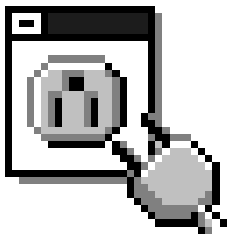
Windows Sockets 2 Application Programming Interface

An Interface for Transparent Network Programming Under Microsoft Windows™

Revision 2.0
Dec 8, 1994



Winsock 2

Preliminary
Subject to Change Without Notice

Draft

Microsoft, Intel and JSB disclaim all warranties and liability for the use of this document and the information contained herein, and assumes no responsibility for any errors which may appear in this document. Microsoft, Intel and JSB make no warranty or license regarding the relationship of this document and the information contained herein to the intellectual property rights of any party. Microsoft, Intel and JSB make no commitment to update the information contained herein.

Table of Contents

Introduction **4**

Summary of Additions and Changes **4**
Support for multiple transports simultaneously 4
Shared Sockets 5
Overlapped I/O and Event Objects 5
Socket Groups 7
Enhanced Functionality During Connection Setup 7
Quality of Service 7
New Socket Option Summary 9

New API Functions **10**
WSAAccept() 10
WSAAsyncSelect() 15
WSAConnect() 21
WSADuplicateSocket() 26
WSAEnumProtocols() 28
WSAEnumNetworkEvents() 31
WSAEventSelect() 33
WSARecv() 38
WSARecvfrom() 41
WSASend() 45
WSASendto() 48
WSASocket() 52
WSACreateEvent 54
WSACloseEvent 55
WSAWaitForMultipleEvents 56
WSASetEvent 58
WSAResetEvent 59
WSAGetOverlappedResult 60
WSAGetQoSByName 62

Winsock 2.0 Header File - Winsock2.h **63**

Introduction

The pages which follow contain new APIs that are proposed for Winsock 2. Except where indicated, they are applicable to both 16 and 32 bit programming environments. These APIs were developed jointly by the Generic API Extensions and Operating Framework functionality groups from within the Winsock Forum.

Summary of Additions and Changes

The paragraphs which follow summarize the major changes and additions in going from Winsock 1.1 to Winsock 2.

Support for multiple transports simultaneously

Winsock 1.1 implementations are all vendor-specific since no standard interface has been defined for use between Winsock.DLL and protocol stacks. Winsock 2 changes the model by defining such an interface and allowing multiple stacks from multiple vendors to be accessed simultaneously from a single Winsock2 DLL. Furthermore, Winsock 2 support is not limited to TCP/IP protocol stacks as is the case for Winsock 1.1.

This is accomplished by creating a formal Service Provider Interface Specification, which exists under separate cover. Included in the SPI document is a definition for a set of functions provided in the Winsock2 DLL for the use of transport providers in installing their transport and making them available via Winsock. This obviates the need for providers to deal with the differences between INI files in Windows 3 environments and the registry in Windows 95 and NT.

Some of the elements in the **WSAData** structure (obtained via a call to **WSAStartup()**) should now be ignored since they no longer apply to a single vendor's stack. These include: *iMaxSockets, iMaxUdpDg,* and *lpVendorInfo*. Two new socket options are introduced to supply provider-specific information: SO_MAX_MSG_SIZE (replaces the iMaxUdpDg element) and PVD_CONFIG (allows any other provider-specific configuration to occur).

An application may use **WSAEnumProtocols()** to discover which transport providers are present and learn information about each as contained in an associated PROTOCOL_INFO struct. Whereas in Winsock 1 there was a small number of well-known socket types and protocol identifiers, the focus will shift for Winsock 2. The existing socket type and protocol identifiers will be retained for compatibility reasons, but many new address family, socket type and protocol values are expected to appear which are unique but not necessarily well known. Applications that desire to be independent of particular protocols are encouraged to examine the PROTOCOL_INFO structure associated with each available transport and select those that offer the required communications attributes (e.g. message vs. byte-stream oriented, reliable vs. unreliable, etc.). Having found one or more transports with suitable attributes, it really won't matter which particular socket type or protocol values are associated with the transport. These values are simply copied out of the PROTOCOL_INFO struct and used as parameters to the **socket()** or **WSASocket()** function.

It is anticipated that a clearinghouse will be established for obtaining unique identifiers for new address families, socket types and protocols. FTP and Web servers will supply current identifier/value mappings, and email can be used to obtain new ones.

Restrictions on select()

In Winsock 2 the FD_SET supplied to the select() function will be constrained to contain sockets from a **single** service provider. This restriction allows the Win32 implementations of Winsock2.DLL to be much simpler than they would otherwise be since all blocking behavior may now be implemented by the transport provider directly. (Unfortunately not true for Win16!) This does not in any way restrict an application from having multiple sockets open using multiple providers. When non-blocking operations are preferred the **WSAAsyncSelect()** function is the solution. Since it takes a socket handle as an input parameter, it doesn't matter what provider is associated with the socket.

When an application needs to block waiting for I/O to occur on a set of sockets which spans multiple providers, the recommended solution is to use **WSAWaitForMultipleEvents()**. The application may also which to take advantage of the **WSAEventSelect()** function which also allows the FD_XXX network events to be associated with an event object and handled from within the event object paradigm.

Shared Sockets

WSADuplicateSocket() is introduced to enable socket sharing by creating a shared socket for a target task (which could be the same task) with respect to a local, existing socket. The new shared socket thus created is only meaningful within the context of the target task. This mechanism is designed to be appropriate for both single-threaded version of Windows (such as Windows 3.1) and preemptive multithreaded versions of Windows (such as Windows 95 and NT).

Overlapped I/O and Event Objects

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

For receiving, application's use **WSARecv()** or **WSARecvFrom()** to supply buffers into which data is to be received prior to the time when data is received by the network. As data arrives, the network places it directly into the application's buffer and thereby avoids the copy operation that would otherwise occur at the time the **recv()** or **recvfrom()** function is invoked. Note that if data arrives when no receive buffers have been posted by the application, the network resorts to the familiar synchronous style of operation where the incoming data is buffered internally until such time as the application issues a receive call and thereby supplies a buffer into which the data may be copied. An exception to this would be if the application used setsockopt() to set the size of the receive buffer to zero. In this instance, data received without a receive buffer being posted would be lost.

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

Overlapped send and receive calls return immediately and the network provides a subsequent indication when send buffers have been consumed or when receive buffers are full. Also, both send and receive operations can be overlapped. The receive functions may be invoked multiple times to post receive buffers in preparation for incoming data, and the send functions may be invoked multiple times to queue up multiple buffers to be sent. Note that while the application can rely upon a series of overlapped send buffers being sent in the order supplied, the corresponding completion indications may occur in a different order. Likewise, on the receiving side, buffers will be filled in the order they are supplied but the completion indications may occur in a different order.

Event Objects as an Underpinning for Completion Indication

Introducing overlapped I/O requires a mechanism for applications to unambiguously associate send and receive requests with their subsequent completion indications. The selected mechanism utilizes event objects which are modeled after Win32 events. Applications use WSACreateEvent () to obtain an async event handle which may then be supplied as a required parameter to the asynchronous versions of send and receive calls (WSASend(), WSASendTo(), WSARecv(), WSARecvFrom()). The event object, which is cleared when first created, is set by the network when the associated overlapped I/O operation has completed (either successfully or with errors).

In order to provide applications with appropriate levels of flexibility, several options are available for receiving completion indications. These include: waiting on (i.e. blocking on) event objects, polling event objects, and callbacks (for 16 bit environments) or asynchronous procedure calls (for 32 bit environments).

Blocking and waiting for Completion Indication -
Applications may also choose to block while waiting for one or more async events to become set using **WSAWaitForMultipleEvents()**. In Win16 implementations, this will utilize a blocking hook as is currently provided for standard blocking socket operations. In Win32 implementations, the process or thread will be truly blocked. Since Winsock 2 event objects are implemented as Win32 events, the native Win32 function **WaitForMultipleObjects()** may also be used for this purpose. This is especially useful if the thread needs to wait on both socket and non-socket events.

Polling for Completion Indication -
Applications which prefer not to block may use **WSAGetOverlappedResults()** to poll for the completion status associated with any particular event object. This function will indicate both whether or not the overlapped operation has completed, and error status when completion has occurred.

Using callbacks or APCs -
The functions used to initiate overlapped I/O (**WSASend, WSASendTo, WSARecv, WSARecvFrom**) all take *lpCompletionRoutine* as an input parameter. This is a pointer to an application-specified function that is called when the overlapped I/O operation has completed (successfully or otherwise). In Win16 environments, callback functions may be invoked in what is essentially interrupt context. Consequently, applications have a very limited set of Windows and runtime library function calls that can be safely made. Network transports will allow send and receive operations to be called within the context of the callback function.

In Windows 95 and NT, the completion function will occur as an asynchronous procedure call (APC) and requires that the thread be in an alertable wait state such as can occur with the function **WSAWaitForMultipleEvents()**.

Relationship of WSAGetXByY() asynchronous task handles to event objects -
The Winsock 1.1 spec includes a number of asynchronous database access routines known collectively as the WSAGetxByY functions. The return value for these is referred to as an "asynchronous task handle". In Winsock 2, these asynchronous task handles are, in fact, event objects. As such they can be waited on by either using **WSAWaitForMultipleEvents()** (or in Win32 by using **WaitForMultipleObjects()**), or they can be polled with **WSAGetOverlappedResults()**.

Editor's Note:
This represents an exception to the normal way of doing things in a couple of areas. First, since these routines also generate a Windows message to indicate completion, the convention of allowing only a single completion indication mechanism in place at one time is violated. Secondly, while the established notion has been that applications create event objects and then explicitly associate them with some indication, this is not followed here since the function invocation causes an event object to come into existence as a side effect. The application may explicitly destroy the event object after the routine completes using WSACloseEvent(), or it may allow the Winsock 2 DLL to just recycle the event object at some future (unspecified) time as is the case for Winsock 1.1's asynchronous task handles.

If we felt that these WSAGetXbyY() routines had a long life ahead of them, we would be a lot more bothered by these inconsistencies than we are. But we hope that their use is soon superseded by the much more capable Winsock 2 name resolution routines. This being the case, we prefer to note but ignore the inconsistencies pointed out above.

Socket Groups

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

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

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

Enhanced Functionality During Connection Setup

WSAAccept () allows an application to obtain caller information before deciding whether or not to accept an incoming connection request. This is done via a callout to an application-supplied condition function. User-to-user data may be specified in **WSAConnect ()** and/or the condition function of **WSAAccept ()** to be transferred to the peer during connection establishment, provided this feature is supported by the service provider.

