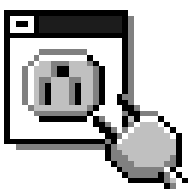
Windows Sockets 2 Application Programming Interface

An Interface for Transparent Network Programming Under Microsoft Windows™

Revision 2.0.5
Dec 23, 1994



Winsock 2

Preliminary
Subject to Change Without Notice

Draft

Disclaimer and Usage Restriction

Microsoft, Intel, and JSB disclaim all warranties and liability for the use of this document and the information contained herein, and assume 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.

This material is provided by Microsoft, Intel and JSB Corporations for use by persons and organizations who are participating in the Winsock Forum's efforts to define version 2 of the Winsock specification. Any other use is prohibited without specific permission.

Table of Contents

1. INTRODUCTION 1
- **1.1 Intended Audience 1**
- **1.2 Status of This Specification 1**
- **1.3 Document Version Conventions 1**

2. SUMMARY OF NEW CONCEPTS, ADDITIONS AND CHANGES 3
- **2.1 Simultaneous Access to Multiple Transport Protocols 3**
- **2.2 Backwards Compatibility For Winsock 1.1 Applications 3**
- **2.3 Making Transport Protocols Available To Winsock 4**
  - 2.3.1 Using Multiple Protocols 4
  - 2.3.2 Multiple Provider Restrictions on select() 5
- **2.4 Protocol Independent Name Resolution 5**
- **2.5 Overlapped I/O and Event Objects 5**
  - 2.5.1 Event Objects as an Underpinning for Completion Indication 6
- **2.6 Quality of Service 7**
  - 2.6.1 The Flow Spec Structures 7
  - 2.6.2 Default Values 8
- **2.7 Socket Groups 8**
- **2.8 Shared Sockets 9**
- **2.9 Enhanced Functionality During Connection Setup 9**
- **2.10 Extended Byte Order Conversion Routines 9**
- **2.11 Summary of New Socket Options 10**
- **2.12 Summary of New Functions 11**

3. SOCKET LIBRARY REFERENCE 12
- **3.1 accept() 12**
- **3.2 bind() 14**
- **3.3 closesocket() 16**
- **3.4 connect() 18**
- **3.5 gethostbyaddr() 21**
- **3.6 gethostbyname() 23**
- **3.7 gethostname() 24**
- **3.8 getprotobyname() 25**
- **3.9 getprotobynumber() 27**
- **3.10 getservbyname() 28**
- **3.11 getservbyport() 30**
- **3.12 getpeername() 31**
- **3.13 getsockname() 32**
- **3.14 getsockopt() 34**
- **3.15 htonl() 39**
- **3.16 htons() 40**
- **3.17 inet_addr() 41**
- **3.18 inet_ntoa() 42**
- **3.19 ioctlsocket() 43**
- **3.20 listen() 45**
- **3.21 ntohl() 47**
- **3.22 ntohs() 48**
- **3.23 recv() 49**
- **3.24 recvfrom() 52**
- **3.25 select() 55**

3.26 send() **58**
3.27 sendto() **61**
3.28 setsockopt() **64**
3.29 shutdown() **68**
3.30 socket() **70**
3.31 WSAAccept() **72**
3.32 WSAAsyncGetHostByAddr() **76**
3.33 WSAAsyncGetHostByName() **79**
3.34 WSAAsyncGetProtoByName() **82**
3.35 WSAAsyncGetProtoByNumber() **85**
3.36 WSAAsyncGetServByName() **88**
3.37 WSAAsyncGetServByPort() **91**
3.38 WSAAsyncSelect() **94**
3.39 WSACancelAsyncRequest() **100**
3.40 WSACancelBlockingCall() **102**
3.41 WSACleanup() **104**
3.42 WSACloseEvent() **105**
3.43 WSAConnect() **106**
3.44 WSACreateEvent() **111**
3.45 WSADuplicateSocket() **112**
3.46 WSAEnumNetworkEvents() **114**
3.47 WSAEnumProtocols() **116**
3.48 WSAEventSelect() **120**
3.49 WSAGetLastError() **125**
3.50 WSAGetOverlappedResult() **126**
3.51 WSAGetQoSByName() **128**
3.52 WSAHtonl() **129**
3.53 WSAHtons() **130**
3.54 WASIsBlocking() **131**
3.55 WSANtohl() **132**
3.56 WSANtohs() **133**
3.57 WSARecv() **134**
3.58 WSARecvfrom() **138**
3.59 WSAResetEvent() **142**
3.60 WSASend() **143**
3.61 WSASendto() **147**
3.62 WSASetBlockingHook() **151**
3.63 WSASetEvent() **153**
3.64 WSASetLastError() **154**
3.65 WSASocket() **155**
3.66 WSAStartup() **157**
3.67 WSAUnhookBlockingHook() **161**
3.68 WSAWaitForMultipleEvents() **162**

APPENDIX A. ERROR CODES AND HEADER FILES **164**
A.1 Error Codes **164**
A.2 Header Files **166**
A.2.1 Berkeley Header Files 166
A.2.2 Winsock Header File - Winsock.h 167

APPENDIX B. WINDOWS SOCKETS INTRODUCTION **172**
B.1 Berkeley Sockets **172**
B.2 Microsoft Windows and Windows-specific extensions **172**
B.3 PROGRAMMING WITH SOCKETS **172**

