address pointer in attempting to use a pointer argument of a call. This error is returned if the *PeerTargetName* parameter was a **NULL** pointer.
WSAEINVAL	An invalid parameter was passed. This error is returned if the socket passed in the *Socket* parameter was not created with an address family of the **AF_INET** or **AF_INET6** and a socket type of **SOCK_DGRAM** or **SOCK_STREAM**. This error is also returned for a connectionless socket if the IP address and port are zero in the **PeerAddress** member of the SOCKET_PEER_TARGET_NAME structure pointed to by the *PeerTargetName* parameter.
WSAEISCONN	The socket is connected. This function is not permitted with a connected socket, whether the socket is connection oriented or connectionless.
WSAEMSGSIZE	A buffer passed was too small.
WSAENOTSOCK	The descriptor passed in the *Socket* parameter is not a valid socket.

Remarks

The **WSASetSocketPeerTargetName** function provides a method to specify the target name that corresponds to a peer security principal. This function is meant to be used by a client application to identify the peer that should be authenticated. A client application should specify the peer target name in order to prevent trusted man-in-the-middle attacks. For connectionless sockets, an application can call the **WSASetSocketPeerTargetName** function multiple times to specify different target names for different peer IP addresses.

This function simplifies having to call the WSAIoctl function with a *dwIoControlCode* parameter set to **SIO_SET_PEER_TARGET_NAME**.

For connection-oriented sockets, the **WSASetSocketPeerTargetName** function should be called before WSAConnect. For connectionless sockets, this function should be called before **WSAConnect** or before the first WSASendTo call directed to the peer address.

An error will be returned if the following conditions are not met.

- The address family of the *Socket* parameter must be either AF_INET or AF_INET6.
- The socket type must be either SOCK_STREAM or SOCK_DGRAM.

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows

Header	ws2tcpip.h
Library	Fwpuclnt.lib
DLL	Fwpuclnt.dll

See also

SOCKET_PEER_TARGET_NAME

Using Secure Socket Extensions

WSADeleteSocketPeerTargetName

WSAImpersonateSocketPeer

WSAQuerySocketSecurity

WSARevertImpersonation

WSASetSocketSecurity

Windows Filtering Platform

Windows Filtering Platform API Functions

Winsock Secure Socket Extensions

WSASetSocketSecurity function (ws2tcpip.h)

10/22/2022 • 4 minutes to read • Edit Online

The **WSASetSocketSecurity** function enables and applies security for a socket.

Syntax

```
INT WSAAPI WSASetSocketSecurity(
  [in]           SOCKET                              Socket,
  [in, optional] const SOCKET_SECURITY_SETTINGS      *SecuritySettings,
  [in]           ULONG                               SecuritySettingsLen,
  [in, optional] LPWSAOVERLAPPED                     Overlapped,
  [in, optional] LPWSAOVERLAPPED_COMPLETION_ROUTINE CompletionRoutine
);
```

Parameters

`[in] Socket`

A descriptor that identifies a socket on which security settings are being applied.

`[in, optional] SecuritySettings`

A pointer to a SOCKET_SECURITY_SETTINGS structure that specifies the security settings to be applied to the socket's traffic. If this parameter is **NULL**, default settings will be applied to the socket.

`[in] SecuritySettingsLen`

The size, in bytes, of the *SecuritySettings* parameter.

`[in, optional] Overlapped`

A pointer to a WSAOVERLAPPED structure. This parameter is ignored for non-overlapped sockets.

`[in, optional] CompletionRoutine`

A pointer to the completion routine called when the operation has been completed. This parameter is ignored for non-overlapped sockets.

Return value

If the function succeeds, the return value is zero. Otherwise, a value of **SOCKET_ERROR** is returned, and a specific error code can be retrieved by calling WSAGetLastError.

Some possible error codes are listed below.

ERROR CODE	MEANING
WSAEAFNOSUPPORT	The specified address family is not supported.

WSAEINVAL	An invalid parameter was passed. This error is returned if the socket passed in the *Socket* parameter was not created with an address family of the **AF_INET** or **AF_INET6** and a socket type of **SOCK_DGRAM** or **SOCK_STREAM**. This error is also returned if the SOCKET_SECURITY_SETTINGS structure pointed to by the *SecuritySettings* parameter has an incorrect value.
WSAEISCONN	The socket is connected. This function is not permitted with a connected socket, whether the socket is connection oriented or connectionless.
WSAEMSGSIZE	A buffer passed was too small.
WSAENOTSOCK	The descriptor passed in the *Socket* parameter is not a valid socket.

Remarks

The primary purpose of the **WSASetSocketSecurity** function is to turn on security for a socket if it is not already enabled by administrative policy. For IPsec, this means that appropriate IPsec filters and policies will be instantiated that will be used to secure this socket. the **WSASetSocketSecurity** function can also be used to set specific security requirements for the socket.

This function simplifies having to call the WSAIoctl function with a *dwIoControlCode* parameter set to **SIO_SET_SECURITY**.

The **WSASetSocketSecurity** function may be called on a *Socket* parameter created with an address family of **AF_INET** or **AF_INET6**.

For a client application using connection-oriented sockets (protocol of **IPPROTO_TCP**), the **WSASetSocketSecurity** function should be called before the connect, ConnectEx, or WSAConnect function is called. If the **WSASetSocketSecurity** function is called after the **connect**, **ConnectEx**, or **WSAConnect** function, **WSASetSocketSecurity** should fail.

For a server application using connection-oriented sockets (protocol of **IPPROTO_TCP**), the **WSASetSocketSecurity** function should be called before the bind function is called. If the **WSASetSocketSecurity** function is called after the **bind** function, **WSASetSocketSecurity** should fail.

For connectionless sockets (protocol of **IPPROTO_UDP**), the application should call the **WSASetSocketSecurity** function immediately after socket or WSASocket call returns.

Server applications should call the setsockopt function to acquire exclusive access to the port used by the socket. This prevents other applications from using the same port. The **setsockopt** function would be called with the *level* parameter set to SOL_SOCKET, the *optname* parameter set to SO_EXCLUSIVEADDRUSE, and the *value* parameter set to nonzero. The **WSASetSocketSecurity** function internally calls the **setsockopt** with SO_EXCLUSIVEADDRUSE to obtain exclusive access to the port. This is to ensure that the socket is not vulnerable to attacks by other applications running on the local computer.

Security settings not set using the **WSASetSocketSecurity** are derived from the system default policy or the administratively configured policy. It is recommended that most applications specify a value of **SOCKET_SECURITY_PROTOCOL_DEFAULT** for the SOCKET_SECURITY_PROTOCOL enumeration in the **SecurityProtocol** member of the **SOCKET_SECURITY_PROTOCOL** pointed to by the *SecuritySettings* parameter. This makes the application neutral to security protocols and allows easier deployments among different systems.

When the *SecuritySettings* parameter points to a SOCKET_SECURITY_SETTINGS_IPSEC structure, the **SecurityProtocol** member of the structure must be set to **SOCKET_SECURITY_PROTOCOL_IPSEC**, not **SOCKET_SECURITY_PROTOCOL_DEFAULT**.

An error will be returned if the following conditions are not met.

- The address family of the *Socket* parameter must be either AF_INET or AF_INET6.
- The socket type must be either SOCK_STREAM or SOCK_DGRAM.
- The application must set its security settings before calling the bind, connect, ConnectEx, or WSAConnect functions.
- The **WSASetSocketSecurity** function can only be called once per socket.

Default Secure Socket IPsec Policy

If the *SecuritySettings* parameter is set to **NULL**, and there is no other administratively specified IPsec policy on the computer, a default security policy based on IPsec will be used to secure the application's traffic. Some type of authentication credential (a user certificate or domain membership, for example) must be present for IPsec to succeed with a default policy.