Quality of Service

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

Flow specs describe a set of characteristics about a proposed connection-oriented, unidirectional flow through the network. An application may associate a pair of flow specs with a socket at the time a connection request is made. Flow specs indicate parametrically what level of service is required and also stipulate whether the application is willing to be flexible if the requested level of service is not available. After a connection is established, the application may retrieve the flow specs associated with the socket and examine the contents to discover the level of service that the network is willing and/or able to provide. If the service provided is not acceptable, the application may close the socket and take whatever action is appropriate (e.g. scale back and ask for a lower quality of service, try again later, notify the user and exit, etc.)

Even after a flow is established, conditions in the network may change resulting in a reduction (or increase) in the available service level. A notification mechanism is included which utilizes the usual Winsock 2 notification techniques to indicate to the application that QOS levels have changed. The app should again retrieve the corresponding flow specs and examine them in order to discover what aspect of the service level has changed.

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

1. Network bandwidth utilization - The manner in which the application's traffic will be injected into the network. This includes specifications for average bandwidth utilization, peak bandwidth, and maximum burst duration.

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

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

4. Cost - This is a place holder for a future time when a meaningful cost metric can be determined.

5. Provider-specific parameters - The flow spec itself can be extended in ways that are particular to specific providers, and the assumed provider can be identified.

An application indicates its desire for a non-default flow spec at the time a connection request is made (see **WSAConnect** () and **WSAAccept**()). Since establishing a flow spec'd connection is likely to involve cooperation and/or negotiation between intermediate routers and hosts, the results of a flow spec request cannot be determined until after the connection operation is fully completed. After this time, the application may use **getsockopt**() to retrieve the resulting flow spec structure so that it can determine what the network was willing and/or able to supply.

The Flow Spec Structures

The Winsock 2 flow spec structure is defined in Winsock2.h and is reproduced here.

```
typedef enum
{
       GuaranteedService,
       BestEffortService
} GUARANTEE;

typedef struct _flowparams
{
       int64          AverageBandwith;// In Bytes/sec
       int64          PeakBandwidth;  // In Bytes/sec
       int64          BurstLength;    // In microseconds
       int64          Latency;        // In microseconds
       int64          DelayVariation;// In microseconds
       GUARANTEE      levelOfGuarantee;// Guaranteed or
                                      // Best Effort
       int32          CostOfCall;     // Reserved for future
                                      // use, must be set to 0
       int32          SizePSP;        // Length of provider
                                      // specific parameters
       UCHAR   ProviderSpecificParams[1];// provider specific
                                      // parameters
} FLOWPARAMS;

typedef struct _QualityOfService
{
       FLOWPARAMS     ForwardFP;      // Caller(Initiator) to callee
       FLOWPARAMS     BackwardFP;     // Callee to caller
} QOS, FAR * LPQOS;
```

Default Values

A default flow spec is associated with each eligible socket at the time it is created. Field values for this default flow spec are indicated below. In all cases these values indicate that no particular flow

characteristics are being requested from the network. Applications only need to modify values those fields which they are interested in, but must be aware that there exists some coupling between fields.

```
AverageBandwidth =     0, not specified
PeakBandwidth =        0, not specified
BurstLength =          0, not specified
Latency =              0, not specified
DelayVariation =       0, not specified
LevelOfGuarantee =     BEST_EFFORT
CostOfCall =           0, reserved for future use
ProviderSpecificParams = 0, none supplied
```

New Socket Option Summary

The new socket options proposed for Winsock2 are summarized in the following table.

Value	Type	Meaning	Default	Note
SO_MAX_MSG_SIZE	int	Maximum size of a message for message-oriented socket types. Has no meaning for stream-oriented sockets.	Implementation dependent	get only
SO_FLOWSPEC	char FAR *	The flow spec of this socket.	NULL	get only
SO_GROUP_ID	GROUP	The identifier of the group to which this socket belongs.	NULL	get only
SO_GROUP_FLOWSPEC	char FAR *	The flow spec of the socket group to which this socket belongs.	NULL	get only
SO_GROUP_PRIORITY	int	The relative priority for sockets that are part of a socket group.	0	
SO_PROTOCOL_INFO	struct PROTOCOL_ INFO	Description of protocol info for protocol that is bound to this socket.	protocol dependent	get only
PVD_CONFIG	char FAR *	An opaque data structure object containing configuration information of the service provider.	Implementation dependent	

New API Functions

WSAAccept()

Description Conditionally accept a connection based on the return value of a condition function, and optionally create and/or join a socket group.

#include <winsock2.h>

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

s	A descriptor identifying a socket which is listening for connections after a **listen**().
addr	An optional pointer to a buffer which receives the address of the connecting entity, as known to the communications layer. The exact format of the *addr* argument is determined by the address family established when the socket was created.
addrlen	An optional pointer to an integer which contains the length of the address *addr*.
lpfnCondition	The procedure instance address of the optional, application-supplied condition function which will make an accept/reject decision based on the caller information passed in as parameters, and optionally create and/or join a socket group by assigning an appropriate value to the result parameter *g* of this function.
dwCallbackData	The callback data passed back to the application as a condition function parameter. This parameter is not interpreted by Winsock.

Remarks This routine extracts the first connection on the queue of pending connections on *s*, and checks it against the condition function, provided the condition function is specified (i.e., not NULL). If the condition function returns CF_ACCEPT, this routine creates a new socket with the same properties as *s* and returns a handle to the new socket, and then optionally creates and/or joins a socket group based on the value of the result parameter *g* in the condition function. If the condition function returns CF_REJECT, this routine rejects the connection request. The condition function runs in the same thread as this routine does, and should return as soon as possible. If the decision cannot be made immediately, the condition function should return CF_DEFER to indicate that no decision has been made, and no action about this connection request should be taken by the service provider. When the application is ready to take action on the connection request, it may invoke **WSAAccept()** again and return either CF_ACCEPT or CF_REJECT as a return value from the condition function.

For synchronous sockets which remain in the (default) blocking mode, if no pending connections are present on the queue, **WSAAccept()** blocks the caller until a connection is present. For synchronous sockets in a non-blocking mode or for overlapped sockets, if this function is called when no pending connections are present on the queue, **WSAAccept()** returns an error as described below. The accepted socket may not be used to accept more connections. The original socket remains open.

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

The prototype of the condition function is as follows:

```
int PASCAL FAR ConditionFunc(
        const struct sockaddr FAR * CallerName,
        int CallerNamelen,
        LPWSABUF lpCallerData,
        LPQOS lpCallerSFlowspec,
        const struct sockaddr FAR * Calleename,
        int CalleeNamelen,
        LPWSABUF lpCalleeData,
        GROUP FAR * g,
        DWORD dwCallbackData );
```

{How about group QOS? Let's leave it out for now since:
1) It adds complexity
2) We aren't sure we need it
3) It isn't for sure that group info comes across the wire}

LPWSABUF and LPQOS are defined in winsock2.h as follows:

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

typedef enum
{
        GuaranteedService,
        BestEffortService
} GUARANTEE;

typedef struct _flowparams
{
        int64           AverageBandwith;// In Bytes/sec
        int64           PeakBandwidth;  // In Bytes/sec
        int64           BurstLength;    // In microseconds
        int64           Latency;        // In microseconds
        int64           DelayVariation; // In microseconds
        GUARANTEE       levelOfGuarantee;// Guaranteed or
                                         // Best Effort
        int32           CostOfCall;     // Reserved for future
                                         // use, must be set to 0
        int32           ProviderId;     // Provider Identifier
```

```
        int32           SizePSP;        // Length of provider
                                        // specific parameters
        UCHAR ProviderSpecificParams[1];// provider specific
                                        // parameters
} FLOWPARAMS;