B.3.1 Winsock Installation Checking 172
B.3.2 Sockets 172
B.3.2.1 Socket Types 172
B.3.2.2 Client-server model 173
B.3.2.3 Out-of-band data 173
B.3.2.4 Broadcasting 174
B.3.3 Byte Ordering 174
B.3.4 Socket Options 175
B.3.5 Database Files 175
B.3.6 Deviation from Berkeley Sockets 175
B.3.6.1 socket data type and error values 175
B.3.6.2 select() and FD_* 176
B.3.6.3 Error codes - errno, h_errno & WSAGetLastError() 176
B.3.6.4 Pointers 177
B.3.6.5 Renamed functions 177
B.3.6.6 Blocking routines & EINPROGRESS 177
B.3.6.7 Maximum number of sockets supported 177
B.3.6.8 Include files 178
B.3.6.9 Return values on API failure 178
B.3.6.10 Raw Sockets 178
B.3.7 Winsock in Multithreaded Versions of Windows 178
B.4 Socket Library Overview 179
B.4.1 Socket Functions 179
B.4.1.1 Blocking/Non blocking & Data Volatility 179
B.4.2 Database Functions 180
B.4.3 Microsoft Windows-specific Extension Functions 181
B.4.3.1 Asynchronous select() Mechanism 183
B.4.3.2 Asynchronous Database Routines 183
B.4.3.3 Hooking Blocking Methods 183
B.4.3.4 Error Handling 183
B.4.3.5 Accessing Winsock DLL from an Intermediate DLL 184

APPENDIX C. OPEN ISSUES 185

1. Introduction

The Windows Sockets 2 specification is a superset of the widely deployed Windows Sockets 1.1 interface. While maintaining full backwards compatibility it extends the Winsock interface in a number of areas including

- Access to protocols other than TCP/IP: Winsock 2 allows an application to use the familiar socket interface to achieve simultaneous access to any number of installed transport protocols.
- Protocol-independent name resolution facilities: Winsock 2 includes a standardized set of APIs for querying and working with the myriad of name resolution domains that exist today (e.g. DNS, SAP, X.500, etc.)
- Overlapped I/O: following the model established in Win32 environments, Winsock 2 incorporates the overlapped paradigm for socket I/O.
- Quality of service: Winsock 2 establishes conventions for applications to negotiate required service levels for parameters such as bandwidth and latency. Other QOS-related enhancements include socket grouping and prioritization, and mechanisms for network-specific QOS extensions.
- Other frequently requested extensions: shared sockets, conditional acceptance, exchange of user data at connection setup/teardown time, protocol-specific extension mechanisms.

1.1 Intended Audience

This document is targeted at persons who are familiar with the sockets network programming paradigm in general and, to a lesser degree, the Windows Sockets version 1.1 interface in particular. Please see appendix B for an introduction to Winsock programming.

Persons who are interested in developing applications that will take advantage of Winsock 2's capabilities will be primarily interested in this API specification. Persons who are interested in making a particular transport protocol available under the Winsock 2 interface will need to be familiar with the Winsock 2 Service Provider Interface Specification as well. This document exists under separate cover.

1.2 Status of This Specification

This document comprises only the API portion of the Winsock 2 specification. The Winsock Forum's Generic API Extensions group, chaired by David Andersen of the Intel Architecture Labs has responsibility for producing and updating this document. The companion SPI specification is being produced by the Operating Framework group, chaired by Keith Moore of Microsoft Corporation. Constructive comments and feedback on this material are being actively solicited and should be directed towards the principle authors as shown below:

API Specification
David B. Andersen
Intel Architecture Labs
david_b_andersen@ccm.jf.intel.com

SPI Specification
Keith Moore
Microsoft Corporation
keithmo@microsoft.com

At the time of this writing, there are a number of open issues that have not been resolved. These are summarized in Appendix C.

1.3 Document Version Conventions

Starting with this release, the API and SPI documents have adopted a 3-part revision identification system. Each revision of the document will be clearly labelled with a release date and a revision identifier such as *X.Y.Z* where:

X is the *major version* of the Winsock specification (currently version 2)

Y is a *major revision* identifier that is incremented each time changes are made that impact binary compatibility with the previous spec revision (e.g. changes in a function's parameter list or new functions being added)

Z is a *minor revision* indicator that is incremented when wording changes or clarifications have been made which do not impact binary compatability with a previous revision.

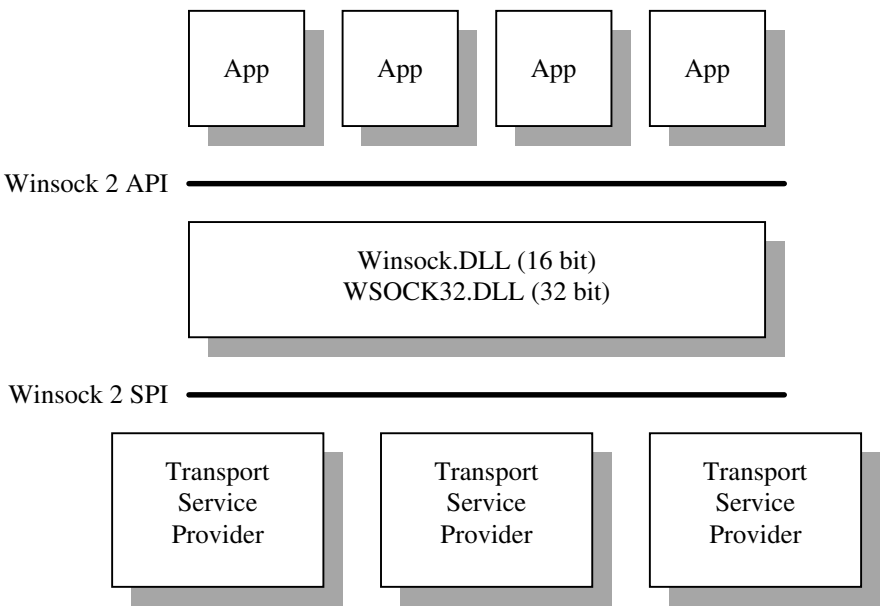
Note that the draft versions of the API and SPI specs distributed at the Dec 12, 1994 Winsock meeting did **not** follow these numbering conventions. Also note that gaps in the minor revision indicator (*Z*) between successive releases of a document are not unusal, especially during the early stages of a document's life when many changes are occuring.