The default IPsec policy has been designed so that IPsec security can be negotiated in as many scenarios as possible.

```
Authip MM policy =
{
 Auth methods = {IKE_ANONYMOUS}
 No impersonation
 Proposals =
 {
   {
     Crypto algos =
     IKE_CIPHER_AES_128,
     IKE_INTEGRITY_SHA1,
     IKE_DH_ECP_256
     MM lifetime = 2 hrs
     QM = 0 (infinite)
   }
   {
     Crypto algos =
     IKE_CIPHER_3DES,
     IKE_INTEGRITY_SHA1,
     IKE_DH_GROUP_2
     MM lifetime = 2 hrs
     QM = 0 (infinite)
   }
 }
}

Authip QM policy =
{
 QM proposals =
 {
   QM lifetime = 1 hr, 55GB,
   Crypto algos =
   IPSEC_TRANSFORM_ESP_AUTH,
   IPSEC_AUTH_TRANSFORM_ID_HMAC_SHA_1_96
   No PFS
 }
 {
   QM lifetime = 1 hr, 55GB,
   Crypto algos =
   IPSEC_TRANSFORM_ESP_AUTH_AND_CIPHER,
   IPSEC_AUTH_TRANSFORM_ID_HMAC_SHA_1_96,
   IPSEC_CIPHER_TRANSFORM_ID_AES_128
   No PFS
 }
```

```
  {
    QM lifetime = 1 hr, 55GB,
    Crypto algos =
    IPSEC_TRANSFORM_ESP_AUTH_AND_CIPHER,
    IPSEC_AUTH_TRANSFORM_ID_HMAC_SHA_1_96,
    IPSEC_CIPHER_TRANSFORM_ID_CBC_3DES
    No PFS
  }
  {
    QM lifetime = 1 hr, 55GB,
    Crypto algos =
    IPSEC_TRANSFORM_AH,
    IPSEC_AUTH_TRANSFORM_ID_HMAC_SHA_1_96
    No PFS
  }
  IPSEC_POLICY_FLAG_ND_BOUNDARY
  ndAllowClearTimeoutSeconds = 10
  saIdleTimeout = {5mins, 1min}
  UM policy =
  {
    {IKE_SSL, Null-Root-Config}
    {IKE_KERBEROS}
    {IKE_SSL, Null-Root-Config}
    No impersonation
  }
 }
```

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows
Header	ws2tcpip.h
Library	Fwpuclnt.lib
DLL	Fwpuclnt.dll

See also

AuthIP in Windows Vista

SOCKET_SECURITY_PROTOCOL

SOCKET_SECURITY_SETTINGS

SOCKET_SECURITY_SETTINGS_IPSEC

SO_EXCLUSIVEADDRUSE

Using Secure Socket Extensions

WSADeleteSocketPeerTargetName

WSAImpersonateSocketPeer

WSAOVERLAPPED

WSAQuerySocketSecurity

WSARevertImpersonation

WSASetSocketPeerTargetName

WSASetSocketSecurity

Windows Filtering Platform

Windows Filtering Platform API Functions

Winsock Secure Socket Extensions

WSASetUdpRecvMaxCoalescedSize function (ws2tcpip.h)

10/22/2022 • 2 minutes to read • Edit Online

Sets the maximum size of a coalesced message set on a UDP socket.

Syntax

```
INT WSASetUdpRecvMaxCoalescedSize(
  [in] SOCKET Socket,
  [in] DWORD  MaxCoalescedMsgSize
);
```

Parameters

`[in] Socket`

A descriptor that identifies a UDP socket.

`[in] MaxCoalescedMsgSize`

Type: **DWORD**

The maximum coalesced message size to be set on the socket for UDP receive coalescing.

Return value

On success, the function returns 0. Otherwise, a value of SOCKET_ERROR is returned, and you can retrieve a specific error code by calling WSAGetLastError.

Remarks

UDP receive coalescing is supported through the UDP_RECV_MAX_COALESCED_SIZE socket option. **WSASetUdpRecvMaxCoalescedSize** is a type-safe wrapper for setting this socket option, and we recommend it over setsockopt.

Requirements

Minimum supported client	Windows 10, version 2004 (10.0; Build 19041)
Minimum supported server	Windows Server, version 2004 (10.0; Build 19041)
Target Platform	Windows
Header	ws2tcpip.h
Library	Ws2_32.lib

DLL	Ws2_32.dll

WSASetUdpSendMessageSize function (ws2tcpip.h)

10/22/2022 • 2 minutes to read • Edit Online

Sets the segmentation message size on a UDP socket.

Syntax

```
INT WSASetUdpSendMessageSize(
  [in] SOCKET Socket,
  [in] DWORD  MsgSize
);
```

Parameters

`[in] Socket`

A descriptor that identifies a UDP socket.

`[in] MsgSize`

Type: **DWORD**

The message size to be set on the socket for UDP segmentation.

Return value

On success, the function returns 0. Otherwise, a value of SOCKET_ERROR is returned, and you can retrieve a specific error code by calling WSAGetLastError.

Remarks

UDP send segmentation is supported through the UDP_SEND_MSG_SIZE socket option. **WSASetUdpSendMessageSize** is a type-safe wrapper for setting this socket option, and we recommend it over setsockopt.

Requirements

Minimum supported client	Windows 10, version 2004 (10.0; Build 19041)
Minimum supported server	Windows Server, version 2004 (10.0; Build 19041)
Target Platform	Windows
Header	ws2tcpip.h
Library	Ws2_32.lib
DLL	Ws2_32.dll

wsipv6ok.h header

10/22/2022 • 2 minutes to read • Edit Online

This header is used by Windows Sockets 2. For more information, see:

- Windows Sockets 2

wsipv6ok.h contains the following programming interfaces:

Functions

gethostbyaddr The gethostbyaddr macro function (wsipv6ok.h) retrieves the host information corresponding to a network address.
gethostbyname The gethostbyname macro function (wsipv6ok.h) retrieves host information corresponding to a host name from a host database.
inet_addr The inet_addr macro function (wsipv6ok.h) converts a string containing an IPv4 dotted-decimal address into a proper address for the IN_ADDR structure.
inet_ntoa The inet_ntoa macro function (wsipv6ok.h) converts an (Ipv4) Internet network address into an ASCII string in Internet standard dotted-decimal format.
WSAAsyncGetHostByAddr The WSAAsyncGetHostByAddr macro function (wsipv6ok.h) asynchronously retrieves host information that corresponds to an address.
WSAAsyncGetHostByName The WSAAsyncGetHostByName macro function (wsipv6ok.h) asynchronously retrieves host information that corresponds to a host name.

gethostbyaddr macro (wsipv6ok.h)

10/22/2022 • 4 minutes to read • Edit Online

[**gethostbyaddr** is no longer recommended for use as of Windows Sockets 2. Instead, use getnameinfo.]

The **gethostbyaddr** function retrieves the host information corresponding to a network address.

Syntax

```
void gethostbyaddr(
  [in]  a,
  [in]  b,
  [in]  c
);
```

Parameters

`[in] a`

A pointer to an address in network byte order.

`[in] b`

The length, in bytes, of the address.

`[in] c`

The type of the address, such as the AF_INET address family type (used with TCP, UDP, and other associated Internet protocols). Possible values for the address family are defined in the Winsock2.h header file.

On the Windows SDK released for Windows Vista and later, the organization of header files has changed and the 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.

Note that the values for the AF_ address family and PF_ protocol family constants are identical (for example, **AF_INET** and **PF_INET**), so either constant can be used.

The table below lists the possible values for address family that are supported.

VALUE	MEANING
AF_INET 2	The Internet Protocol version 4 (IPv4) address family.
AF_NETBIOS 17	The NetBIOS address family. This address family is only supported if a Windows Sockets provider for NetBIOS is installed.
AF_INET6 23	The Internet Protocol version 6 (IPv6) address family.