typedef struct _QualityOfService
{
        FLOWPARAMS ForwardFP;       // Caller(Initiator) to callee
        FLOWPARAMS BackwardFP;      // Callee to caller
} QOS, FAR * LPQOS;
```

ConditionFunc is a placeholder for the application-supplied function name. In 16-bit Windowns environments, it is invoked in the same thread as **WSAAccept()**, thus no other Winsock functions can be called except **WSAIsBlocking()** and **WSACancelBlockingCall()**. The actual condition function must reside in a DLL or application module and be exported in the module definition file. You must use **MakeProcInstance()** to get a procedure-instance address for the callback function.

The *lpCallerId* and *lpCallerData* are value parameters which contain the address of the connecting entity and any user data that was sent along with the connection request, respectively.

lpCallerSFlowspec contains two blocks of memory containing the flow specs for socket *s*, one for each direction, specified by the caller. The forward or backward QOS values will be set to NULL as appropriate for any unidirectional sockets.. The first part of each memory block is struct FLOWSPEC, optionally followed by any service provider specific portion. Thus, *lpSFlowspec->Flen* and *lpSFlowspec->Blen* must be larger than or equal to the size of struct FLOWSPEC. A NULL value for *lpSFlowspec* indicates no caller supplied flow spec.

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

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

if *g* is an existing socket group id, add *s* to this group, provided all the requirements set by this group are met; or
if *g* = SG_UNCONSTRAINED_GROUP, create an unconstrained socket group and have *s* as the first member; or
if *g* = SG_CONSTRAINED_GROUP, create a constrained socket group and have *s* as the first member; or
if *g* = NULL, no group operation is performed.

For unconstrained groups, any set of sockets may be grouped together as long as they are supported by a single service provider and are connection-oriented. A constrained socket

group further requires that connections on all grouped sockets be to the same host. For newly created socket groups, the new group id can be retrieved by using **getsockopt()** with option SO_GROUP_ID, if this operation completes successfully.

Return Value

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

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

Error Codes

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

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

WSAAsyncSelect()

Description Request event notification for a socket.

#include < winsock2.h >

int PASCAL FAR WSAAsyncSelect (SOCKET *s*, **HWND** *hWnd*,
unsigned int *wMsg*, **long** *lEvent* **);**

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

Remarks This function is used to request that the Winsock2 DLL should send a message to the window *hWnd* whenever it detects any of the network events specified by the *lEvent* parameter. The message which should be sent is specified by the *wMsg* parameter. The socket for which notification is required is identified by *s*.

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

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

Value	Meaning
FD_READ	Want to receive notification of readiness for reading
FD_WRITE	Want to receive notification of readiness for writing
FD_OOB	Want to receive notification of the arrival of out-of-band data
FD_ACCEPT	Want to receive notification of incoming connections
FD_CONNECT	Want to receive notification of completed connection
FD_CLOSE	Want to receive notification of socket closure
FD_QOS	Want to receive notification of socket Quality of Service (QOS) changes
FD_GROUP_QOS	Want to receive notification of socket group Quality of Service (QOS) changes

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

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

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

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

To cancel all notification – i.e., to indicate that Winsock2 should send no further messages related to network events on the socket – *lEvent* should be set to zero.

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

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

Since an **accept()'**ed socket has the same properties as the listening socket used to accept it, any **WSAAsyncSelect()** events set for the listening socket apply to the accepted socket. For example, if a listening socket has **WSAAsyncSelect()** events FD_ACCEPT, FD_READ, and FD_WRITE, then any socket accepted on that listening socket will also have FD_ACCEPT, FD_READ, and FD_WRITE events with the same *wMsg* value used for messages. If a different *wMsg* or events are desired, the application should call **WSAAsyncSelect()**, passing the accepted socket and the desired new information.[1]

When one of the nominated network events occurs on the specified socket *s*, the application's window *hWnd* receives message *wMsg*. The *wParam* argument identifies the socket on which a network event has occurred. The low word of *lParam* specifies the network event that has occurred. The high word of *lParam* contains any error code. The error code be any error as defined in **Winsock2.h**.

The error and event codes may be extracted from the *lParam* using the macros WSAGETSELECTERROR and WSAGETSELECTEVENT, defined in **Winsock2.h** as:

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

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

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

Value	Meaning
FD_READ	Socket *s* ready for reading
FD_WRITE	Socket *s* ready for writing
FD_OOB	Out-of-band data ready for reading on socket s.

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

FD_ACCEPT	Socket *s* ready for accepting a new incoming connection
FD_CONNECT	Connection initiated on socket *s* completed
FD_CLOSE	Connection identified by socket *s* has been closed
FD_QOS	Quality of Service associated with socket *s* has changed.
FD_GROUP_QOS	Quality of Service associated with the socket group to which *s* belongs has changed.

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

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

As in the case of the **select()** function, **WSAAsyncSelect()** will frequently be used to determine when a data transfer operation (**send()** or **recv()**) can be issued with the expectation of immediate success. Nevertheless, a robust application must be prepared for the possibility that it may receive a message and issue a Winsock2 call which returns WSAEWOULDBLOCK immediately. For example, the following sequence of events is possible:

(i)	data arrives on socket **s**; Winsock2 posts **WSAAsyncSelect** message
(ii)	application processes some other message
(iii)	while processing, application issues an **ioctlsocket(s, FIONREAD...)** and notices that there is data ready to be read
(iv)	application issues a **recv(s,...)** to read the data
(v)	application loops to process next message, eventually reaching the **WSAAsyncSelect** message indicating that data is ready to read
(vi)	application issues **recv(s,...)**, which fails with the error WSAEWOULDBLOCK.

Other sequences are possible.

TheWinsock2 DLL will not continually flood an application with messages for a particular network event. Having successfully posted notification of a particular event to an application window, no further message(s) for that network event will be posted to the application window until the application makes the function call which implicitly reenables notification of that network event.

Event	Re-enabling function
FD_READ	**recv()** or **recvfrom()**
FD_WRITE	**send()** or **sendto()**
FD_OOB	**recv()**
FD_ACCEPT	**accept()** or **WSAAcceptEx()** unless the error code returned is WSATRY_AGAIN indicating that the condition function returned CF_DEFER
FD_CONNECT	NONE
FD_CLOSE	NONE
FD_QOS	**getsockopt()** with option SO_FLOWSPEC
FD_GROUP_QOS	**getsockopt()** with option SO_GROUP_FLOWSPEC

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

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

(i) network transport stack receives 100 bytes of data on socket **s** and causes Winsock2 to post an FD_READ message.
(ii) The application issues **recv(s, buffptr, 50, 0)** to read 50 bytes.
(iii) another FD_READ message is posted since there is still data to be read.

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

If an event has already happened when the application calls **WSAAsyncSelect()** or when the reenabling function is called, then a message is posted as appropriate. All the events have persistence beyond the occurrence of their respective events. For example, consider the following sequence: 1) an application calls **listen()**, 2) a connect request is received but not yet accepted, 3) the application calls **WSAAsyncSelect()** specifying that it wants to receive FD_ACCEPT messages for the socket. Due to the persistence of events, Winsock2 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 accepted with **accept()**, and then after a **send()** or **sendto()** fails with WSAEWOULDBLOCK and buffer space becomes available. Therefore, an application can assume that sends are possible starting from the first FD_WRITE message and lasting until a send returns WSAEWOULDBLOCK. After such a failure the application will be notified that sends are again possible with an FD_WRITE message.

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

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

The FD_CLOSE message is posted when a close indication is received for the virtual circuit corresponding to the socket. In TCP terms, this means that the FD_CLOSE is posted when the connection goes into the FIN WAIT or CLOSE WAIT states. This results from the remote end performing a **shutdown()** on the send side or a **closesocket().**

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

The FD_QOS or FD_GROUP_QOS message is posted when any field in the flow spec associated with socket *s* or the socket group that *s* belongs to has changed, respectively. Applications might use **getsocketopt()** with option SO_FLOWSPEC or SO_GROUP_FLOWSPEC to get the current QOS for socket *s* or for the socket group *s* belongs to, respectively.

Error Codes

WSANOTINITIALISED	A successful **WSAStartup()** must occur before using this API.
WSAENETDOWN	The network subsystem has failed.
WSAEINVAL	Indicates that one of the specified parameters was invalid, or the specified socket is in an invalid state.
WSAEINPROGRESS	A blocking Winsock2 call is in progress, or the service provider is still processing a callback function (see section **Error! Reference source not found.**).
WSAENOTSOCK	The descriptor is not a socket.

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

Event: FD_CONNECT

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

Event: FD_CLOSE

Error Code	Meaning
WSAENETDOWN	The network subsystem has failed.

WSAECONNRESET	The connection was reset by the remote side.
WSAECONNABORTED	The connection was aborted due to timeout or other failure.

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

Error Code	Meaning
WSAENETDOWN	The network subsystem has failed.

See Also

select(), WSACallbackSelect()

WSAConnect()

Description Establish a connection to a peer, create and/or join a socket group, and specify needed quality of service based on the supplied flow spec.

#include <winsock2.h>

int WSAAPI WSAConnect (SOCKET *s*, **const struct sockaddr FAR *** *name*, **int** *namelen*, **LPWSABUF** *lpCallerData*, **LPWSABUF** *lpCalleeData*, **GROUP** *g*, **LPQOS** *lpSFlowspec*, **LPQOS** *lpGFlowspec* **);**

s	A descriptor identifying an unconnected socket.
name	The name of the peer to which the socket is to be connected.
namelen	The length of the *name*.
lpCallerData	A pointer to the user data that is to be transferred to the peer during connection establishment.
lpCalleeData	A pointer to the user data that is to be transferred back from the peer during connection establishment.
g	The identifier of the socket group.
lpSFlowspec	A pointer to the flow specs for socket *s*, one for each direction.
lpGFlowspec	A pointer to the flow specs for the socket group to be created, one for each direction, if the value of parameter *g* is SG_CONSTRAINED_GROUP. Otherwise, this parameter is ignored.

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. For connection-oriented sockets (e.g., type SOCK_STREAM), an active connection is initiated to the foreign host using *name* (an address in the name space of the socket; for a detailed description, please see **bind()**). When this call completes successfully, the socket is ready to send/receive data.

For a connectionless socket (e.g., type SOCK_ DGRAM), the operation performed by **WSAConnect()** is merely to establish a default destination address so that the socket may be used on subsequent connection-oriented send and receive operations (**send(), WSASend(), recv(), WSARecv()**). On connectionless sockets, exchange of user to user data is not possible and the corresponding parameters will be silently ignored.

If the socket, *s*, is unbound, unique values are assigned to the local association by theWinsock provider, and the socket is marked as bound. Note that if the address field of the *name* structure is all zeroes, **WSAConnect()** will return the error WSAEADDRNOTAVAIL.

The application is responsible for allocating any memory space pointed to directly or indirectly by any of the parameters it specifies. **LPWSABUF** and **LPQOS** are defined in winsock2.h as follows:

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

```
typedef enum
{
      GuaranteedService,
      BestEffortService
} GUARANTEE;

typedef struct _flowparams
{
      int64          AverageBandwith;// In Bytes/sec
      int64          PeakBandwidth;  // In Bytes/sec
      int64          BurstLength;    // In microseconds
      int64          Latency;        // In microseconds
      int64          DelayVariation; // In microseconds
      GUARANTEE      levelOfGuarantee;// Guaranteed or
                                     // Best Effort
      int32          CostOfCall;     // Reserved for future
                                     // use, must be set to 0
      int32          ProviderId;     // Provider Identifier
      int32          SizePSP;        // Length of provider
                                     // specific parameters
      UCHAR ProviderSpecificParams[1];// provider specific
                                     // parameters
} FLOWPARAMS;