2. Summary of New Concepts, Additions and Changes

The paragraphs which follow summarize the major changes and additions in going from Winsock 1.1 to Winsock 2. For detailed information about how to use a specific function or feature, please refer to the appropriate API description(s) in section 3.

2.1 Simultaneous Access to Multiple Transport Protocols

Winsock 2 provides simultaneous access to multiple transport protocols by changing the Winsock architecture. With Winsock 1.1, the DLL which implements the Winsock interface is supplied by the vendor of the TCP/IP protocol stack. The interface between the Winsock DLL and the underlying stack was both unique and proprietary. Winsock 2 changes the model by defining a standard interface between the Winsock DLL and protocol stacks. This makes it possible for multiple stacks from multiple vendors to be accessed simultaneously from a single Winsock DLL. Furthermore, Winsock 2 support is not limited to TCP/IP protocol stacks as is the case for Winsock 1.1. The Winsock 2 architecture (which is WOSA compliant) is illustrated as follows:



Note: 16 bit applications utilize Winsock.DLL, 32 bit applications use WSOCK32., but hereafter, for simplicity's sake both will be referred to simply as Winsock.DLL. This is reasonable since there are **no** syntactic differences between the two.

With the above architecture, it is no longer necessary (or even desirable) for each stack vendor to supply their own implementation of Winsock.DLL, since a single Winsock.DLL must work across all stacks. The Winsock DLL should thus be viewed in the same light as an operating system component. Microsoft has committed to create a Winsock.DLL for Winsock 2 which will be freely available for both Windows 95 and Windows NT operating systems. Intel Corporation has expressed a willingness to produce a Winsock 2 compliant Winsock.DLL for Windows 3.1 and Windows 3.11 environments, provided that sufficient demand is evidenced. Please refer to Appendix Y for a discussion of open issues surrounding the development and source control of these DLLs.

2.2 Backwards Compatibility For Winsock 1.1 Applications

Even though the underlying Winsock architecture has changed, this will be invisible to existing applications that use Winsock 1.1. Version 2 of Winsock.DLL will be completely backwards compatible

with version 1.1, assuming that at least one TCP/IP stack has been installed on the system and registered with Winsock 2. In the WSAStartup() sequence, a Winsock 1.1 app will ask for version 1.1. The Winsock 2 compliant DLL will cheerfully agree since all of the Winsock 1.1 semantics are retained in Winsock 2. The new Winsock DLL does not need to modify its behavior (other than returning the desired version number in the WSAData structure) regardless of whether version 1.1 or version 2 is requested by an application, as only Winsock 2 applications will know how to make use of the added functionality that is present.

Winsock 1.1 applications currently use certain elements from the **WSAData** structure (obtained via a call to **WSAStartup()**) to obtain information about the underlying TCP/IP stack.. These include: *iMaxSockets, iMaxUdpDg,* and *lpVendorInfo*. While Winsock 2 applications will know to ignore these values (since they cannot uniformly apply to all available protocol stacks), meaningful values must of course still be supplied to avoid breaking Winsock 1.1 applications. The information supplied will be obtained from (and hence only be applicable to) the "default" TCP/IP stack. The method for designating the default TCP/IP stack is TBD.

2.3 Making Transport Protocols Available To Winsock

In order for a transport protocol to be accessible via Winsock it must be properly installed on the system and registered with Winsock. The version 2 Winsock.DLL will export a set of APIs which perform the registration process. These include creating a new registration and removing an existing one. When new registrations are created the caller (presumed to be the stack vendor's installation script) supplies one or more filled in PROTOCOL_INFO structs which contain a complete set of information about the protocol. . Please refer to the SPI document for details on how this is accomplished. Note that transport stacks which are thus installed are considered to be Winsock service providers, and in this document are frequently referred to as such

2.3.1 Using Multiple Protocols

An application may use **WSAEnumProtocols()** to discover which transport protocols are present and obtain information about each as contained in the associated PROTOCOL_INFO struct. In most instances, there will be a single PROTOCOL_INFO struct for each protocol. Some protocols however, are able to exhibit multiple behaviors. For example the SPX protocol is message-oriented (i.e. the sender's message boundaries are preserved by the network), but the receiving end may choose to ignore these message boundaries and treat the socket as a byte stream. Thus there could reasonably be two different PROTOCOL_INFO struct entries for SPX, one for each of these behaviors.

Whereas in Winsock 1 there is a single address family (AF_INET) comprising a small number of well-known socket types and protocol identifiers, the focus will shift for Winsock 2. The existing address family, 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. Not being well known need not pose a problem since applications that desire to be protocol-independent are encouraged to select protocols for use on the basis of their suitability rather than the particular values assigned to their *socket_type* or *protocol* fields Protocol suitability is indicated by the communications attributes (e.g. message vs. byte-stream oriented, reliable vs. unreliable, etc.) contained within the protocol's PROTOCOL_INFO struct.. Selecting protocols on the basis of suitability as opposed to well-known protocol names and socket types allows protocol-independent applications to take advantage of new transport protocols and their associated media types as they become available.

In terms of the well-known client/server paradigm, the server half of a client/server application will benefit by establishing listening sockets on <u>all</u> suitable transport protocols. The client, then, may establish its connection using <u>any</u> suitable protocol. This would enable, for example, a client application to be unmodified whether it was running on a desktop system connected via LAN or on a laptop using a wireless network.

It is planned that a Winsock 2 clearinghouse be established for protocol stack vendors to obtain unique identifiers for new address families, socket types and protocols. FTP and world-wide web servers will supply current identifier/value mappings, and an email procedure will be used to request allocation of new ones. JSB Corporation has indicated a willingness to initially establish and operate this clearinghouse service.