Return value

None

Remarks

The **gethostbyaddr** function returns a pointer to the hostent structure that contains the name and address corresponding to the given network address.

The memory for the hostent structure returned by the **gethostbyaddr** function is allocated internally by the Winsock DLL from thread local storage. Only a single **hostent** structure is allocated and used, no matter how many times the **gethostbyaddr** or gethostbyname functions are called on the thread. The returned **hostent** structure must be copied to an application buffer if additional calls are to be made to the **gethostbyaddr** or **gethostbyname** functions on the same thread. Otherwise, the return value will be overwritten by subsequent **gethostbyaddr** or **gethostbyname** calls on the same thread. The internal memory allocated for the returned **hostent** structure is released by the Winsock DLL when the thread exits.

An application should not try to release the memory used by the returned hostent structure. The application must never attempt to modify this structure or to free any of its components. Furthermore, only one copy of this structure is allocated per thread, so the application should copy any information it needs before issuing any other function calls to **gethostbyaddr** or gethostbyname.

Although **gethostbyaddr** no longer recommended for use as of Windows Sockets 2 and the getnameinfo function should be used, **gethostbyaddr** is capable of returning a NetBIOS name; **getnameinfo** is not. Developers requiring NetBIOS name resolution may need to use **gethostbyaddr** until their applications are completely independent of NetBIOS names.

> **Note** The capability to perform reverse lookups using the **gethostbyaddr** function is convenient, but such lookups are considered inherently unreliable, and should be used only as a hint.

Example Code

The following example demonstrates the use of the **gethostbyaddr** function.

```
#include <winsock2.h>
#include <ws2tcpip.h>
#include <stdio.h>

int main(int argc, char **argv)
{

    //-----------------------------------------
    // Declare and initialize variables
    WSADATA wsaData;
    int iResult;

    DWORD dwError;
    int i = 0;
    int bIpv6 = 0;

    struct hostent *remoteHost;
    char *host_addr;
    struct in_addr addr = { 0 };
    IN6_ADDR addr6;

    char **pAlias;

    // Validate the parameters
    if (argc < 2) {
```

```
    if (argc < 2) {
        printf("usage: %s 4 ipv4address\n", argv[0]);
        printf(" or\n");
        printf("usage: %s 6 ipv6address\n", argv[0]);
        printf("  to return the hostname\n");
        printf("       %s 4 127.0.0.1\n", argv[0]);
        printf("       %s 6 0::1\n", argv[0]);
        return 1;
    }
    // Validate parameters
    if (atoi(argv[1]) == 4)
        bIpv6 = 0;
    else if (atoi(argv[1]) == 6)
        bIpv6 = 1;
    else {
        printf("usage: %s 4 ipv4address\n", argv[0]);
        printf(" or\n");
        printf("usage: %s 6 ipv6address\n", argv[0]);
        printf("  to return the hostname\n");
        printf("       %s 4 127.0.0.1\n", argv[0]);
        printf("       %s 6 0::1\n", argv[0]);
        return 1;
    }

    // Initialize Winsock
    iResult = WSAStartup(MAKEWORD(2, 2), &wsaData);
    if (iResult != 0) {
        printf("WSAStartup failed: %d\n", iResult);
        return 1;
    }

    host_addr = argv[2];

    printf("Calling gethostbyaddr with %s\n", host_addr);
    if (bIpv6 == 1) {
        {
            iResult = inet_pton(AF_INET6, host_addr, &addr6);
            if (iResult == 0) {
                printf("The IPv6 address entered must be a legal address\n");
                return 1;
            } else
                remoteHost = gethostbyaddr((char *) &addr6, 16, AF_INET6);
        }
    } else {
        addr.s_addr = inet_addr(host_addr);
        if (addr.s_addr == INADDR_NONE) {
            printf("The IPv4 address entered must be a legal address\n");
            return 1;
        } else
            remoteHost = gethostbyaddr((char *) &addr, 4, AF_INET);
    }

    if (remoteHost == NULL) {
        dwError = WSAGetLastError();
        if (dwError != 0) {
            if (dwError == WSAHOST_NOT_FOUND) {
                printf("Host not found\n");
                return 1;
            } else if (dwError == WSANO_DATA) {
                printf("No data record found\n");
                return 1;
            } else {
                printf("Function failed with error: %ld\n", dwError);
                return 1;
            }
        }
    } else {
        printf("Function returned:\n");
        printf("\tOfficial name: %s\n", remoteHost->h_name);
        for (pAlias = remoteHost->h_aliases; *pAlias != 0; pAlias++) {
```

```
        for (pAlias = remoteHost->h_aliases; *pAlias != 0; pAlias++) {
            printf("\tAlternate name #%d: %s\n", ++i, *pAlias);
        }
        printf("\tAddress type: ");
        switch (remoteHost->h_addrtype) {
        case AF_INET:
            printf("AF_INET\n");
            break;
        case AF_INET6:
            printf("AF_INET6\n");
            break;
        case AF_NETBIOS:
            printf("AF_NETBIOS\n");
            break;
        default:
            printf(" %d\n", remoteHost->h_addrtype);
            break;
        }
        printf("\tAddress length: %d\n", remoteHost->h_length);

        if (remoteHost->h_addrtype == AF_INET) {
            while (remoteHost->h_addr_list[i] != 0) {
                addr.s_addr = *(u_long *) remoteHost->h_addr_list[i++];
                printf("\tIPv4 Address #%d: %s\n", i, inet_ntoa(addr));
            }
        } else if (remoteHost->h_addrtype == AF_INET6)
            printf("\tRemotehost is an IPv6 address\n");
    }

    return 0;
}
```

Windows Phone 8: This function is supported for Windows Phone Store apps on Windows Phone 8 and later.

Windows 8.1 and **Windows Server 2012 R2**: This function is supported for Windows Store apps on Windows 8.1, Windows Server 2012 R2, and later.

Requirements

Minimum supported client	Windows 8.1, Windows Vista [desktop apps | UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps | UWP apps]
Target Platform	Windows
Header	wsipv6ok.h (include Winsock2.h, Winsock.h)
Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

GetAddrInfoEx

GetAddrInfoW

WSAAsyncGetHostByAddr

Winsock Functions

Winsock Reference

addrinfo

addrinfoW

getaddrinfo

gethostbyname

hostent

gethostbyname macro (wsipv6ok.h)

10/22/2022 • 4 minutes to read • Edit Online

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

> **Note** The **gethostbyname** function has been deprecated by the introduction of the getaddrinfo function. Developers creating Windows Sockets 2 applications are urged to use the **getaddrinfo** function instead of **gethostbyname**.

Syntax

```
void gethostbyname(
  [in]  a
);
```

Parameters

`[in] a`

A pointer to the **null**-terminated name of the host to resolve.

Return value

None

Remarks

The **gethostbyname** function returns a pointer to a hostent structure—a structure allocated by Windows Sockets. The **hostent** structure contains the results of a successful search for the host specified in the *name* parameter.

If the host specified in the *name* parameter has both IPv4 and IPv6 addresses, only the IPv4 addresses will be returned. The **gethostbyname** function can only return IPv4 addresses for the *name* parameter. The getaddrinfo function and associated addrinfo structure should be used if IPv6 addresses for the host are required or if both IPv4 and IPv6 addresses for the host are required.

If the *name* parameter points to an empty string or *name* is **NULL**, the returned string is the same as the string returned by a successful gethostname function call (the standard host name for the local computer).

If the *name* parameter contains a string representation of a legal IPv4 address, then the binary IPv4 address that represents the string is returned in the hostent structure. The **h_name** member of the **hostent** structure contains the string representation of the IPv4 address and the **h_addr_list** contains the binary IPv4 address. If the *name* parameter contains a string representation of an IPv6 address or an illegal IPv4 address, then the **gethostbyname** function will fail and return WSANO_DATA.