typedef struct _QualityOfService
{
      FLOWPARAMS ForwardFP;      // Caller(Initiator) to callee
      FLOWPARAMS BackwardFP;     // Callee to caller
} QOS, FAR * LPQOS;
```

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

Parameter *g* is used to indicate the appropriate actions on socket groups:

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

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

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

if *g* = NULL, no operation is performed, and is equivalent to connect().
For unconstrained groups, any set of sockets may be grouped together as long as they are supported by a single service provider and are connection-oriented. A constrained socket group requires that connections on all grouped sockets be to the same host. For newly created socket groups, the new group id can be retrieved by using **getsockopt()** with option SO_GROUP_ID, if this operation completes successfully.

lpSFlowspec specifies two blocks of memory containing the flow specs for socket *s*, one for each direction. 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 forward or backward QOS values will be ignored, respectively, for any unidirectional sockets.. The first part of each memory block is struct FLOWSPEC, optionally followed by any service provider specific portion. Thus, *lpSFlowspec->Flen* and *lpSFlowspec->Blen* must be larger than or equal to the size of struct FLOWSPEC. A NULL value for *lpSFlowspec* indicates no application supplied flow spec.

lpGFlowspec specifies two blocks of memory containing the flow specs for the socket group to be created, one for each direction, provided that the value of parameter *g* is SG_CONSTRAINED_GROUP. Otherwise, these values are ignored. The first part of each memory block is struct FLOWSPEC, optionally followed by any service provider specific portion. Thus, *lpGFlowspec->Flen* and *lpGFlowspec->Blen* must be larger than or equal to the size of struct FLOWSPEC. A NULL value for *lpGFlowspec* indicates no application-supplied group flow spec.

Comments

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

Return Value

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

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

On a non-blocking socket, if the return value is SOCKET_ERROR an application should call **WSAGetLastError().** If this indicates an error code of WSAEWOULDBLOCK, then your application can either:

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

2. If your application is using **WSAAsyncSelect()** to indicate interest in connection events, then your application will receive an FD_CONNECT notification when the connect operation is complete.

3. If your applicationis using **WSAEventSelect()** to indicate interest in connection events, then the associated event object will be signaled when the connect operation is complete.

Error Codes

WSANOTINITIALISED	A successful **WSAStartup()** must occur before using this API.

WSAENETDOWN	The network subsystem has failed.
WSAEADDRINUSE	The specified address is already in use.
WSAEINTR	The (blocking) call was canceled via **WSACancelBlockingCall().**
WSAEINPROGRESS	A blocking Winsock call is in progress, or the service provider is still processing a callback function (see section **Error! Reference source not found.**).
WSAEADDRNOTAVAIL	The specified address is not available from the local machine.
WSAEAFNOSUPPORT	Addresses in the specified family cannot be used with this socket.
WSAECONNREFUSED	The attempt to connect was forcefully rejected.
WSAEDESTADDREQ	A destination address is required.
WSAEFAULT	The *namelen* argument is incorrect, the buffer length for *lpCalleeData*, *lpSFlowspec*, and *lpGFlowspec* are too small, or the buffer length for *lpCallerData* is too large.
WSAEINVAL	The parameter *g* specified in the condition function is not a valid value, or the parameter *s* is a listening socket.
WSAEISCONN	The socket is already connected.
WSAEMFILE	No more socket descriptors are available.
WSAENETUNREACH	The network can't be reached from this host at this time.
WSAENOBUFS	No buffer space is available. The socket cannot be connected.
WSAENOTSOCK	The descriptor is not a socket.
WSAEOPNOTSUPP	The flow specs specified in *lpSFlowspec* and *lpGFlowspec* cannot be satisfied.
WSAEPROTONOSUPPORT	The *lpCallerData* augment is not supported by the service provider.
WSAETIMEDOUT	Attempt to connect timed out without establishing a connection

WSAEWOULDBLOCK	The socket is marked as non-blocking and the connection cannot be completed immediately. It is possible to **select()** the socket while it is connecting by **select()**ing it for writing.

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

WSADuplicateSocket()

Description Create a shared socket for a specified task.

#include <winsock2.h>

SOCKET WSAAPI WSADuplicateSocket (SOCKET *s*, **WSATASK** *hTargetTask* **);**

s	Specifies the local socket descriptor.
hTargetTask	Specifies the handle of the target task for which the shared socket will be used.

Remarks This function is used to enable socket sharing by creating a shared socket. Shared sockets are created in the context of the source task by supplying an existing, local socket descriptor and a handle to the target task (which could be the same task as the source task) for which the sharedsocket will be used. The newly created shared socket descriptor only has meaning within the context of the target task.

To get the handle of the target task, it will generally be necessary to use some form of interprocess communication (IPC), which is out of the scope of this specification. Since the created shared socket only has meaning in the target task, the source task must pass the value of the shared socket descriptor to the target task, again via some IPC mechanism.

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

Comments Shared sockets may be used in all places where regular sockets are used and are, in fact, indistinguishable from them. Shared sockets derived from a common regular socket or its derivatives share all aspects of the underlying common socket object with the exception of the notification mechanism. Reference counting is employed to ensure that the underlying socket object is not closed until the last shared socket is closed.

Since the collection of attributes which comprise a socket object's option set is shared, setting any socket option on a shared socket may have a global effect. For example, if one task uses **ioctlsocket()** on a shared socket to set it into non-blocking mode, this change is visible to all of the shared sockets that reference the underlying common socket object.

Each shared socket has an independent notification mechanism which conforms to the usual Winsock conventions. Thus if two or more tasks are sharing an underlying socket object and each requests overlapped notification via Windows messages when data is ready to be read, all such tasks will receive their stipulated message in an unspecified sequence. The first task to perform a read will get some or all of the available data, the others will get what's left, if any. In other words, it is completely up to tasks which share a socket to coordinate their access to the socket.

As an aside, we note that simply invoking the **WSADuplicateSocket()** function on a socket *s,* causes *s* to become a shared socket which references an underlying socket object.

Error Codes

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

See Also

WSAEnumProtocols()

Description Retrieve information about available transport protocols.

#include <winsock2.h>

int WSAAPI WSAEnumProtocols (LPDWORD *lpdwProtocols*, **LPVOID** *lpProtocolBuffer,* **LPDWORD** *lpdwBufferLength*)**;**

lpdwProtocols a NULL-terminated array of protocol ids. This parameter is optional; if *lpdwProtocols* is NULL, information on all available protocols is returned, otherwise information is retrieved only for those protocols listed in the array.

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

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

Remarks This function is used to discover information about the collection of transport protocols installed on the local machine. The *lpdwProtocols* parameter can be used as a filter to constrain the amount of information provided. Normally it will be supplied as a NULL pointer which will cause the routine to return information on all available transport protocols.

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

Definitions **PROTOCOL_INFO Structure:**

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

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

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

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

be delivered, but that any data that is delivered is delivered in the order in which it was sent.

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

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

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

XP1_EXPEDITED_DATA - the protocol supports expedited (urgent) data.

XP1_CONNECT_DATA - the protocol supports connect data.

XP1_DISCONNECT_DATA - the protocol supports disconnect data.

XP1_SUPPORTS_BROADCAST - the protocol supports a broadcast mechanism.

XP1_SUPPORTS_MULTICAST - the protocol supports a multicast mechanism.

XP1_QOS_SUPPORTED - the protocol supports quality of service requests.

XP1_ENCRYPTS - the protocol supports data encryption.

XP1_INTERRUPT - for 16 bit environments (only), the protocol allows **send()/WSASend()** and **recv()/WSARecv()** to be invoked in interrupt context.

XP1_UNI_SEND - the protocol is unidirectional in the send direction.

XP1_UNI_RECV - the protocol is unidirectional in the recv direction.

DWORD *dwServiceFlags2* - reserved for additional protocol attribute definitions

DWORD *dwServiceFlags3*- reserved for additional protocol attribute definitions

DWORD *dwServiceFlags4* - reserved for additional protocol attribute definitions

INT *iProviderID*- A unique identifier assigned to the underlying transport service provider at the time it was installed under Winsock 2. This value is useful for instances where more than one service provider is able to implement a particular protocol. An application may use the *iProviderID* value to distinguish between providers that might otherwise be indistinguishable.

INT *iVersion* -Protocol version identifier.

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

INT *iMaxSockAddr* - The maximum address length.

INT *iMinSockAddr* - The minimum address length.

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

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

BOOL *bMultiple* - A flag to indicate that this is one of two or more entries for a single protocol 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.

BOOL *bFirst* - A flag to indicate that this is the prime or most frequently used entry for a protocol which is capable of implementing multiple behaviors.

DWORD *dwMessageSize* - The maximum message size supported by the protocol. This is the maximum size that can be sent from any of the host's local interfaces. For protocols which do not support message framing, the actual maximum that can be sent to a given address may be less. The following special values are defined:

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

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

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

LPTSTR *lpProtocol* - a pointer to a human-readable name identifying the protocol, for example "SPX2".

DWORD *dwNameSpaces* - information on which name spaces can be found by the transport this protocol is contained within. Value encoding is TBD.

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

WSAEnumNetworkEvents()

Description Discover occurrences of network events for the indicated socket.

#include <winsock2.h>

int WSAAPI WSAEnumNetworkEvents (SOCKET *s*, **WSAEVENT** *hEventObject*, **LPWSANETWORKEVENT** *lpNetworkEvents*, **LPINT** *lpiCount*)**;**

s A descriptor identifying the socket.

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

lpNetworkEvents An array of WSANETWORKEVENT structs, each of which records an occurred network event and the associated error code.

lpiCount The number of elements in the array. Upon returning, this parameter indicates the actual number of elements in the array, or the minimum number of elememts needed to retrieve all the network events if the return value is WSAENOBUFS.

Remarks This function is used to discover which network events have occurred for the indicated socket since the last invocation of this function. It is intended for use in conjunction with **WSAEventSelect(),** which associates an event object with one or more network events. The socket's internal record of network events is copied to *lpNetworkEvents*, whereafter the internal network events record is cleared. If *hEventObject* is non-null, the indicated event object is also reset. The Winsock2 DLL 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.

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

Event: FD_CONNECT

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

Event: FD_CLOSE

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

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

Error Code	Meaning
WSAENETDOWN	The network subsystem has failed.

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

Error Codes

WSANOTINITIALISED	A successful **WSAStartup()** must occur before using this API.
WSAENETDOWN	The network subsystem has failed.
WSAEINVAL	Indicates that one of the specified parameters was invalid
WSAEINPROGRESS	A blocking Winsock call is in progress, or the service provider is still processing a callback function (see section **Error! Reference source not found.**).
WSAENOBUFS	The supplied buffer is too small.

See Also **WSAEventSelect()**

WSAEventSelect()

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

#include <winsock2.h>

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

s	A descriptor identifying the socket.
hEventObject	A handle identifying the event object to be associated with the supplied set of FD_XXX network events.
lNetworkEvents	A bitmask which specifies the combination of FD_XXX network events in which the application has interest.

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

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

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

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

Value	Meaning
FD_READ	Want to receive notification of readiness for reading
FD_WRITE	Want to receive notification of readiness for writing
FD_OOB	Want to receive notification of the arrival of out-of-band data
FD_ACCEPT	Want to receive notification of incoming connections
FD_CONNECT	Want to receive notification of completed connection
FD_CLOSE	Want to receive notification of socket closure
FD_QOS	Want to receive notification of socket Quality of Service (QOS) changes
FD_GROUP_QOS	Want to receive notification of socket group Quality of Service (QOS) changes

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

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

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