2.3.2 Multiple Provider Restrictions on select()

In Winsock 2 the FD_SET supplied to the select() function will be constrained to contain sockets associated with a single service provider. 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 descriptor as an input parameter, it doesn't matter what provider is associated with the socket. When an application needs to use blocking semantics on a set of sockets which spans multiple providers, the recommended solution is to use **WSAWaitForMultipleEvents()**. The application may also choose to take advantage of the **WSAEventSelect()** function which allows the FD_XXX network events to be associated with an event object and handled from within the event object paradigm (described below).

2.4 Protocol Independent Name Resolution

Winsock 2 will include APIs which standardize the way applications access and use the various network naming services. Thus an application will not need to be cognizant of the widely differing interfaces associated with name services such as DNS, NIS, X.500, SAP, etc. Details on these APIs are not included in this specification at this time, and will be supplied at a later date by the Winsock Forum's Name Resolution functionality group.

2.5 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, applications use **WSARecv()** or **WSARecvFrom()** to supply buffers into which data is to be received. If one or more buffers are posted prior to the time when data has been received by the network, data will be placed into the user's buffers immediately as it arrives and thereby avoid the copy operation that would otherwise occur at the time the **recv()** or **recvfrom()** function is invoked. If data is already present when receive buffers are posted, it is copied immediately into the user's buffers. If data arrives when no receive buffers have been posted by the application, the network resorts to the familiar synchronous style of operation 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, reliable protocols would only allow data to be received when application buffers had been posted, and data on unreliable protocols would be lost.

On the sending side, applications use **WSASend()** or **WSASendTo()** to supply pointers to filled buffers and then agree to not disturb the buffers in any way until 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.

2.5.1 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. In Winsock 2 this is accomplished via event objects which are modeled after Win32 events. Applications use **WSACreateEvent ()** to obtain an event object handle which may then be supplied as a required parameter to the overlapped versions of send and receive calls (**WSASend(), WSASendTo(), WSARecv(), WSARecvFrom()**). The event object, which is cleared when first created, is set by the 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 socket I/O completion routines.

Blocking and waiting for Completion Indication -
Applications may also choose to block while waiting for one or more event objects 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 whether or not the overlapped operation has completed, and, if completed, arrange for **WSAGetLastError()** to retrieve the error status of the overlapped operation.

Using socket I/O completion routines -
The functions used to initiate overlapped I/O (**WSASend, WSASendTo, WSARecv, WSARecvFrom**) all take *lpCompletionRoutine* as an 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 a preemptive VMM context which is sometimes referred to as an interrupt context. Applications transmitting time-sensitive data (e.g. video and audio) are benefited by being able to receive such indications in this low latency, preemptive manner, but must be aware that in this special context a very limited set of runtime and Windows library functions can be safely made. As a rule, an application should confine itself to the same set of runtime functions that the Windows documentation indicates may safely be called during a multimedia timer callback function.

In Windows 95 and NT, the completion function follows the same rules as stipulated for Win32 file I/O completion routines. The completion function will not be invoked until the thread is in an alertable wait state such as can occur when the function **WSAWaitForMultipleEvents()** is invoked.

In all environments, transports do allow an application to invoke send and receive operations from within the context of the socket I/O completion function, and guarantee that, for a given socket, I/O completion functions will not be nested. This permits time-sensitive data transmissions to occur entirely within a preemptive context.

2.6 Quality of Service

Note: the QOS material is undergoing extensive review and modification. The following paragraphs are almost guaranteed to be wrong.

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. In conjunction with this, two new network events have been defined: FD_QOS and FD_GROUP_QOS. Upon being notified of either of these events, 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.

2.6.1 The Flow Spec Structures

The Winsock 2 flow spec structure is defined in Winsock.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;
```

2.6.2 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
```

2.7 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 or circuit connection. The sockets within a group are then multiplexed in the usual manner over this call. By allowing the application to identify the sockets that make up a group and to specify the required group attributes, 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.

2.8 Shared Sockets

WSADuplicateSocket() is introduced to enable socket sharing across processes by creating an additional socket descriptor to an underlying socket which thus becomes shared. The function takes as input both the local socket descriptor and a handle to the target process. It returns a new descriptor for the socket which is only valid in the context of the target process (which is not precluded from being the same as the original process). 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). Note however, that sockets may be shared amongst threads in a given process without using the WSADuplicateSocket() function, since a socket descriptor is valid in all a process' threads.

The two (or more) descriptors that reference a shared socket may be used independently as far as I/O is concerned. However, the Winsock interface does not implement any type of access control, so it is up to the processes involved to coordinate their operations on a shared socket. A typical use for shared sockets is to have one process reading and a different process writing.

Since what is duplicated are the socket descriptors and not the underlying socket, all of the state associated with a socket is held in common across all the descriptors. For example a **setsockopt()** operation performed using one descriptor is subsequently visible using a **getsockopt()** from any or all descriptors. A process may call **closesocket()** on a duplicated socket and the descriptor will become deallocated. The underlying socket, however, will remain open until **closesocket()** is called by the last remaining descriptor.

The manner in which event notification is handled on shared sockets is currently under discussion. Two proposals have been made. In the first, the notification properties are associated completely with each descriptor and are thus entirely independent from one another. Assume processes A, B and C all have descriptors to a shared socket. It would be allowed for process A to register interest in FD_READ and FD_CLOSE, process B to register for FD_CLOSE and FD_WRITE, and process C to register for FD_READ and FD_WRITE. When data was available to be read on the socket, both process A and C would receive notification. Likewise, if the socket were closed by the remote end, both process A and B would be notified.

In the second proposal, any particular FD_XXX network event may be delivered only to a single process. Thus if process A registered for FD_READ and FD_CLOSE, process B would be precluded from registering for FD_CLOSE as well.