The memory for the hostent structure returned by the **gethostbyname** function is allocated internally by the Winsock DLL from thread local storage. Only a single **hostent** structure is allocated and used, no matter how many times the gethostbyaddr or **gethostbyname** functions are called on the thread. The returned **hostent** structure must be copied to an application buffer if additional calls are to be made to the **gethostbyname** or

gethostbyaddr functions on the same thread. Otherwise, the return value will be overwritten by subsequent **gethostbyname** or **gethostbyaddr** calls on the same thread. The internal memory allocated for the returned **hostent** structure is released by the Winsock DLL when the thread exits.

An application should not try to release the memory used by the returned hostent structure. The application must never attempt to modify this structure or to free any of its components. Furthermore, only one copy of this structure is allocated per thread, so the application should copy any information it needs before issuing any other function calls to **gethostbyname** or gethostbyaddr .

The **gethostbyname** function cannot take an IP address string as a parameter passed to it in the *name* and resolve it to a host name. Such a request is treated exactly as a string representation of an IPv4 address or an unknown host name were passed. An application can use the inet_addr to convert an IPv4 address string to a binary IPv4 address, then use another function, gethostbyaddr, to resolve the IPv4 address to a host name.

> **Note** The **gethostbyname** function does not check the size of the *name* parameter before passing the buffer. With an improperly sized *name* parameter, heap corruption can occur.

Example Code

The following examples demonstrates the use of the **gethostbyname** function.

```
#include <winsock2.h>
#include <ws2tcpip.h>
#include <stdio.h>
#include <windows.h>
#pragma comment(lib, "ws2_32.lib")

int main(int argc, char **argv)
{

    //-----------------------------------------
    // Declare and initialize variables
    WSADATA wsaData;
    int iResult;

    DWORD dwError;
    int i = 0;

    struct hostent *remoteHost;
    char *host_name;
    struct in_addr addr;

    char **pAlias;

    // Validate the parameters
    if (argc != 2) {
        printf("usage: %s hostname\n", argv[0]);
        printf("  to return the IP addresses for the host\n");
        printf("       %s www.contoso.com\n", argv[0]);
        printf(" or\n");
        printf("       %s IPv4string\n", argv[0]);
        printf("  to return an IPv4 binary address for an IPv4string\n");
        printf("       %s 127.0.0.1\n", argv[0]);
        return 1;
    }
    // Initialize Winsock
    iResult = WSAStartup(MAKEWORD(2, 2), &wsaData);
    if (iResult != 0) {
        printf("WSAStartup failed: %d\n", iResult);
        return 1;
    }
```

```
    host_name = argv[1];

    printf("Calling gethostbyname with %s\n", host_name);
    remoteHost = gethostbyname(host_name);

    if (remoteHost == NULL) {
        dwError = WSAGetLastError();
        if (dwError != 0) {
            if (dwError == WSAHOST_NOT_FOUND) {
                printf("Host not found\n");
                return 1;
            } else if (dwError == WSANO_DATA) {
                printf("No data record found\n");
                return 1;
            } else {
                printf("Function failed with error: %ld\n", dwError);
                return 1;
            }
        }
    } else {
        printf("Function returned:\n");
        printf("\tOfficial name: %s\n", remoteHost->h_name);
        for (pAlias = remoteHost->h_aliases; *pAlias != 0; pAlias++) {
            printf("\tAlternate name #%d: %s\n", ++i, *pAlias);
        }
        printf("\tAddress type: ");
        switch (remoteHost->h_addrtype) {
        case AF_INET:
            printf("AF_INET\n");
            break;
        case AF_NETBIOS:
            printf("AF_NETBIOS\n");
            break;
        default:
            printf(" %d\n", remoteHost->h_addrtype);
            break;
        }
        printf("\tAddress length: %d\n", remoteHost->h_length);

        i = 0;
        if (remoteHost->h_addrtype == AF_INET)
        {
            while (remoteHost->h_addr_list[i] != 0) {
                addr.s_addr = *(u_long *) remoteHost->h_addr_list[i++];
                printf("\tIP Address #%d: %s\n", i, inet_ntoa(addr));
            }
        }
        else if (remoteHost->h_addrtype == AF_NETBIOS)
        {
            printf("NETBIOS address was returned\n");
        }
    }

    return 0;
}
```

Windows Phone 8: This function is supported for Windows Phone Store apps on Windows Phone 8 and later.

Windows 8.1 and **Windows Server 2012 R2**: This function is supported for Windows Store apps on Windows 8.1, Windows Server 2012 R2, and later.

Requirements

Minimum supported client	Windows 8.1, Windows Vista [desktop apps | UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps | UWP apps]
Target Platform	Windows
Header	wsipv6ok.h (include Winsock2.h, Winsock.h)
Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

GetAddrInfoEx

GetAddrInfoW

WSAAsyncGetHostByName

Winsock Functions

Winsock Reference

addrinfo

addrinfoW

getaddrinfo

gethostbyaddr

gethostname

hostent

inet_addr

inet_addr macro (wsipv6ok.h)

10/22/2022 • 3 minutes to read • Edit Online

The **inet_addr** function converts a string containing an IPv4 dotted-decimal address into a proper address for the IN_ADDR structure.

Syntax

```
void inet_addr(
  [in]  a
);
```

Parameters

`[in] a`

A **NULL**-terminated character string representing a number expressed in the Internet standard "." (dotted) notation.

Return value

None

Remarks

The **inet_addr** function interprets the character string specified by the *cp* parameter. This string represents a numeric Internet address expressed in the Internet standard "." notation. The value returned is a number suitable for use as an Internet address. All Internet addresses are returned in IP's network order (bytes ordered from left to right). If you pass in " " (a space) to the **inet_addr** function, **inet_addr** returns zero.

On Windows Vista and later, the RtlIpv4StringToAddress function can be used to convert a string representation of an IPv4 address to a binary IPv4 address represented as an IN_ADDR structure. On Windows Vista and later, the RtlIpv6StringToAddress function can be used to convert a string representation of an IPv6 address to a binary IPv6 address represented as an **IN6_ADDR** structure.

Internet Addresses

Values specified using the "." notation take one of the following forms:

a.b.c.d a.b.c a.b a

When four parts are specified, each is interpreted as a byte of data and assigned, from left to right, to the 4 bytes of an Internet address. When an Internet address is viewed as a 32-bit integer quantity on the Intel architecture, the bytes referred to above appear as "d.c.b.a". That is, the bytes on an Intel processor are ordered from right to left.

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

INTERNET ADDRESS VALUE	MEANING

"4.3.2.16"	Decimal
"004.003.002.020"	Octal
"0x4.0x3.0x2.0x10"	Hexadecimal
"4.003.002.0x10"	Mix

The **inet_addr** function supports the decimal, octal, hexadecimal, and mixed notations for the string passed in the *cp* parameter.

> **Note** The following notations are only used by Berkeley software, and nowhere else on the Internet. For compatibility with Berkeley software, the **inet_addr** function also supports the additional notations specified below.

When a three-part address is specified, the last part is interpreted as a 16-bit quantity and placed in the right-most 2 bytes of the network address. This makes the three-part address format convenient for specifying Class B network addresses as "128.net.host''

When a two-part address is specified, the last part is interpreted as a 24-bit quantity and placed in the right-most 3 bytes of the network address. This makes the two-part address format convenient for specifying Class A network addresses as "net.host''.

When only one part is given, the value is stored directly in the network address without any byte rearrangement.

Windows Phone 8: This function is supported for Windows Phone Store apps on Windows Phone 8 and later.

Windows 8.1 and **Windows Server 2012 R2**: This function is supported for Windows Store apps on Windows 8.1, Windows Server 2012 R2, and later.