```
rc = WSAEventSelect(s, hEventObject1, FD_READ);
rc = WSAEventSelect(s, hEventObject2, FD_WRITE);
```

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 needs to call **WSACloseEvent()** to explicitly close the event object and free any resources.

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

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

Comments As in the case of the **select()** and **WSAAsyncSelect()** functions, **WSAEventSelect()** will frequently be used to determine when a data transfer operation (**send()** or **recv()**) can be issued with the expectation of immediate success. Nevertheless, a robust application must be prepared for the possibility that the event object is set and it issues a Winsock call which returns WSAEWOULDBLOCK immediately. For example, the following sequence of operations is possible:

---

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

(i) data arrives on socket **s**; Winsock sets the **WSAEventSelect** event object
(ii) application does some other processing
(iii) while processing, application issues an **ioctlsocket(s, FIONREAD...)** and notices that there is data ready to be read
(iv) application issues a **recv(s,...)** to read the data
(v) application eventually waits on event object specified in **WSAEventSelect**, which returns immediately indicating that data is ready to read
(vi) application issues **recv(s,...)**, which fails with the error WSAEWOULDBLOCK.

Other sequences are possible.

Having successfully recorded the occurrence of the network event (by setting the corresponding bit in the internal network event record) and signaled the associated event object, no further actions are taken for that network event until the application makes the function call which implicitly reenables the setting of that network event and signaling of the associated event object.

Network Event	Re-enabling function
FD_READ	**recv()** or **recvfrom()**
FD_WRITE	**send()** or **sendto()**
FD_OOB	**recv()**
FD_ACCEPT	**accept()** or **WSAAccept()** unless the error code returned is WSATRY_AGAIN indicating that the condition function returned CF_DEFER
FD_CONNECT	NONE
FD_CLOSE	NONE
FD_QOS	**getsockopt()** with option SO_FLOWSPEC
FD_GROUP_QOS	**getsockopt()** with option SO_GROUP_FLOWSPEC

Any call to the reenabling routine, even one which fails, results in reenabling of recording and setting for the relevant network event and event object, respectively.

For FD_READ, FD_OOB, FD_ACCEPT, FD_QOS and FD_GROUP_QOS network events, network event recording and event object setting are "level-triggered." This means that if the reenabling routine is called and the relevant network condition is still valid after the call, the network event is recorded and the associated event object is set . This allows an application to be event-driven and not be concerned with the amount of data that arrives at any one time. Consider the following sequence:

(i) transport provider receives 100 bytes of data on socket **s** and causes Winsock2 DLL to record the FD_READ network event and set the associated event object.
(ii) The application issues **recv(s, buffptr, 50, 0)** to read 50 bytes.
(iii) The transport provider causes WINSOCK DLL to record the FD_READ network event and sets the associated event object again since there is still data to be read.

With these semantics, an application need not read all available data in response to an FD_READ network event --a single **recv()** in response to each FD_READ network event is appropriate.

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. All the network events have persistence beyond the occurrence of their respective events. For example, consider the following sequence: 1) an application calls **listen()**, 2) a connect request is received but not yet accepted, 3) the application calls **WSAEventSelect()** specifying that it is interested in the FD_ACCEPT network event for the socket. Due to the persistence of network events, Winsock records the FD_ACCEPT network event and sets the associated event object immediately.

The FD_WRITE network event is handled slightly differently. An FD_WRITE network event is recorded when a socket is first connected with **connect()** or accepted with **accept()**, and then after a **send()** or **sendto()** 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 settting and lasting until a send returns WSAEWOULDBLOCK. After such a failure the application will find out that sends are again possible when an FD_WRITE network event is recorded and the associated event object is set .

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

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

The FD_CLOSE network event is recorded when a close indication is received for the virtual circuit corresponding to the socket. In TCP terms, this means that the FD_CLOSE is recorded when the connection goes into the FIN WAIT or CLOSE WAIT states. This results from the remote end performing a **shutdown()** on the send side or a **closesocket().**

Please note Winsock will record ONLY an FD_CLOSE network event to indicate closure of a virtual circuit. It will NOT record an FD_READ network event to indicate this condition.

The FD_QOS or FD_GROUP_QOS network event is recorded when any field in the flow spec associated with socket *s* or the socket group that *s* belongs to has changed, respectively. Applications should use **getsocketopt()** with option SO_FLOWSPEC or SO_GROUP_FLOWSPEC to get the current QOS for socket *s* or for the socket group *s* belongs to, respectively.

Error Codes

WSANOTINITIALISED	A successful **WSAStartup()** must occur before using this API.

WSAENETDOWN	The network subsystem has failed.
WSAEINVAL	Indicates that one of the specified parameters was invalid, or the specified socket is in an invalid state.
WSAEINPROGRESS	A blocking Winsock call is in progress, or the service provider is still processing a callback function (see section **Error! Reference source not found.**).
WSAENOTSOCK	The descriptor is not a socket.

See Also **WSACloseEvent(),WSACreateEvent(), WSAEnumNetworkEvents(), WSAGetOverlappedResult(),WSAWaitForMultipleEvents().**

WSARecv()

Description Receive data from a socket, possibly using overlapped I/O.

#include <winsock2.h>

int WSAAPI WSARecv (SOCKET *s*, **LPVOID** *lpBuffer*, **DWORD** *nNumberOfBytesToRecv*, **LPDWORD** *lpNumberOfBytesRecvd*, **LPINT** *lpFlags,* **LPWSAOVERLAPPED** *lpOverlapped*, **LPWSAOVERLAPPED_COMPLETION_ROUTINE** *lpCompletionRoutine* **);**

s	A descriptor identifying a connected socket.
lpBuffer	A pointer to the buffer for the incoming data.
nNumberOfBytesToRecv	The number of bytes to receive from the network.
lpNumberOfBytesRecvd	A pointer to the number of bytes received by this call.
lpFlags	A pointer to flags.
lpOverlapped	A pointer to a WSAOVERLAPPED structure (ignored for non-overlapped sockets).
lpCompletionRoutine	A pointer to the completion routine called when the receive operation has been completed (ignored for non-overlapped sockets)..

Remarks This function is used on a connection-oriented socket specified by *s*. For overlapped sockets it is used to post a buffer into which incoming data will be placed as it becomes available. For non-overlapped sockets it behaves the same as the standard recv() function except that the *flags* parameter is both an input and an output parameter. It can also be used on connectionless sockets which have a stipulated default peer address established via the **connect()** or **WSAConnect()** functions.

For byte stream style sockets (e.g., type SOCK_STREAM), incoming data is placed into the buffer until the buffer is filled. For message-oriented sockets (e.g., type SOCK_DGRAM), an incoming message is placed into the supplied buffer, up to the size of the buffer supplied. If the message is larger than the buffer supplied, the buffer is filled with the first part of the message. The MSG_PARTIAL flag is set for the socket, if this feature is supported by the underlying protocol, and subsequent receive operation(s) will retrive the rest of the message. Otherwise, the excess data is lost, and **WSARecv()** returns the error WSAEMSGSIZE.
If the socket is connection-oriented and the remote side has shut down the connection gracefully, **WSARecv()** will fail with the error WSAEDISCON. If the connection has been reset, a **WSARecv()** will fail with the error WSAECONNRESET.

Overlapped socket I/O:
This function may be called from within the completion routine of a previous **WSARead()**, **WSAReadFrom()**, **WSASend()** or **WSASendTo()** function. In Win16 environments, this function may also be called from within interrupt context provided that the XP1_INTERRUPT bit in the associated PROTOCOL_INFO struct is TRUE.

The *lpOverlapped* parameter much be valid for the duration of the overlapped operation. The WSAOVERLAPPED structure has the following form:

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

If the *lpCompletionRoutine* parameter is NULL, the *hEvent* field of *lpOverlapped* must be a valid event object handle which is signaled when the overlapped operation completes. An application can use **WSAWaitForMultipleEvents()** or **WSAGetOverlappedResult()** to wait or poll on the event object.

If *lpCompletionRoutine* is not NULL, the *hEvent* field is ignored and can be used by the application to pass context information to the completion routine.

In Win16 environments, the service provider may invoke the application's completion routine from within an interrupt context. Application developers should assume that this will be the case and restrict any calls made from within the completion routine to those that are safe for interrupt context.

In Win32 environments, completion functions are invoked following the usual rules for overlapped procedure calls or APCs. Specifically, the calling thread must be in an alertable wait (e.g., using **WSAWaitForMultipleEvents**()) in order for the completion routine to be invoked.. If the calling thread is not in an alertable wait when the overlapped operation has been completed, the system queues the completion routine call until the thread enters an alterable wait. The prototype of the completion routine is as follows:

VOID CALLBACK CompletionRoutine(**DWORD** *dwError***,**
DWORD *cbTransferred***, LPWSAOVERLAPPED** *lpOverlapped* **);**

CompletionRoutine is a placeholder for an application-defined or library-defined function name. *dwError* specifies the completion status for the overlapped operation as indicated by *lpOverlapped*. *cbTransferred* specifies the number of bytes received. This function does not return a value.

Returning from this function allows invocation of another pending completion routine for this socket. In Win32 environments, all waiting completion routines are called before the alterable thread's wait is satisfied with a return code of WSA_IO_COMPLETION. The completion routines may be called in any order, not necessarily in the same order the overlapped operations are completed. However, the posted buffers are guaranteed to be filled in the same order they are supplied

Upon the completion of the overlapped operation, the *lpNumberOfBytesRecvd* parameter is filled with the number of bytes received, and, 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*.

Return Value

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

Error Codes

WSANOTINITIALISED	A successful **WSAStartup()** must occur before using this API.
WSAENETDOWN	The network subsystem has failed.
WSAENOTCONN	The socket is not connected.
WSAENETRESET	The connection must be reset because the service provider dropped it.
WSAENOTSOCK	The descriptor is not a socket.
WSAEOPNOTSUPP	MSG_OOB was specified, but the socket is not stream style such as type SOCK_STREAM, out-of-band data is not supported in the communication domain associated with this socket, or the socket is unidirectional and supports only send operations.
WSAESHUTDOWN	The socket has been shutdown; it is not possible to **WSARecv()** on a socket after **shutdown()** has been invoked with *how* set to SD_RECEIVE or SD_BOTH.
WSAEWOULDBLOCK	Overlapped sockets: There are too many outstanding overlapped I/O requests. Non-overlapped sockets: The socket is marked as non-blocking and the receive operation cannot be completed immediately.
WSAEMSGSIZE	The message was too large to fit into the specified buffer and was truncated.
WSAEINVAL	The socket has not been bound with **bind()**, or the socket is not created with the overlapped flag**.**
WSAECONNABORTED	The virtual circuit was aborted due to timeout or other failure.
WSAECONNRESET	The virtual circuit was reset by the remote side.
WSAEDISCON	The remote side gracefully close the connection.

See Also **WSACloseEvent(),WSACreateEvent(), WSAGetOverlappedResult(), WSASocket(), WSAWaitForMultipleEvents()**

WSARecvfrom()

Description Receive a datagram and store the source address, possibly using overlapped I/O.

#include <winsock2.h>

int WSAAPI WSARecvfrom (SOCKET *s*, **LPVOID** *lpBuffer*, **DWORD** *nNumberOfBytesToRecv*, **LPDWORD** *lpNumberOfBytesRecvd*, **LPINT** *lpFlags,* **LPVOID** *lpFrom*, **LPINT** *lpFromlen,* **LPWSAOVERLAPPED** *lpOverlapped*, **LPWSAOVERLAPPED_COMPLETION_ROUTINE** *lpCompletionRoutine* **);**

s	A descriptor identifying a socket
lpBuffer	A pointer to the buffer for the incoming data.
nNumberOfBytesToRecv	The number of bytes to receive from the network.
lpNumberOfBytesRecvd	A pointer to the number of bytes received by this call.
lpFlags	A pointer to flags.
lpFrom	An optional pointer to a buffer which will hold the source address upon the completion of the overlapped operation.
lpFromlen	An optional pointer to the size of the *from* buffer.
lpOverlapped	A pointer to a WSAOVERLAPPED structure (ignored for non-overlapped sockets)..
lpCompletionRoutine	A pointer to the completion routine called when the receive operation has been completed (ignored for non-overlapped sockets)..

Remarks For overlapped sockets, this function is used to post a buffer into which incoming data will be placed as it becomes available on a (possibly connected) socket. The data must have already been received by the transport for non-overlapped sockets.

For connectionless socket types, the address from which the data originated is copied to the buffer pointed by *lpFrom*. The value pointed to by *lpFromlen* is initialized to the size of this buffer, and is modified on return to indicate the actual size of the address stored there. The *lpFrom* and *lpFromlen* parameters are ignored for connection-oriented sockets.

For byte stream style sockets (e.g., type SOCK_STREAM), incoming data is placed into the buffer until the buffer is filled. For message-oriented sockets, an incoming message is placed into the supplied buffer, up to the size of the buffer supplied. If the message is larger than the buffer supplied, the buffer is filled with the first part of the message. The MSG_PARTIAL flag is set for the socket, if this feature is supported by the underlying protocol, and subsequent receive operation(s) will retrive the rest of the message. Otherwise, the excess data is lost, and **WSARecvfrom()** returns the error code WSAEMSGSIZE.

If the socket is connection-oriented and the remote side has shut down the connection gracefully, a **WSARecvfrom()** will fail with the error WSAEDISCON. If the connection has been reset **WSARecvfrom()** will fail with the error WSAECONNRESET.

Overlapped socket I/O:
This function may be called from within the completion routine of a previous **WSARead**(), **WSAReadFrom**(), **WSASend**() or **WSASendTo**() function. In Win16 environments, this function may also be called from within interrupt context provided that the XP1_INTERRUPT bit in tthe associated PROTOCOL_INFO struct is TRUE.

The *lpOverlapped* parameter much be valid for the duration of the overlapped operation. The WSAOVERLAPPED structure has the following form:

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

If the *lpCompletionRoutine* parameter is NULL, the *hEvent* field of *lpOverlapped* must be an event object handle which is signaled when the overlapped operation completes. An application can use **WSAWaitForMultipleEvents**() or **WSAGetOverlappedResult**() to wait or poll on the event object.

If *lpCompletionRoutine* is not NULL, the *hEvent* field is ignored and can be used by the application to pass context information to the completion routine.

In Win16 environments, the service provider may invoke the application's completion routine from within an interrupt context. Application developers should assume that this will be the case and restrict any calls made from within the completion routine to those that are safe for interrupt context.

In Win32 environments, completion functions are invoked following the usual rules for asynchronous procedure calls or APCs. Specifically, the calling thread must be in an alertable wait (e.g., using **WSAWaitForMultipleEvents**()) in order for the completion routine to be invoked.. If the calling thread is not in an alertable wait when the overlapped operation has been completed, the system queues the completion routine call until the thread enters an alterable wait. The prototype of the completion routine is as follows:

VOID CALLBACK CompletionRoutine(**DWORD** *dwError***,**
DWORD *cbTransferred***, LPWSAOVERLAPPED** *lpOverlapped* **);**

CompletionRoutine is a placeholder for an application-defined or library-defined function name. *dwError* specifies the completion status for the overlapped operation as indicated by *lpOverlapped*. *cbTransferred* specifies the number of bytes received. This function does not return a value.

Returning from this function allows invocation of another pending completion routine for this socket. In Win32 environments, all waiting completion routines are called before the alterable thread's wait is satisfied with a return code of WSA_IO_COMPLETION. The completion routines may be called in any order, not necessarily in the same order the overlapped operations are completed. However, the posted buffers are guaranteed to be filled in the same order they are supplied