2.9 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 callback to an application-supplied condition function. User-to-user data specified via parameters in **WSAConnect()** and/or the condition function of **WSAAccept()** may be transferred to the peer during connection establishment, provided this feature is supported by the service provider.

2.10 Extended Byte Order Conversion Routines

Winsock 2 does not assume that the network byte order for all protocols is the same. Therefore a set of conversion routines are supplied for converting 16 and 32 bit quantities to and from network byte order.

These routines take as in input parameter a boolean which specifies whether the desired network byte order is “big-endian” or “little-endian”. Also, the PROTOCOL_INFO struct for each protocol includes a flag to indicate the byte ordering to be used for the protocol.

2.11 Summary of New Socket Options

The new socket options proposed for Winsock2 are summarized in the following table. More detailed information is provided in section 3 under **getsockopt()** and/or **setsockopt()**.

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_QOS	char FAR *	The QOS parameters 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_QOS	char FAR *	The QOS parameters 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	

2.12 Summary of New Functions

The new API functions proposed for Winsock2 are summarized in the following table.

WSAAccept() *	An extended version of accept() which allows for conditional acceptance and socket grouping.
WSACloseEvent()	Destroys an event object.
WSAConnect()	An extended version of connect() which allows for socket grouping and QOS specification.
WSACreateEvent()	Creates an event object.
WSADuplicateSocket()	Allow an underlying socket to be shared by creating a virtual socket.
WSAEnumNetworkEvents()	Discover occurrences of network events.
WSAEnumProtocols()	Retrieve information about each available protocol.
WSAEventSelect()	Associate network events with an event object.
WSAGetOverlappedResult()	Get completion status of overlapped operation.
WSAGetQOSByName()	Supply QOS parameters based on a well-known service name.
WSAHtonl()	Extended version of htonl()
WSAHtons()	Extended version of htons()
WSANtohl()	Extended version of ntohl()
WSANtohs()	Extended version of ntohs()
WSARecv()	An extended version of recv() which accommodates overlapped sockets and provides the *flags* parameter as IN OUT
WSARecvfrom()	An extended version of recvfrom() which accommodates overlapped sockets and provides the *flags* parameter as IN OUT
WSAResetEvent()	Resets an event object.
WSASend()	An extended version of send() which accommodates overlapped sockets
WSASendto()	An extended version of sendto() which accommodates overlapped sockets
WSASetEvent()	Sets an event object.
WSASocket()	An extended version of socket() which takes a PROTOCOL_INFO struct as input and allows overlapped sockets to be created.
WSAWaitForMultipleEvents()	Blocks on multiple event objects.

3. SOCKET LIBRARY REFERENCE

This chapter presents the socket library routines in alphabetical order, and describes each routine in detail.

In each routine it is indicated that the header file **winsock.h** must be included. Appendix A.2 lists the Berkeley-compatible header files which are supported. These are provided for compatibility purposes only, and each of them will simply include **winsock.h**. The Windows header file **windows.h** is also needed, but **winsock.h** will include it if necessary.

3.1 accept()

Description Accept a connection on a socket.

#include <winsock.h>

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

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*.

Remarks This routine extracts the first connection on the queue of pending connections on *s*, creates a new socket with the same properties as *s* and returns a handle to the new socket. If no pending connections are present on the queue, and the socket is not marked as non-blocking, **accept()** blocks the caller until a connection is present. If the socket is marked non-blocking and no pending connections are present on the queue, **accept()** 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.

Return Value If no error occurs, **accept**() returns a value of type SOCKET which is a descriptor for the accepted socket. Otherwise, a value of INVALID_SOCKET is returned, and a specific error code 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.
WSAENETDOWN	The network subsystem has failed.
WSAEFAULT	The *addrlen* argument is too small.
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 **B.3.6.6**).
WSAEINVAL	**listen()** was not invoked prior to **accept()**.
WSAEMFILE	The queue is non-empty upon entry to **accept()** and there are no descriptors available.
WSAENOBUFS	No buffer space is available.
WSAENOTSOCK	The descriptor is not a socket.
WSAEOPNOTSUPP	The referenced socket is not a type that supports connection-oriented service.
WSAEWOULDBLOCK	The socket is marked as non-blocking and no connections are present to be accepted.

See Also **bind()**, **connect()**, **listen()**, **select()**, **socket()**, **WSAAsyncSelect(), WSAAccept()**

3.2 bind()

Description Associate a local address with a socket.

#include <winsock.h>

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

s A descriptor identifying an unbound socket.

name The address to assign to the socket. The sockaddr structure is defined as follows:

```
struct sockaddr {
      u_short     sa_family;
      char        sa_data[14];
};
```

namelen The length of the *name*.

Remarks This routine is used on an unconnected connectionless or connection-oriented socket, before subsequent **connect**()s or **listen**()s. When a socket is created with **socket**(), it exists in a name space (address family), but it has no name assigned. **bind**() establishes the local association of the socket by assigning a local name to an unnamed socket.

As an example, in the Internet address family, a name consists of three parts: the address family, a host address, and a port number which identifies the application. In Winsock 2, the *name* parameter is not strictly interpreted as a pointer to a "sockaddr" struct. It is cast this way for Windows Sockets compatibility. Service Providers are free to regard it as a pointer to a block of memory of size *namelen*. The first two bytes in this block (corresponding to "sa_family" in the "sockaddr" declaration) must contain the address family that was used to create the socket. Otherwise an error WSAEFAULT will occur.