Examples

The following code example shows how to use the **inet_addr** function.

```
#define WIN32_LEAN_AND_MEAN

#include <winsock2.h>
#include <stdio.h>
#include <windows.h>


// need link with Ws2_32.lib
#pragma comment(lib, "Ws2_32.lib")

int __cdecl main(int argc, char **argv)
{

    //-----------------------------------------
    // Declare and initialize variables
    WSADATA wsaData;
    int iResult;

    unsigned long ulAddr = INADDR_NONE;

    // Validate the parameters
    if (argc != 2) {
        printf("usage: %s <IPv4 address>\n", argv[0]);
        printf("  inetaddr converts a string containing an\n");
        printf("  IPv4 address in one of the supported formats\n");
        printf("  to a unsigned long representing an IN_ADDR\n");
        printf("      %s 192.168.16.34\n", argv[0]);
        return 1;
    }
    // Initialize Winsock
    iResult = WSAStartup(MAKEWORD(2, 2), &wsaData);
    if (iResult != 0) {
        printf("WSAStartup failed: %d\n", iResult);
        return 1;
    }

//--------------------------------
// Call inet_addr(). If the call succeeds,
// the result variable will hold a IN_ADDR
    ulAddr = inet_addr(argv[1]);
    if ( ulAddr == INADDR_NONE ) {
        printf("inet_addr failed and returned INADDR_NONE\n");
        WSACleanup();
        return 1;
    }

    if (ulAddr == INADDR_ANY) {
        printf("inet_addr failed and returned INADDR_ANY\n");
        WSACleanup();
        return 1;
    }

    printf("inet_addr returned success\n");

    // Here we could implement code to retrieve each address and
    // print out the hex bytes
    // for(i=0, ptr= (Char*) &ulAddr; i < 4; i++, ptr++) {

    WSACleanup();
    return 0;
}
```

Requirements

Minimum supported client	Windows 8.1, Windows Vista [desktop apps | UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps | UWP apps]
Target Platform	Windows
Header	wsipv6ok.h (include Winsock2.h, Winsock.h)
Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

IN6_ADDR

IN_ADDR

InetNtop

RtlIpv4AddressToString

RtlIpv4AddressToStringEx

RtlIpv4StringToAddress

RtlIpv4StringToAddressEx

RtlIpv6AddressToString

RtlIpv6AddressToStringEx

RtlIpv6StringToAddress

RtlIpv6StringToAddressEx

Winsock Functions

Winsock Reference

inet_ntoa

inet_ntoa macro (wsipv6ok.h)

10/22/2022 • 2 minutes to read • Edit Online

The **inet_ntoa** function converts an (Ipv4) Internet network address into an ASCII string in Internet standard dotted-decimal format.

Syntax

```
void inet_ntoa(
  [in]  a
);
```

Parameters

`[in] a`

An in_addr structure that represents an Internet host address.

Return value

None

Remarks

The **inet_ntoa** function takes an Internet address structure specified by the *in* parameter and returns a **NULL**-terminated ASCII string that represents the address in "." (dot) notation as in "192.168.16.0", an example of an IPv4 address in dotted-decimal notation. The string returned by **inet_ntoa** resides in memory that is allocated by Windows Sockets. The application should not make any assumptions about the way in which the memory is allocated. The string returned is guaranteed to be valid only until the next Windows Sockets function call is made within the same thread. Therefore, the data should be copied before another Windows Sockets call is made.

The WSAAddressToString function can be used to convert a sockaddr structure containing an IPv4 address to a string representation of an IPv4 address in Internet standard dotted-decimal notation. The advantage of the **WSAAddressToString** function is that it supports both IPv4 and IPv6 addresses. Another advantage of the **WSAAddressToString** function is that there are both ASCII and Unicode versions of this function.

On Windows Vista and later, the RtlIpv4AddressToString function can be used to convert an IPv4 address represented as an IN_ADDR structure to a string representation of an IPv4 address in Internet standard dotted-decimal notation. On Windows Vista and later, the RtlIpv6AddressToString function can be used to convert an IPv6 address represented as an **IN6_ADDR** structure to a string representation of an IPv6 address.

Windows Phone 8: This function is supported for Windows Phone Store apps on Windows Phone 8 and later.

Windows 8.1 and **Windows Server 2012 R2**: This function is supported for Windows Store apps on Windows 8.1, Windows Server 2012 R2, and later.

Requirements

Minimum supported client	Windows 8.1, Windows Vista [desktop apps | UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps | UWP apps]
Target Platform	Windows
Header	wsipv6ok.h (include Winsock2.h, Winsock.h)
Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

IN6_ADDR

InetNtop

RtlIpv4AddressToString

RtlIpv4AddressToStringEx

RtlIpv4StringToAddress

RtlIpv4StringToAddressEx

RtlIpv6AddressToString

RtlIpv6AddressToStringEx

RtlIpv6StringToAddress

RtlIpv6StringToAddressEx

SOCKADDR

WSAAddressToString

Winsock Functions

Winsock Reference

in_addr

inet_addr

WSAAsyncGetHostByAddr macro (wsipv6ok.h)

10/22/2022 • 3 minutes to read • Edit Online

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

> **Note** The **WSAAsyncGetHostByAddr** function is not designed to provide parallel resolution of several addresses. Therefore, applications that issue several requests should not expect them to be executed concurrently. Alternatively, applications can start another thread and use the getnameinfo function to resolve addresses in an IP-version agnostic manner. Developers creating Windows Sockets 2 applications are urged to use the **getnameinfo** function to enable smooth transition to IPv6 compatibility.

Syntax

```
void WSAAsyncGetHostByAddr(
  [in]   a,
  [in]   b,
  [in]   c,
  [in]   d,
  [in]   e,
  [out]  f,
  [in]   g
);
```

Parameters

`[in] a`

Handle of the window that will receive a message when the asynchronous request completes.

`[in] b`

Message to be received when the asynchronous request completes.

`[in] c`

Pointer to the network address for the host. Host addresses are stored in network byte order.

`[in] d`

Length of the address, in bytes.

`[in] e`

Type of the address.

`[out] f`

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

`[in] g`

Size of data area for the *buf* parameter, in bytes.

Return value

None

Remarks

The **WSAAsyncGetHostByAddr** function is an asynchronous version of gethostbyaddr. It is used to retrieve the host name and address information that corresponds to a network address. Windows Sockets initiates the operation and returns to the caller immediately, passing back an opaque, asynchronous task handle that the application can use to identify the operation. When the operation is completed, the results (if any) are copied into the buffer provided by the caller and a message is sent to the application's window.

When the asynchronous operation has completed, the application window indicated by the *hWnd* parameter receives message in the *wMsg* parameter. The *wParam* parameter contains the asynchronous task handle as returned by the original function call. The high 16 bits of *lParam* contain any error code. The error code can be any error as defined in Winsock2.h. An error code of zero indicates successful completion of the asynchronous operation.

On successful completion, the buffer specified to the original function call contains a hostent structure. To access the members of this structure, the original buffer address is cast to a **hostent** structure pointer and accessed as appropriate.

If the error code is WSAENOBUFS, the size of the buffer specified by *buflen* in the original call was too small to contain all the resulting information. In this case, the low 16 bits of *lParam* contain the size of buffer required to supply all the requisite information. If the application decides that the partial data is inadequate, it can reissue the **WSAAsyncGetHostByAddr** function call with a buffer large enough to receive all the desired information (that is, no smaller than the low 16 bits of *lParam*).

The buffer specified to this function is used by Windows Sockets to construct a structure together with the contents of data areas referenced by members of the same hostent structure. To avoid the WSAENOBUFS error, the application should provide a buffer of at least MAXGETHOSTSTRUCT bytes (as defined in Winsock2.h).