Upon the completion of the overlapped operation, the *lpNumberOfBytesRecvd* parameter is filled with the number of bytes received, and, 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*.

Return Value

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

Error Codes

WSANOTINITIALISED	A successful **WSAStartup()** must occur before using this API.
WSAENETDOWN	The network subsystem has failed.
WSAEFAULT	The *lpFromlen* argument was invalid: the *lpFrom* buffer was too small to accommodate the peer address.
WSAEINVAL	The socket has not been bound with **bind()**, or the socket is not created with the overlapped flag**.**
WSAENETRESET	The connection must be reset because the Winsock provider dropped it.
WSAENOTCONN	The socket is not connected (connection-oriented sockets only).
WSAENOTSOCK	The descriptor is not a socket.
WSAEOPNOTSUPP	MSG_OOB was specified, but the socket is not stream style such as type SOCK_STREAM, out-of-band data is not supported in the communication domain associated with this socket, or the socket is unidirectional and supports only send operations.
WSAESHUTDOWN	The socket has been shutdown; it is not possible to **WSARecvfrom()** on a socket after **shutdown()** has been invoked with *how* set to SD_RECEIVE or SD_BOTH.
WSAEWOULDBLOCK	Overlapped sockets: There are too many outstanding overlapped I/O requests. . Non-overlapped sockets: The socket is marked as non-blocking and the receive operation cannot be completed immediately.
WSAEMSGSIZE	The message was too large to fit into the specified buffer and was truncated.
WSAECONNABORTED	The virtual circuit was aborted due to timeout or other failure.
WSAECONNRESET	The virtual circuit was reset by the remote side.
WSAEDISCON	The remote side gracefully close the connection.

See Also **WSACloseEvent(),WSACreateEvent(), WSAGetOverlappedResult(), WSASocket(), WSAWaitForMultipleEvents()**

WSASend()

Description Send data on a connected socket using overlapped I/O.

#include <winsock2.h>

int WSAAPI WSASend (SOCKET *s*, **LPVOID** *lpBuffer*, **DWORD** *nNumberOfBytesToSend*, **LPDWORD** *lpNumberOfBytesSent*, **int** *iFlags,* **LPWSAOVERLAPPED** *lpOverlapped*, **LPWSAOVERLAPPED_COMPLETION_ROUTINE** *lpCompletionRoutine* **);**

s	A descriptor identifying a connected socket which was created using **WSASocket()** with flag WSA_FLAG_OVERLAPPED.
lpBuffer	A pointer to the buffer for the outgoing data.
nNumberOfBytesToSend	The number of bytes to send to the network.
lpNumberOfBytesSent	A pointer to the number of bytes sent by this call.
iFlags	Flags.
lpOverlapped	A pointer to a WSAOVERLAPPED structure.
lpCompletionRoutine	A pointer to the completion routine called when the send operation has been.

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

Note that the successful completion of a **WSASend()** does not indicate that the data was successfully delivered. Upon completion of the overlapped operation, the value pointed to by the *lpNumberOfBytesSent* parameter is updated with the number of bytes sent.

This function may be called from within the completion routine of a previous **WSARead**(), **WSAReadFrom**(), **WSASend()** or **WSASendTo()** function. In Win16 environments, this function may also be called from within interrupt context provided that the XP1_INTERRUPT bit in the socket's PROTOCOL_INFO struct is TRUE.

The *lpOverlapped* parameter much be valid for the duration of the overlapped operation. The WSAOVERLAPPED structure has the following form:

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

If the *lpCompletionRoutine* parameter is NULL, the *hEvent* field of *lpOverlapped* must be a valid event object handle which is signaled when the overlapped operation

completes. An application can use **WSAWaitForMultipleEvents()** or **WSAGetOverlappedResult()** to wait or poll on the event object.

If *lpCompletionRoutine* is not NULL, the *hEvent* field is ignored and can be used by the application to pass context information to the completion routine.

In Win16 environments, the service provider may invoke the application's completion routine from within an interrupt context. Application developers should assume that this will be the case and restrict any calls made from within the completion routine to those that are safe for interrupt context.

In Win32 environments, completion functions are invoked following the usual rules for asynchronous procedure calls or APCs. Specifically, the calling thread must be in an alertable wait (e.g., using **WSAWaitForMultipleEvents()**) in order for the completion routine to be invoked.. If the calling thread is not in an alertable wait when the overlapped operation has been completed, the system queues the completion routine call until the thread enters an alterable wait. The prototype of the completion routine is as follows:

VOID CALLBACK CompletionRoutine(DWORD *dwError***,**
DWORD *cbTransferred***, LPWSAOVERLAPPED** *lpOverlapped* **);**

CompletionRoutine is a placeholder for an application-defined or library-defined function name. *dwError* specifies the completion status for the overlapped operation as indicated by *lpOverlapped*. *cbTransferred* specifies the number of bytes sent. This function does not return a value.

Returning from this function allows invocation of another pending completion routine for this socket. In Win32 environments, all waiting completion routines are called before the alterable thread's wait is satisfied with a return code of WSA_IO_COMPLETION. The completion routines may be called in any order, not necessarily in the same order the overlapped operations are completed. However, the posted buffers are guaranteed to be sent in the same order they are supplied

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

Value	Meaning
MSG_DONTROUTE	Specifies that the data should not be subject to routing. A Winsock service provider may choose to ignore this flag; see also the discussion of the SO_DONTROUTE option in section **Error! Reference source not found.**.
MSG_PARTIAL	Specifies that *lpBuffer* only contains a partial message. Note that this flag will be ignored by transports which do not support partial message transmissions.

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

Error Codes

WSANOTINITIALISED	A successful **WSAStartup()** must occur before using this API.
WSAENETDOWN	The network subsystem has failed.
WSAEACCES	The requested address is a broadcast address, but the appropriate flag was not set.
WSAEFAULT	The *lpBuffer* argument is not in a valid part of the user address space.
WSAENETRESET	The connection must be reset because the Winsock provider dropped it.
WSAENOBUFS	The Winsock provider reports a buffer deadlock.
WSAENOTCONN	The socket is not connected.
WSAENOTSOCK	The descriptor is not a socket.
WSAEOPNOTSUPP	MSG_OOB was specified, but the socket is not stream style such as type SOCK_STREAM, out-of-band data is not supported in the communication domain associated with this socket, or the socket is unidirectional and supports only receive operations.
WSAESHUTDOWN	The socket has been shutdown; it is not possible to **WSASend()** on a socket after **shutdown()** has been invoked with how set to SD_SEND or SD_BOTH.
WSAEWOULDBLOCK	There are too many outstanding overlapped I/O requests.
WSAEMSGSIZE	The socket is message-oriented, and the message is larger than the maximum supported by the underlying transport.
WSAEINVAL	The socket has not been bound with **bind()**, or the socket is not created with the overlapped flag**.**
WSAECONNABORTED	The virtual circuit was aborted due to timeout or other failure.
WSAECONNRESET	The virtual circuit was reset by the remote side.

See Also **WSACloseEvent(),WSACreateEvent(), WSAGetOverlappedResult(), WSASocket(), WSAWaitForMultipleEvents()**

WSASendto()

Description Send data to a specific destination using overlapped I/O.

#include <winsock2.h>

int WSAAPI WSASendto (SOCKET *s*, **LPVOID** *lpBuffer*, **DWORD** *nNumberOfBytesToSend*, **LPDWORD** *lpNumberOfBytesSent*, **int** *iFlags,* **LPVOID** *lpTo,* **int** *iTolen,* **LPWSAOVERLAPPED** *lpOverlapped*, **LPWSAOVERLAPPED_COMPLETION_ROUTINE** *lpCompletionRoutine* **);**

s	A descriptor identifying a connected socket which was created using **WSASocket()** with flag WSA_FLAG_OVERLAPPED.
lpBuffer	A pointer to the buffer for the outgoing data.
nNumberOfBytesToSend	The number of bytes to send to the network.
lpNumberOfBytesSent	A pointer to the number of bytes sent by this call.
iFlags	Flags.
lpTo	An optional pointer to the address of the target socket.
iTolen	The size of the address in *lpTo*.
lpOverlapped	A pointer to a WSAOVERLAPPED structure.
lpCompletionRoutine	A pointer to the completion routine called when the send operation has been completed.

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

Note that the successful completion of a **WSASendto()** does not indicate that the data was successfully delivered. Upon completion of the overlapped operation, the value pointed to by the *lpNumberOfBytesSent* parameter is updated with the number of bytes sent.

WSASendto() is normally used on a connectionless socket to send a datagram to a specific peer socket identified by the *lpTo* parameter. On a connection-oriented socket, the *lpTo* and *iTolen* parameters are ignored; in this case the **WSASendto()** is equivalent to **WSASend**().

This function may be called from within the completion routine of a previous **WSARead**(), **WSAReadFrom**(), **WSASend()** or **WSASendTo()** function. In Win16 environments, this function may also be called from within interrupt context provided that the XP1_INTERRUPT bit in the socket's PROTOCOL_INFO struct is TRUE.

The *lpOverlapped* parameter much be valid for the duration of the overlapped operation. The WSAOVERLAPPED structure has the following form:

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

If the *lpCompletionRoutine* parameter is NULL, the *hEvent* field of *lpOverlapped* must be an event object handle which is signaled when the overlapped operation completes. An application can use **WSAWaitForMultipleEvents()** or **WSAGetOverlappedResult()** to wait or poll on the event object.

If *lpCompletionRoutine* is not NULL, the *hEvent* field is ignored and can be used by the application to pass context information to the completion routine.

In Win16 environments, the service provider may invoke the application's completion routine from within an interrupt context. Application developers should assume that this will be the case and restrict any calls made from within the completion routine to those that are safe for interrupt context.

In Win32 environments, completion functions are invoked following the usual rules for asynchronous procedure calls or APCs. Specifically, the calling thread must be in an alertable wait (e.g., using **WSAWaitForMultipleEvents()**) in order for the completion routine to be invoked.. If the calling thread is not in an alertable wait when the overlapped operation has been completed, the system queues the completion routine call until the thread enters an alterable wait. The prototype of the completion routine is as follows:

VOID CALLBACK CompletionRoutine(**DWORD** *dwError***,**
DWORD *cbTransferred***, LPWSAOVERLAPPED** *lpOverlapped* **);**

CompletionRoutine is a placeholder for an application-defined or library-defined function name. *dwError* specifies the completion status for the overlapped operation as indicated by *lpOverlapped*. *cbTransferred* specifies the number of bytes sent. This function does not return a value.

Returning from this function allows invocation of another pending completion routine for this socket. In Win32 environments, all waiting completion routines are called before the alterable thread's wait is satisfied with a return code of WSA_IO_COMPLETION. The completion routines may be called in any order, not necessarily in the same order the overlapped operations are completed. However, the posted buffers are guaranteed to be sent in the same order they are supplied

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

Value	Meaning
MSG_DONTROUTE	Specifies that the data should not be subject to routing. A WINSOCK service provider may choose to ignore this flag; see also the discussion

of the SO_DONTROUTE option in section **Error! Reference source not found.**.

MSG_PARTIAL Specifies that *lpBuffer* only contains a partial message. Note that this flag will be ignored by transports which do not support partial message transmissions.

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

Error Codes

WSANOTINITIALISED	A successful **WSAStartup()** must occur before using this API.
WSAENETDOWN	The network subsystem has failed.
WSAEACCES	The requested address is a broadcast address, but the appropriate flag was not set.
WSAEFAULT	The *lpBuffer* or *lpTo* parameters are not part of the user address space, or the *lpTo* argument is too small.
WSAENETRESET	The connection must be reset because the Winsock provider dropped it.
WSAENOBUFS	The Winsock provider reports a buffer deadlock.
WSAENOTCONN	The socket is not connected (connection-oriented sockets only)
WSAENOTSOCK	The descriptor is not a socket.
WSAEOPNOTSUPP	MSG_OOB was specified, but the socket is not stream style such as type SOCK_STREAM, out-of-band data is not supported in the communication domain associated with this socket, or the socket is unidirectional and supports only receive operations.
WSAESHUTDOWN	The socket has been shutdown; it is not possible to **WSASendto()** on a socket after **shutdown()** has been invoked with how set to SD_SEND or SD_BOTH.
WSAEWOULDBLOCK	There are too many outstanding overlapped I/O requests.
WSAEMSGSIZE	The socket is message-oriented, and the message is larger than the maximum supported by the underlying transport.
WSAEINVAL	The socket has not been bound with **bind()**, or the socket is not created with the overlapped flag**.**
WSAECONNABORTED	The virtual circuit was aborted due to timeout or other failure.

WSAECONNRESET	The virtual circuit was reset by the remote side.
WSAEADDRNOTAVAIL	The specified address is not available from the local machine.
WSAEAFNOSUPPORT	Addresses in the specified family cannot be used with this socket.
WSAEDESTADDRREQ	A destination address is required.
WSAENETUNREACH	The network can't be reached from this host at this time.

See Also **WSACloseEvent(),WSACreateEvent(), WSAGetOverlappedResult(), WSASocket(), WSAWaitForMultipleEvents()**

WSASocket()

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

{ May be revised to use PROTOCOL_INFO as an input param in place of *af, type, protocol* parameters}

#include <winsock2.h>

SOCKET WSAAPI WSASocket (int *af***, int** *type***, int** protocol, **int** *iProviderID*, **int** *iFlags*)**;**

af	An address family specification.
type	A type specification for the new socket.
protocol	A particular protocol to be used with the socket, or 0 if the caller does not wish to specify a protocol.
iProviderID	The identifier of the service provider to be selected.
iFlags	The socket attribute specification.

Remarks **WSASocket()** causes a socket descriptor and any related resources to be allocated and bound to the transport service provider specified by *iProviderID*. The provider IDs of service providers can be obtained by using **WSAEnumProviders()**. If *iProviderID* is set to be -1, this indicates to the Winsock DLL that it should determine for itself which service provider to use based on the supplied *af*, *type*, and *protocol* parameters.

If *protocol* is not specified (i.e., equal to zero), the default for the specified socket type is used. However, the address family may be given as AF_UNSPEC (unspecified), in which case the *protocol* parameter must be specified. The protocol number to use is particular to the "communication domain" in which communication is to take place.

The *iFlags* parameter may be used to specify the attributes of the socket by or-ing any of the following Flags:

Flag	Meaning
WSA_FLAG_OVERLAPPED	This flag causes an overlapped socket to be created. Overlapped sockets must utilize **WSASend()**, **WSASendto()**, **WSARecv()**, **WSARecvfrom()** for I/O operations, and allows multiple of these to be initiated and in progress simultaneously. Overlapped sockets are always non-blocking.

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

The communications protocols used to implement a reliable, connection-oriented socket ensure that data is not lost or duplicated. If data for which the peer protocol has buffer space cannot be successfully transmitted within a reasonable length of time, the

connection is considered broken and subsequent calls will fail with the error code set to WSAETIMEDOUT.

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

Return Value

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

Error Codes

WSANOTINITIALISED	A successful **WSAStartup()** must occur before using this API.
WSAENETDOWN	The network subsystem has failed.
WSAEAFNOSUPPORT	The specified address family is not supported.
WSAEINPROGRESS	A blocking Winsock call is in progress, or the service provider is still processing a callback function (see section **Error! Reference source not found.**).
WSAEMFILE	No more socket descriptors are available.
WSAENOBUFS	No buffer space is available. The socket cannot be created.
WSAEPROTONOSUPPORT	The specified protocol is not supported.
WSAEPROTOTYPE	The specified protocol is the wrong type for this socket.
WSAESOCKTNOSUPPORT	The specified socket type is not supported in this address family.

See Also

accept(), **bind()**, **connect()**, **getsockname()**, **getsockopt()**, **setsockopt()**, **listen()**, **recv()**, **recvfrom()**, **select()**, **send()**, **sendto()**, **shutdown()**, **ioctlsocket()**.

WSACreateEvent()

Description Creates a new event object.

#include <winsock2.h>

WSAEVENT WSAAPI WSACreateEvent(VOID);

Remarks The event object created by this function is manual reset, with an initial state of nonsignaled. If a Win32 application desires auto reset events, it may call the native **CreateEvent()** Win32 API directly.

The Win32 implementation of this function is:

```
#define WSACreateEvent() \
        CreateEvent( NULL, FALSE, FALSE, NULL );
```

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

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

Error Codes ERROR_NOT_ENOUGH_MEMORY Not enough free memory available to create the event object.

See Also **WSACloseEvent().**

WSACloseEvent()

Description Closes an open event object handle.

#include <winsock2.h>

BOOL WSAAPI WSACloseEvent(WSAEVENT *hEvent* **);**

hEvent Identifies an open event object handle.

Remarks The Win32 implementation of this function is:

```
#define WSACloseEvent( h ) \
        CloseHandle( h )
```

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

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

Error Codes ERROR_INVALID_HANDLE *hEvent* is not a valid event object handle.

See Also **WSACreateEvent().**

WSAWaitForMultipleEvents()

Description Returns either when any one or when all of the specified event objects are in the signaled state, or when the time-out interval elapses.

#include <winsock2.h>

DWORD WSAAPI WSAWaitForMultipleEvents(DWORD *cEvents***, const WSAEVENT FAR *** *lphEvents***, BOOL** *fWaitAll***, DWORD** *dwTimeout***, BOOL** *fAlertable* **);**

cEvents Specifies the number of event object handles in the array pointed to by *lphEvents*. The maximum number of event object handles is WSA_MAXIMUM_WAIT_EVENTS.

lphEvents Points to an array of event object handles.

fWaitAll Specifies the wait type. If TRUE, the function returns when all event objects in the lphEvents array are signaled at the same time. If FALSE, the function returns when any one of the event objects is signaled. In the latter case, the return value indicates the event object whose state caused the function to return.

dwTimeout Specifies the time-out interval, in milliseconds. The function returns if the interval elapses, even if conditions specified by the *fWaitAll* parameter are not satisfied. If *dwTimeout* is zero, the function tests the state of the specified event objects and returns immediately. If *dwTimeout* is WSA_INFINITE, the function's time-out interval never elapses.

fAlertable Specifies whether the function returns when the system queues an I/O completion routine for execution by the calling thread. If TRUE, the function returns and the completion routine is executed. If FALSE, the function does not return and the completion routine is not executed. Note that this parameter is ignored in Win16.

Remarks In nonpreemptive environments (Win16), the blocking hook is called if this function must wait for the event(s) to become signaled. In preemptive environments (Win32), the blocking hook is never called from within this function.

The Win32 implementation for this function is:

```
#define WSAWaitForMultipleEvents( c, p, w, t, a ) \
        WaitForMultipleEventsEx( (c), (p), (w), (t), (a) )
```

Return Value If the function succeeds, the return value indicates the event object that caused the function to return.

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

The return value upon success is one of the following values:

Value Meaning
WSA_WAIT_OBJECT_0

to
(WSA_WAIT_OBJECT_0
+ cObjects - 1) If *fWaitAll* is TRUE, the return value indicates that the state of all specified event objects is signaled. If *fWaitAll* is FALSE, the return value minus WAIT_OBJECT_0 indicates the *lphEvents* array index of the object that satisfied the wait.

WAIT_IO_COMPLETION (Win32 only) One or more I/O completion routines are queued for execution.

WSA_WAIT_TIMEOUT The time-out interval elapsed and the conditions specified by the *fWaitAll* parameter are not satisfied.

Error Codes

ERROR_NOT_ENOUGH_MEMORY Not enough free memory available to complete the operation

ERROR_INVALID_HANDLE One or more of the values in the *lphEvents* array is not a valid event object handle.

ERROR_INVALID_PARAMETER The *cEvents* parameter does not contain a valid handle count.

See Also **WSACreateEvent(), WSASetEvent().**

WSASetEvent()

Description Sets the state of the specified event object to signaled.

#include <winsock2.h>

BOOL WSAAPI WSASetEvent(WSAEVENT *hEvent* **);**

hEvent Identifies an open event object handle.

Remarks The Win32 implementation of this function is:

```
#define WSASetEvent( h ) \
        SetEvent( h )
```

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

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

Error Codes ERROR_INVALID_HANDLE *hEvent* is not a valid event object handle.

See Also **WSACreateEvent(), WSAResetEvent().**

WSAResetEvent()

Description Resets the state of the specified event object to nonsignaled.

#include <winsock2.h>

BOOL WSAAPI WSAResetEvent(WSAEVENT *hEvent* **);**

hEvent Identifies an open event object handle.

Remarks The Win32 implementation of this function is:

```
#define WSAResetEvent( h )
        ResetEvent( h )
```

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

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

Error Codes ERROR_INVALID_HANDLE *hEvent* is not a valid event object handle.

See Also **WSACreateEvent(), WSASetEvent().**

WSAGetOverlappedResult()

Description Returns the results of an overlapped operation on the specified socket.

#include <winsock2.h>

BOOL WSAAPI WSAGetOverlappedResult(SOCKET *sock*, **LPWSAOVERLAPPED** *lpOverlapped*, **LPDWORD** *lpcbTransfer*, **BOOL** *fWait*, **LPDWORD** *lpdwFlags* **);**

sock	Identifies the socket. This is the same socket that was specified when the overlapped operation was started by a call to **WSARecv(), WSARecvFrom(), WSASend(), WSASendTo(), WSAConnect(), or WSAAccept()**.
lpOverlapped	Points to a WSAOVERLAPPED structure that was specified when the overlapped operation was started.
lpcbTransfer	Points to a 32-bit variable that receives the number of bytes that were actually transferred by a send or receive operation.
fWait	Specifies whether the function should wait for the pending overlapped operation to complete. If TRUE, the function does not return until the operation has been completed. If FALSE and the operation is still pending, the function returns FALSE and the WSAGetLastError function returns WSA_IO_INCOMPLETE.
lpdwFlags	Points to a 32-bit variable that will receive one or more flags that supplement the completion status. For example, if partial data is received over a message-oriented transport, this is indicated here.

Remarks The results reported by the **WSAGetOverlappedResult()** function are those of the specified socket's last overlapped operation to which the specified WSAOVERLAPPED structure was provided, and for which the operation's results were pending. A pending operation is indicated when the function that started the operation returns FALSE, and the **WSAGetLastError()** function returns WSA_IO_PENDING. When an I/O operation is pending, the function that started the operation resets the *hEvent* member of the WSAOVERLAPPED structure to the nonsignaled state. Then when the pending operation has been completed, the system sets the event object to the signaled state.

If the *fWait* parameter is TRUE, **WSAGetOverlappedResult()** determines whether the pending operation has been completed by waiting for the event object to be in the signaled state.

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

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

Error Codes

ERROR_INVALID_HANDLE	One of the handles involved with this operation is invalid. Either *sock* is not a valid socket handle, or the *hEvent* field of the WSAOVERLAPPED structure does not contain a valid event object handle.
ERROR_INVALID_PARAMETER	One of the parameters is unacceptable.

ERROR_IO_INCOMPLETE	*fWait* is FALSE and the I/O operation has not yet completed.

See Also **WSACreateEvent(), WSAWaitForMultipleEvents(), WSARecv(), WSARecvFrom(), WSASend(), WSASendTo(), WSAConnect(), WSAAccept().**

WSAGetQoSByName()

Description Initializes the QoS based on a template.

#include <winsock2.h>

BOOL WSAAPI WSAGetQoSByName(SOCKET *sock***, LPWSABUF** *lpQosName***, LPQOS** *lpQoS***);**

sock Identifies the socket. This is the same socket that was specified when the overlapped operation was started by a call to **WSARecv(), WSARecvFrom(), WSASend(), WSASendTo(), WSAConnect(), or WSAAccept()**.

lpQosName Specifies the QoS template name

Remarks Initializes the QoS structure based on a named template.

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

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

Error Codes

ERROR_INVALID_PARAMETER The specified QoS template is invalid.

See Also **WSAConnect(), WSAAccept(), getsockopt(), select(), WSAAsyncSelect().**

Winsock 2.0 Header File - Winsock2.h

```
/* WINSOCK2.H--definitions to be used with the WINSOCK2.DLL and WINSOCK2 applications.
 *
 * This header file corresponds to version 2.0 of the Winsock specification.
 *
 * This file includes parts which are Copyright (c) 1982-1986 Regents
 * of the University of California.  All rights reserved.  The
 * Berkeley Software License Agreement specifies the terms and
 * conditions for redistribution.
 */

#ifdef WIN32

#define WSATASK HANDLE

#else //WIN16

#define WSATASK HTASK

#endif // WIN32

typedef enum
{
       GuaranteedService,
       BestEffortService
} GUARANTEE;

typedef struct _flowparams
{
       int64          AverageBandwith;// In Bytes/sec
       int64          PeakBandwidth; // In Bytes/sec
       int64          BurstLength;   // In microseconds
       int64          Latency;       // In microseconds
       int64          DelayVariation;// In microseconds
       GUARANTEE      levelOfGuarantee;// Guaranteed or
                                     // Best Effort
       int32          CostOfCall;    // Reserved for future
                                     // use, must be set to 0
       int32          ProviderId;    // Provider Identifier
       int32          SizePSP;       // Length of provider
                                     // specific parameters
       UCHAR   ProviderSpecificParams[1];// provider specific
                                     // parameters
} FLOWPARAMS;

typedef struct _QualityOfService
{
       FLOWPARAMS     ForwardFP;     // Caller(Initiator) to callee
       FLOWPARAMS     BackwardFP;    // Callee to caller
} QOS, FAR * LPQOS;

typedef int (CALLBACK * LPCONDITIONPROC) (const struct sockaddr FAR * CallerName,
                                          int CallerNamelen,
                                          LPWSABUF lpCallerData,
                                          LPQOS lpSFlowspec,
                                          const struct sockaddr FAR * CalleeName
                                          int CalleeNamelen,
                                          LPWSABUF lpCalleeData,
                                          GROUP FAR * g,
                                          DWORD dwCallbackData);

typedef stuct _WSANETWORKEVENTS {
       long lNetworkEvent,
       int iErrorCode
} WSANETWORKEVENTS, LPWSANETWORKEVENTS;

#define   WSAEDISCON    ????
```