If an application does not care what local address is assigned to it, it may specify the manifest constant value ADDR_ANY for the sa_data field of the name parameter. This allows the underlying service provider to use any appropriate network address, potentially simplifying application programming in the presence of multi-homed hosts (i.e., hosts that have more than one network interface and address). For TCP/IP, if the port is specified as 0, the service provider will assign a unique port to the application with a value between 1024 and 5000. The application may use **getsockname()** after **bind()** to learn the address that has been assigned to it, but note that **getsockname()** will not necessarily be able to supply the address until the socket is connected, since several addresses may be valid if the host is multi-homed.

Return Value If no error occurs, **bind()** returns 0. Otherwise, it returns SOCKET_ERROR, 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.

WSAEADDRINUSE	The specified address is already in use. (See the SO_REUSEADDR socket option under **setsockopt()**.)
WSAEFAULT	The *namelen* argument is too small, the *name* argument contains incorrect address format for the associated address family, or the first two bytes of the memory block specified by *name* does not match the address family associate with the socket descriptor *s*.
WSAEINPROGRESS	A blocking Winsock call is in progress, or the service provider is still processing a callback function (see section **B.3.6.6**).
WSAEINVAL	The socket is already bound to an address.
WSAENOBUFS	Not enough buffers available, too many connections.
WSAENOTSOCK	The descriptor is not a socket.

See Also **connect()**, **listen()**, **getsockname()**, **setsockopt()**, **socket()**, **WSACancelBlockingCall()**.

3.3 closesocket()

Description Close a socket.

#include <winsock.h>

int WSAAPI closesocket (SOCKET *s* **);**

s A descriptor identifying a socket.

Remarks This function closes a socket. More precisely, it releases the socket descriptor *s*, so that further references to *s* will fail with the error WSAENOTSOCK. If this is the last reference to an underlying socket, the associated naming information and queued data are discarded.

The semantics of **closesocket**() are affected by the socket options SO_LINGER and SO_DONTLINGER as follows:

Option	Interval	Type of close	Wait for close?
SO_DONTLINGER	Don't care	Graceful	No
SO_LINGER	Zero	Hard	No
SO_LINGER	Non-zero	Graceful	Yes

If SO_LINGER is set (i.e. the *l_onoff* field of the linger structure is non-zero; see sections B.3.4 , 3.14 and 3.28) with a zero timeout interval (*l_linger* is zero), **closesocket()** is not blocked even if queued data has not yet been sent or acknowledged. This is called a "hard" or "abortive" close, because the socket's virtual circuit is reset immediately, and any unsent data is lost. Any **recv()** call on the remote side of the circuit will fail with WSAECONNRESET.

If SO_LINGER is set with a non-zero timeout interval, the **closesocket()** call blocks until the remaining data has been sent or until the timeout expires. This is called a graceful disconnect. Note that if the socket is set to non-blocking and SO_LINGER is set to a non-zero timeout, the call to **closesocket()** will fail with an error of WSAEWOULDBLOCK.

If SO_DONTLINGER is set on a stream socket (i.e. the *l_onoff* field of the linger structure is zero; see sections B.3.4 , 3.14 and 3.28), the **closesocket()** call will return immediately. However, any data queued for transmission will be sent if possible before the underlying socket is closed. This is also called a graceful disconnect. Note that in this case the Winsock provider may not release the socket and other resources for an arbitrary period, which may affect applications which expect to use all available sockets.

Return Value If no error occurs, **closesocket()** 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.

WSAENOTSOCK	The descriptor is not a socket.
WSAEINPROGRESS	A blocking Winsock call is in progress, or the service provider is still processing a callback function (see section **B.3.6.6**).
WSAEINTR	The (blocking) call was canceled via **WSACancelBlockingCall().**
WSAEWOULDBLOCK	The socket is marked as nonblocking and SO_LINGER is set to a nonzero timeout value.

See Also **accept()**, **socket()**, **ioctlsocket()**, **setsockopt()**, **WSAAsyncSelect(), WSADuplicateSocket()**.

3.4 connect()

Description Establish a connection to a peer.

#include <winsock.h>

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

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*.

Remarks This function is used to create a connection to the specified destination. 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 the socket call completes successfully, the socket is ready to send/receive data.

For a connectionless socket (e.g., type SOCK_DGRAM), the operation performed by **connect()** is merely to establish a default destination address which will be used on subsequent **send**() and **recv**() calls.

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

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, **connect()** 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 **B.3.6.6**).
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, or the *name* argument contains incorrect address format for the associated address family.
WSAEINVAL	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.
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(), getsockname(), socket(), select(), WSAAsyncSelect(), WSAConnect().**

3.5 gethostbyaddr()

Description Get host information corresponding to an address.

#include <winsock.h>

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

addr	A pointer to an address in network byte order.
len	The length of the address.
type	The type of the address.

Remarks **gethostbyaddr**() returns a pointer to the following structure which contains the name(s) and address which correspond to the given address.

```
struct hostent {
      char FAR *  h_name;
      char FAR * FAR *  h_aliases;
      short h_addrtype;
      short h_length;
      char FAR * FAR *  h_addr_list;
};
```

The members of this structure are:

Element	Usage
h_name	Official name of the host (PC).
h_aliases	A NULL-terminated array of alternate names.
h_addrtype	The type of address being returned.
h_length	The length, in bytes, of each address.
h_addr_list	A NULL-terminated list of addresses for the host. Addresses are returned in network byte order.

The macro h_addr is defined to be h_addr_list[0] for compatibility with older software.

The pointer which is returned points to a structure which is allocated by Winsock . The application must never attempt to modify this structure or to free any of its components. Furthermore, only one copy of this structure is allocated per thread, and so the application should copy any information which it needs before issuing any other Winsock API calls.

Return Value If no error occurs, **gethostbyaddr()** returns a pointer to the hostent structure described above. Otherwise it returns a NULL pointer and a specific error number may be retrieved by calling **WSAGetLastError().**

Error Codes

WSANOTINITIALISED	A successful **WSAStartup()** must occur before using this API.
WSAENETDOWN	The network subsystem has failed.