The error code and buffer length should be extracted from the *lParam* using the macros **WSAGETASYNCERROR** and **WSAGETASYNCBUFLEN**, defined in Winsock2.h as:

```
#include <windows.h>

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

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

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]

Target Platform	Windows
Header	wsipv6ok.h (include Winsock2.h, Winsock.h)
Library	Ws2_32.lib
DLL	Ws2_32.dll

See also

WSACancelAsyncRequest

Winsock Functions

Winsock Reference

getaddrinfo

gethostbyaddr

getnameinfo

hostent

WSAAsyncGetHostByName macro (wsipv6ok.h)

10/22/2022 • 3 minutes to read • Edit Online

The **WSAAsyncGetHostByName** function asynchronously retrieves host information that corresponds to a host name.

> **Note** The **WSAAsyncGetHostByName** function is not designed to provide parallel resolution of several names. Therefore, applications that issue several requests should not expect them to be executed concurrently. Alternatively, applications can start another thread and use the getaddrinfo function to resolve names in an IP-version agnostic manner. Developers creating Windows Sockets 2 applications are urged to use the **getaddrinfo** function to enable smooth transition to IPv6 compatibility.

Syntax

```
void WSAAsyncGetHostByName(
  [in]   a,
  [in]   b,
  [in]   c,
  [out]  d,
  [in]   e
);
```

Parameters

`[in] a`

Handle of the window that will receive a message when the asynchronous request completes.

`[in] b`

Message to be received when the asynchronous request completes.

`[in] c`

Pointer to the null-terminated name of the host.

`[out] d`

Pointer to the data area to receive the hostent data. The data area must be larger than the size of a **hostent** structure because the specified data area is used by Windows Sockets to contain a **hostent** structure and all of the data referenced by members of the **hostent** structure. A buffer of MAXGETHOSTSTRUCT bytes is recommended.

`[in] e`

Size of data area for the *buf* parameter, in bytes.

Return value

None

Remarks

The **WSAAsyncGetHostByName** function is an asynchronous version of gethostbyname, and is used to retrieve host name and address information corresponding to a host name. Windows Sockets initiates the operation and returns to the caller immediately, passing back an opaque asynchronous task handle that which the application can use to identify the operation. When the operation is completed, the results (if any) are copied into the buffer provided by the caller and a message is sent to the application's window.

When the asynchronous operation has completed, the application window indicated by the *hWnd* parameter receives message in the *wMsg* parameter. The *wParam* parameter contains the asynchronous task handle as returned by the original function call. The high 16 bits of *lParam* contain any error code. The error code can be any error as defined in Winsock2.h. An error code of zero indicates successful completion of the asynchronous operation.

On successful completion, the buffer specified to the original function call contains a hostent structure. To access the elements of this structure, the original buffer address should be cast to a **hostent** structure pointer and accessed as appropriate.

If the error code is WSAENOBUFS, the size of the buffer specified by *buflen* in the original call was too small to contain all the resulting information. In this case, the low 16 bits of *lParam* contain the size of buffer required to supply all the requisite information. If the application decides that the partial data is inadequate, it can reissue the **WSAAsyncGetHostByName** function call with a buffer large enough to receive all the desired information (that is, no smaller than the low 16 bits of *lParam*).

The buffer specified to this function is used by Windows Sockets to construct a hostent structure together with the contents of data areas referenced by members of the same **hostent** structure. To avoid the WSAENOBUFS error, the application should provide a buffer of at least MAXGETHOSTSTRUCT bytes (as defined in Winsock2.h).

The error code and buffer length should be extracted from the *lParam* using the macros **WSAGETASYNCERROR** and **WSAGETASYNCBUFLEN**, defined in Winsock2.h as:

```
#include <windows.h>

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

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

WSAAsyncGetHostByName is guaranteed to resolve the string returned by a successful call to gethostname.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	wsipv6ok.h (include Winsock2.h, Winsock.h)
Library	Ws2_32.lib

DLL	Ws2_32.dll

See also

WSACancelAsyncRequest

Winsock Functions

Winsock Reference

getaddrinfo

gethostbyname

getnameinfo

hostent

wsnwlink.h header

10/22/2022 • 2 minutes to read • Edit Online

This header is used by Windows Sockets 2. For more information, see:

- Windows Sockets 2

wsnwlink.h contains the following programming interfaces:

Structures

IPX_ADDRESS_DATA The IPX_ADDRESS_DATA structure provides information about a specific adapter to which IPX is bound. Used in conjunction with getsockopt function calls that specify IPX_ADDRESS in the optname parameter.
IPX_NETNUM_DATA The IPX_NETNUM_DATA structure provides information about a specified IPX network number. Used in conjunction with getsockopt function calls that specify IPX_GETNETINFO in the optname parameter.
IPX_SPXCONNSTATUS_DATA The IPX_SPXCONNSTATUS_DATA structure provides information about a connected SPX socket.

IPX_ADDRESS_DATA structure (wsnwlink.h)

10/22/2022 • 2 minutes to read • Edit Online

The **IPX_ADDRESS_DATA** structure provides information about a specific adapter to which IPX is bound. Used in conjunction with getsockopt function calls that specify IPX_ADDRESS in the *optname* parameter.

Syntax

```
typedef struct _IPX_ADDRESS_DATA {
  INT     adapternum;
  UCHAR   netnum[4];
  UCHAR   nodenum[6];
  BOOLEAN wan;
  BOOLEAN status;
  INT     maxpkt;
  ULONG   linkspeed;
} IPX_ADDRESS_DATA, *PIPX_ADDRESS_DATA;
```

Members

`adapternum`

0-based adapter number.

`netnum[4]`

IPX network number for the associated adapter.

`nodenum[6]`

IPX node address for the associated adapter.

`wan`

Specifies whether the adapter is on a wide area network (WAN) link. When **TRUE**, the adapter is on a WAN link.

`status`

Specifies whether the WAN link is up. **FALSE** indicates that the WAN link is up or the adapter is not on a WAN. Compare with the **wan** member to determine the meaning.

`maxpkt`

Maximum allowable packet size, excluding the IPX header.

`linkspeed`

Link speed, returned in 100 byte-per-second increments. For example, a 9600 byte-per-second link would return a value of 96.

Remarks

Adapter numbers are base zero, so if there are eight adapters on a given computer, they are numbered 0-7. To determine the number of adapters present on the computer, call the getsockopt function with IPX_MAX_ADAPTER_NUM.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	wsnwlink.h

See also

getsockopt

IPX_NETNUM_DATA structure (wsnwlink.h)

10/22/2022 • 2 minutes to read • Edit Online

The **IPX_NETNUM_DATA** structure provides information about a specified IPX network number. Used in conjunction with getsockopt function calls that specify IPX_GETNETINFO in the *optname* parameter.

Syntax

```
typedef struct _IPX_NETNUM_DATA {
  UCHAR  netnum[4];
  USHORT hopcount;
  USHORT netdelay;
  INT    cardnum;
  UCHAR  router[6];
} IPX_NETNUM_DATA, *PIPX_NETNUM_DATA;
```

Members

`netnum[4]`

IPX network number for which information is being requested.

`hopcount`

Number of hops to the IPX network being queried, in machine order.

`netdelay`

Network delay tick count required to reach the IPX network, in machine order.

`cardnum`

Adapter number used to reach the IPX network. The adapter number is zero based, such that if eight adapters are on a given computer, they are numbered 0-7.

`router[6]`

Media Access Control (MAC) address of the next-hop router in the path between the computer and the IPX network. This value is zero if the computer is directly attached to the IPX network.

Remarks

If information about the IPX network is in the computer's IPX cache, the call will return immediately. If not, RIP requests are used to resolve the information.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]

Header	wsnwlink.h

See also

getsockopt

IPX_SPXCONNSTATUS_DATA structure (wsnwlink.h)

10/22/2022 • 2 minutes to read • Edit Online

The **IPX_SPXCONNSTATUS_DATA** structure provides information about a connected SPX socket. Used in conjunction with getsockopt function calls that specify IPX_SPXGETCONNECTIONSTATUS in the *optname* parameter. All numbers in **IPX_SPXCONNSTATUS_DATA** are in Novell (high-low) order.

Syntax

```
typedef struct _IPX_SPXCONNSTATUS_DATA {
  UCHAR  ConnectionState;
  UCHAR  WatchDogActive;
  USHORT LocalConnectionId;
  USHORT RemoteConnectionId;
  USHORT LocalSequenceNumber;
  USHORT LocalAckNumber;
  USHORT LocalAllocNumber;
  USHORT RemoteAckNumber;
  USHORT RemoteAllocNumber;
  USHORT LocalSocket;
  UCHAR  ImmediateAddress[6];
  UCHAR  RemoteNetwork[4];
  UCHAR  RemoteNode[6];
  USHORT RemoteSocket;
  USHORT RetransmissionCount;
  USHORT EstimatedRoundTripDelay;
  USHORT RetransmittedPackets;
  USHORT SuppressedPacket;
} IPX_SPXCONNSTATUS_DATA, *PIPX_SPXCONNSTATUS_DATA;
```

Members

`ConnectionState`

Specifies the connection state.

`WatchDogActive`

Specifies whether watchdog capabilities are active.

`LocalConnectionId`

Specifies the local connection ID.

`RemoteConnectionId`

Specifies the remote connection ID.

`LocalSequenceNumber`

Specifies the local sequence number.

`LocalAckNumber`

Specifies the local acknowledgment (ACK) number.

`LocalAllocNumber`

Specifies the local allocation number.

`RemoteAckNumber`

Specifies the remote acknowledgment (ACK) number.

`RemoteAllocNumber`

Specifies the remote allocation number.

`LocalSocket`

Specifies the local socket.

`ImmediateAddress[6]`

Specifies the IPX address to which the local computer is attached.

`RemoteNetwork[4]`

Specifies the network to which the remote host is attached.

`RemoteNode[6]`

Specifies the remote node.

`RemoteSocket`

Specifies the remote socket.

`RetransmissionCount`

Specifies the number of retransmissions.

`EstimatedRoundTripDelay`

Specifies the estimated round trip–time, in milliseconds, delay for a given packet.

`RetransmittedPackets`

Specifies the number of retransmitted packets on the socket.

`SuppressedPacket`

Specifies the number of suppressed packets on the socket.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	wsnwlink.h

See also

getsockopt

wsrm.h header

10/22/2022 • 2 minutes to read • Edit Online

This header is used by Windows Sockets 2. For more information, see:

- Windows Sockets 2

wsrm.h contains the following programming interfaces:

Structures

RM_FEC_INFO The RM_FEC_INFO structure specifies settings for using forward error correction (FEC) with Reliable Multicast. This structure is used with the RM_USE_FEC socket option.
RM_RECEIVER_STATS Provides statistical information for a Reliable Multicast receiver session. This structure is used with the RM_RECEIVER_STATISTICS socket option.
RM_SEND_WINDOW The RM_SEND_WINDOW structure specifies the Reliable Multicast send window. This structure is used with the RM_RATE_WINDOW_SIZE socket option.
RM_SENDER_STATS Provides statistical information for a Reliable Multicast sender session. This structure is used with the RM_SENDER_STATISTICS socket option.

Enumerations

eWINDOW_ADVANCE_METHOD The eWINDOW_ADVANCE_METHOD enumeration specifies the window advance mode used for Reliable Multicast.

eWINDOW_ADVANCE_METHOD enumeration (wsrm.h)

10/22/2022 • 2 minutes to read • Edit Online

The **eWINDOW_ADVANCE_METHOD** enumeration specifies the window advance mode used for Reliable Multicast.

Syntax

```
typedef enum eWINDOW_ADVANCE_METHOD {
  E_WINDOW_ADVANCE_BY_TIME = 1,
  E_WINDOW_USE_AS_DATA_CACHE
} ;
```

Constants

`E_WINDOW_ADVANCE_BY_TIME` Value: *1* Window advances based on time. This is the default mode.
`E_WINDOW_USE_AS_DATA_CACHE` Use the receive window as a data cache.

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2003 [desktop apps only]
Header	wsrm.h

See also

Reliable Multicast Programming

Socket Options

RM_FEC_INFO structure (wsrm.h)

10/22/2022 • 2 minutes to read • Edit Online

The **RM_FEC_INFO** structure specifies settings for using forward error correction (FEC) with Reliable Multicast. This structure is used with the RM_USE_FEC socket option.

Syntax

```
typedef struct _RM_FEC_INFO {
  USHORT  FECBlockSize;
  USHORT  FECProActivePackets;
  UCHAR   FECGroupSize;
  BOOLEAN fFECOnDemandParityEnabled;
} RM_FEC_INFO;
```

Members

`FECBlockSize`

Maximum number of packets that can be sent for any group, including original data and parity packets. Maximum and default value is 255.

`FECProActivePackets`

Number of packets to send proactively with each group. Use this option when the network is dispersed, and upstream NAK requests would have an impact on throughput.

`FECGroupSize`

Number of packets to be treated as one group for the purpose of computing parity packets. Group size must be a power of two. In lossy networks, keep the group size relatively small.

`fFECOnDemandParityEnabled`

Specifies whether the sender is enabled for sending parity repair packets. When **TRUE**, receivers should only request parity repair packets.

Remarks

The RM_USE_FEC socket option notifies the Reliable Multicast sender to apply forward error correction techniques to send repair data. there are three modes of using forward error correction:

1. Pro-active parity packets only
2. OnDemand parity packets only
3. Both pro-active and OnDemand parity packets

Since the use of this structure implies the need for forward error correction, either the **FECProActivePackets** or **fFECOnDemandParityEnabled** member must be nonzero, otherwise the function call fails.

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2003 [desktop apps only]
Header	wsrm.h

See also

RM_USE_FEC

Reliable Multicast Programming

Socket Options

RM_RECEIVER_STATS structure (wsrm.h)

10/22/2022 • 2 minutes to read • Edit Online

The **RM_RECEIVER_STATS** structure provides statistical information for a Reliable Multicast receiver session. This structure is used with the RM_RECEIVER_STATISTICS socket option.

Syntax

```
typedef struct _RM_RECEIVER_STATS {
  ULONGLONG NumODataPacketsReceived;
  ULONGLONG NumRDataPacketsReceived;
  ULONGLONG NumDuplicateDataPackets;
  ULONGLONG DataBytesReceived;
  ULONGLONG TotalBytesReceived;
  ULONGLONG RateKBitsPerSecOverall;
  ULONGLONG RateKBitsPerSecLast;
  ULONGLONG TrailingEdgeSeqId;
  ULONGLONG LeadingEdgeSeqId;
  ULONGLONG AverageSequencesInWindow;
  ULONGLONG MinSequencesInWindow;
  ULONGLONG MaxSequencesInWindow;
  ULONGLONG FirstNakSequenceNumber;
  ULONGLONG NumPendingNaks;
  ULONGLONG NumOutstandingNaks;
  ULONGLONG NumDataPacketsBuffered;
  ULONGLONG TotalSelectiveNaksSent;
  ULONGLONG TotalParityNaksSent;
} RM_RECEIVER_STATS;
```

Members

`NumODataPacketsReceived`

Type: **ULONGULONG**

The number of original data (ODATA) sequences received.

`NumRDataPacketsReceived`

Type: **ULONGULONG**

The number of repair data (RDATA) sequences received.

`NumDuplicateDataPackets`

Type: **ULONGULONG**

The number of duplicate sequences received.

`DataBytesReceived`

Type: **ULONGULONG**

The number of data bytes received.

`TotalBytesReceived`

Type: **ULONGULONG**

The total bytes received, consisting of source path message (SPM), original data (ODATA) and repair data (RDATA) sequences.