WSAHOST_NOT_FOUND	Authoritative Answer Host not found.
WSATRY_AGAIN	Non-Authoritative Host not found, or SERVERFAIL.
WSANO_RECOVERY	Non recoverable errors, FORMERR, REFUSED, NOTIMP.
WSANO_DATA	Valid name, no data record of requested type.
WSAEINPROGRESS	A blocking Winsock call is in progress, or the service provider is still processing a callback function (see section **B.3.6.6**).
WSAEINTR	The (blocking) call was canceled via **WSACancelBlockingCall().**

See Also **WSAAsyncGetHostByAddr()**, **gethostbyname**(),

3.6 gethostbyname()

Description Get host information corresponding to a hostname.

#include <winsock.h>

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

name A pointer to the name of the host.

Remarks **gethostbyname**() returns a pointer to a hostent structure as described under **gethostbyaddr**(). The contents of this structure correspond to the hostname *name*.

The pointer which is returned points to a structure which is allocated by Winsock. The application must never attempt to modify this structure or to free any of its components. Furthermore, only one copy of this structure is allocated per thread, and so the application should copy any information which it needs before issuing any other Winsock API calls.

A **gethostbyname()** implementation must not resolve IP address strings passed to it. Such a request should be treated exactly as if an unknown host name were passed. An application with an IP address string to resolve should use **inet_addr()** to convert the string to an IP address, then **gethostbyaddr()** to obtain the hostent structure.

Return Value If no error occurs, **gethostbyname()** returns a pointer to the hostent structure described above. Otherwise it returns a NULL pointer and a specific error number may be retrieved by calling **WSAGetLastError()**.

Error Codes

WSANOTINITIALISED	A successful **WSAStartup()** must occur before using this API.
WSAENETDOWN	The network subsystem has failed.
WSAHOST_NOT_FOUND	Authoritative Answer Host not found.
WSATRY_AGAIN	Non-Authoritative Host not found, or SERVERFAIL.
WSANO_RECOVERY	Non recoverable errors, FORMERR, REFUSED, NOTIMP.
WSANO_DATA	Valid name, no data record of requested type.
WSAEINPROGRESS	A blocking Winsock call is in progress, or the service provider is still processing a callback function (see section **B.3.6.6**).
WSAEINTR	The (blocking) call was canceled via **WSACancelBlockingCall().**

See Also **WSAAsyncGetHostByName**(), **gethostbyaddr**()

3.7 gethostname()

Description Return the standard host name for the local machine.

#include <winsock.h>

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

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

namelen The length of the buffer.

Remarks This routine returns the name of the local host into the buffer specified by the *name* parameter. The host name is returned as a null-terminated string. The form of the host name is dependent on the Winsock provider--it may be a simple host name, or it may be a fully qualified domain name. However, it is guaranteed that the name returned will be successfully parsed by **gethostbyname()** and **WSAAsyncGetHostByName().**

Return Value If no error occurs, **gethostname()** returns 0, otherwise it returns SOCKET_ERROR and a specific error code may be retrieved by calling **WSAGetLastError()**.

Error Codes

WSAEFAULT	The *namelen* parameter is too small
WSANOTINITIALISED	A successful **WSAStartup()** must occur before using this API.
WSAENETDOWN	The network subsystem has failed.
WSAEINPROGRESS	A blocking Winsock call is in progress, or the service provider is still processing a callback function (see section **B.3.6.6**).

See Also **gethostbyname(), WSAAsyncGetHostByName().**

3.8 getprotobyname()

Description Get protocol information corresponding to a protocol name.

#include <winsock.h>

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

name A pointer to a protocol name.

Remarks **getprotobyname**() returns a pointer to the following structure which contains the name(s) and protocol number which correspond to the given protocol *name*.

```
struct protoent {
      char FAR *  p_name;
      char FAR * FAR *  p_aliases;
      short p_proto;
};
```

The members of this structure are:

Element	Usage
p_name	Official name of the protocol.
p_aliases	A NULL-terminated array of alternate names.
p_proto	The protocol number, in host byte order.

The pointer which is returned points to a structure which is allocated by the Winsock library. The application must never attempt to modify this structure or to free any of its components. Furthermore only one copy of this structure is allocated per thread, and so the application should copy any information which it needs before issuing any other Winsock API calls.

Return Value If no error occurs, **getprotobyname()** returns a pointer to the protoent structure described above. Otherwise it returns a NULL pointer and a specific error number may be retrieved by calling **WSAGetLastError()**.

Error Codes

WSANOTINITIALISED	A successful **WSAStartup()** must occur before using this API.
WSAENETDOWN	The network subsystem has failed.
WSANO_RECOVERY	Non recoverable errors, FORMERR, REFUSED, NOTIMP.
WSANO_DATA	Valid name, no data record of requested type.
WSAEINPROGRESS	A blocking Winsock call is in progress, or the service provider is still processing a callback function (see section **B.3.6.6**).
WSAEINTR	The (blocking) call was canceled via **WSACancelBlockingCall().**

See Also **WSAAsyncGetProtoByName()**, **getprotobynumber()**

3.9 getprotobynumber()

Description Get protocol information corresponding to a protocol number.

#include <winsock.h>

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

number A protocol number, in host byte order.

Remarks This function returns a pointer to a protoent structure as described above in **getprotobyname()**. The contents of the structure correspond to the given protocol number.

The pointer which is returned points to a structure which is allocated by Winsock. The application must never attempt to modify this structure or to free any of its components. Furthermore, only one copy of this structure is allocated per thread, and so the application should copy any information which it needs before issuing any other Winsock API calls.

Return Value If no error occurs, **getprotobynumber()** returns a pointer to the protoent structure described above. Otherwise it returns a NULL pointer and a specific error number may be retrieved by calling **WSAGetLastError()**.