`RateKBitsPerSecOverall`

Type: **ULONGULONG**

An internally calculated receive rate from the beginning of the session, in kilobits per second.

`RateKBitsPerSecLast`

Type: **ULONGULONG**

The receive rate for the period specified by INTERNAL_RATE_CALCULATION_FREQUENCY.

`TrailingEdgeSeqId`

Type: **ULONGULONG**

The oldest sequence identifier in the receive window.

`LeadingEdgeSeqId`

Type: **ULONGULONG**

The newest sequence identifier in the receive window.

`AverageSequencesInWindow`

Type: **ULONGULONG**

The average number of sequences in the receive window.

`MinSequencesInWindow`

Type: **ULONGULONG**

The minimum number of sequences in the receive window.

`MaxSequencesInWindow`

Type: **ULONGULONG**

The maximum number of sequences in the receive window.

`FirstNakSequenceNumber`

Type: **ULONGULONG**

The sequence number for the first outstanding negative acknowledgment (NAK).

`NumPendingNaks`

Type: **ULONGULONG**

The number of sequences awaiting a NAK confirmation.

`NumOutstandingNaks`

Type: **ULONGULONG**

The number of sequences awaiting repair data (RDATA).

`NumDataPacketsBuffered`

Type: **ULONGULONG**

The number of packets currently buffered.

`TotalSelectiveNaksSent`

Type: **ULONGULONG**

The number of selective NAKs sent this session.

`TotalParityNaksSent`

Type: **ULONGULONG**

The number of parity NAKs sent this session.

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2003 [desktop apps only]
Header	wsrm.h

See also

RM_SENDER_STATS

Reliable Multicast Programming

Socket Options

RM_SEND_WINDOW structure (wsrm.h)

10/22/2022 • 2 minutes to read • Edit Online

The **RM_SEND_WINDOW** structure specifies the Reliable Multicast send window. This structure is used with the RM_RATE_WINDOW_SIZE socket option.

Syntax

```
typedef struct _RM_SEND_WINDOW {
  ULONG RateKbitsPerSec;
  ULONG WindowSizeInMSecs;
  ULONG WindowSizeInBytes;
} RM_SEND_WINDOW;
```

Members

`RateKbitsPerSec`

Transmission rate for the send window, in kilobits per second.

`WindowSizeInMSecs`

Window size for the send window, in milliseconds.

`WindowSizeInBytes`

Window size for the session, in bytes.

Remarks

Any combination of the three available members may be set for a given socket option call. For example, one, any two, or all three members may be specified during a setsockopt function call. Regardless of settings, Windows enforces the following ratio: **TransmissionRate** == (**WindowSizeBytes**/**WindowSizeMSecs**) * 8. As such, setting any two parameters effectively sets the third to ensure optimum performance.

The combination of these members can affect the resources used on a PGM sender's computer. For example, a large transmission rate value combined with a large window size results in more required buffer space.

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2003 [desktop apps only]
Header	wsrm.h

See also

IPPROTO_RM Socket Options

Reliable Multicast Programming

Socket Options

setsockopt

RM_SENDER_STATS structure (wsrm.h)

10/22/2022 • 2 minutes to read • Edit Online

The **RM_SENDER_STATS** structure provides statistical information for a Reliable Multicast sender session. This structure is used with the RM_SENDER_STATISTICS socket option.

Syntax

```
typedef struct _RM_SENDER_STATS {
  ULONGLONG DataBytesSent;
  ULONGLONG TotalBytesSent;
  ULONGLONG NaksReceived;
  ULONGLONG NaksReceivedTooLate;
  ULONGLONG NumOutstandingNaks;
  ULONGLONG NumNaksAfterRData;
  ULONGLONG RepairPacketsSent;
  ULONGLONG BufferSpaceAvailable;
  ULONGLONG TrailingEdgeSeqId;
  ULONGLONG LeadingEdgeSeqId;
  ULONGLONG RateKBitsPerSecOverall;
  ULONGLONG RateKBitsPerSecLast;
  ULONGLONG TotalODataPacketsSent;
} RM_SENDER_STATS;
```

Members

`DataBytesSent`

Type: **ULONGULONG**

Number of client data bytes sent out.

`TotalBytesSent`

Type: **ULONGULONG**

Total bytes sent, consisting of source path message (SPM), original data (ODATA) and repair data (RDATA) sequences.

`NaksReceived`

Type: **ULONGULONG**

Number of NAKs received.

`NaksReceivedTooLate`

Type: **ULONGULONG**

Number of NAKs received after the send window advanced beyond the NAK'ed sequence.

`NumOutstandingNaks`

Type: **ULONGULONG**

Number of NAKs for which responses have not been sent.

`NumNaksAfterRData`

Type: **ULONGULONG**

Number of NAKs after repair data (RDATA) sequences were sent for which responses have not been sent.

`RepairPacketsSent`

Type: **ULONGULONG**

Number of repair data (RDATA) packets sent.

`BufferSpaceAvailable`

Type: **ULONGULONG**

Number of partial messages dropped.

`TrailingEdgeSeqId`

Type: **ULONGULONG**

Oldest sequence identifier in the send window.

`LeadingEdgeSeqId`

Type: **ULONGULONG**

Newest sequence identifier in the send window.

`RateKBitsPerSecOverall`

Type: **ULONGULONG**

Internally calculated send rate from the beginning of the session, in kilobits per second.

`RateKBitsPerSecLast`

Type: **ULONGULONG**

Send rate for the period specified by INTERNAL_RATE_CALCULATION_FREQUENCY.

`TotalODataPacketsSent`

Type: **ULONGULONG**

Total original data (ODATA) packets sent.

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2003 [desktop apps only]
Header	wsrm.h

See also

RM_RECEIVER_STATS

Reliable Multicast Programming

Socket Options

wtypesbase.h header

10/22/2022 • 2 minutes to read • Edit Online

This header is used by multiple technologies. For more information, see:

- Component Object Model (COM)
- Windows Sockets 2

wtypesbase.h contains the following programming interfaces:

Functions

OLESTR Transforms string literals into Unicode strings.

Structures

BLOB The BLOB (wtypesbase.h) structure, derived from Binary Large Object, contains information about a block of data.
COAUTHIDENTITY Contains a user name and password.
COAUTHINFO Contains the authentication settings used while making a remote activation request from the client computer to the server computer.
SECURITY_ATTRIBUTES The SECURITY_ATTRIBUTES structure contains the security descriptor for an object and specifies whether the handle retrieved by specifying this structure is inheritable.

Enumerations

CLSCTX Values that are used in activation calls to indicate the execution contexts in which an object is to be run.
MSHCTX Specifies the destination context, which is the process in which the unmarshaling is to be done.

MSHLFLAGS	Specifies why the marshaling is to be done.

BLOB structure (wtypesbase.h)

10/22/2022 • 2 minutes to read • Edit Online

The **BLOB** structure, derived from Binary Large Object, contains information about a block of data.

Syntax

```
typedef struct tagBLOB {
  ULONG cbSize;
  BYTE  *pBlobData;
} BLOB, *LPBLOB;
```

Members

`cbSize`

Size of the block of data pointed to by **pBlobData**, in bytes.

`pBlobData`

Pointer to a block of data.

Remarks

The structure name **BLOB** comes from the acronym BLOB, which stands for Binary Large Object.

This structure does not describe the nature of the data pointed to by **pBlobData**.

> **Note** Windows Sockets defines a similar **BLOB** structure in Wtypes.h. Using both header files in the same source code file creates redefinition–compile time errors.

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	wtypesbase.h (include Wtypes.h, Nspapi.h, Winsock2.h, Wtypes.h, Nspapi.h, Winsock2.h)

See also

Bluetooth and BLOB

SERVICE_INFO