Error Codes

WSANOTINITIALISED	A successful **WSAStartup()** must occur before using this API.
WSAENETDOWN	The network subsystem has failed.
WSANO_RECOVERY	Non recoverable errors, FORMERR, REFUSED, NOTIMP.
WSANO_DATA	Valid name, no data record of requested type.
WSAEINPROGRESS	A blocking Winsock call is in progress, or the service provider is still processing a callback function (see section **B.3.6.6**).
WSAEINTR	The (blocking) call was canceled via **WSACancelBlockingCall().**

See Also **WSAAsyncGetProtoByNumber()**, **getprotobyname()**

3.10 getservbyname()

Description Get service information corresponding to a service name and protocol.

#include <winsock.h>

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

name	A pointer to a service name.
proto	An optional pointer to a protocol name. If this is NULL, **getservbyname()** returns the first service entry for which the *name* matches the s_name or one of the s_aliases. Otherwise **getservbyname()** matches both the *name* and the *proto*.

Remarks **getservbyname()** returns a pointer to the following structure which contains the name(s) and service number which correspond to the given service *name*.

```
struct servent {
      char FAR *  s_name;
      char FAR * FAR *  s_aliases;
      short s_port;
      char FAR *  s_proto;
};
```

The members of this structure are:

Element	Usage
s_name	Official name of the service.
s_aliases	A NULL-terminated array of alternate names.
s_port	The port number at which the service may be contacted. Port numbers are returned in network byte order.
s_proto	The name of the protocol to use when contacting the service.

The pointer which is returned points to a structure which is allocated by the Winsock library. The application must never attempt to modify this structure or to free any of its components. Furthermore only one copy of this structure is allocated per thread, and so the application should copy any information which it needs before issuing any other Winsock API calls.

Return Value If no error occurs, **getservbyname()** returns a pointer to the servent structure described above. Otherwise it returns a NULL pointer and a specific error number may be retrieved by calling **WSAGetLastError().**

Error Codes

WSANOTINITIALISED	A successful **WSAStartup()** must occur before using this API.
WSAENETDOWN	The network subsystem has failed.
WSANO_RECOVERY	Non recoverable errors, FORMERR, REFUSED, NOTIMP.

WSANO_DATA	Valid name, no data record of requested type.
WSAEINPROGRESS	A blocking Winsock call is in progress, or the service provider is still processing a callback function (see section **B.3.6.6**).
WSAEINTR	The (blocking) call was canceled via **WSACancelBlockingCall().**

See Also **WSAAsyncGetServByName()**, **getservbyport()**

3.11 getservbyport()

Description Get service information corresponding to a port and protocol.

#include <winsock.h>

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

port	The port for a service, in network byte order.
proto	An optional pointer to a protocol name. If this is NULL, **getservbyport()** returns the first service entry for which the *port* matches the s_port. Otherwise **getservbyport()** matches both the *port* and the *proto*.

Remarks **getservbyport()** returns a pointer a servent structure as described above for **getservbyname()**.

The pointer which is returned points to a structure which is allocated by Winsock. The application must never attempt to modify this structure or to free any of its components. Furthermore, only one copy of this structure is allocated per thread, and so the application should copy any information which it needs before issuing any other Winsock API calls.

Return Value If no error occurs, **getservbyport()** returns a pointer to the servent structure described above. Otherwise it returns a NULL pointer and a specific error number may be retrieved by calling **WSAGetLastError().**

Error Codes

WSANOTINITIALISED	A successful **WSAStartup()** must occur before using this API.
WSAENETDOWN	The network subsystem has failed.
WSANO_RECOVERY	Non recoverable errors, FORMERR, REFUSED, NOTIMP.
WSANO_DATA	Valid name, no data record of requested type.
WSAEINPROGRESS	A blocking Winsock call is in progress, or the service provider is still processing a callback function (see section **B.3.6.6**).
WSAEINTR	The (blocking) call was canceled via **WSACancelBlockingCall().**

See Also **WSAAsyncGetServByPort()**, **getservbyname()**

3.12 getpeername()

Description Get the address of the peer to which a socket is connected.

#include <winsock.h>

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

s	A descriptor identifying a connected socket.
name	The structure which is to receive the name of the peer.
namelen	A pointer to the size of the *name* structure.

Remarks **getpeername()** retrieves the name of the peer connected to the socket *s* and stores it in the struct sockaddr identified by *name*. It is used on a connected socket.

On return, the *namelen* argument contains the actual size of the name returned in bytes.

Return Value If no error occurs, **getpeername()** 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 namelen argument is not large enough.
WSAEINPROGRESS	A blocking Winsock call is in progress, or the service provider is still processing a callback function (see section **B.3.6.6**).
WSAENOTCONN	The socket is not connected.
WSAENOTSOCK	The descriptor is not a socket.

See Also **bind()**, **socket()**, **getsockname()**.

3.13 getsockname()

Description Get the local name for a socket.

#include <winsock.h>

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

s	A descriptor identifying a bound socket.
name	Receives the address (name) of the socket.
namelen	The size of the *name* buffer.

Remarks **getsockname()** retrieves the current name for the specified socket descriptor in *name*. It is used on a bound and/or connected socket specified by the *s* parameter. The local association is returned. This call is especially useful when a **connect()** call has been made without doing a **bind()** first; this call provides the only means by which you can determine the local association which has been set by the system.

On return, the *namelen* argument contains the actual size of the name returned in bytes.

If a socket was bound to an unspecified address (e.g., ADDR_ANY), indicating that any of the host's addresses within the specified address family should be used for the socket, **getsockname()** will not necessarily return information about the host address, unless the socket has been connected with **connect()** or **accept().** A Winsock application must not assume that the address will be specified unless the socket is connected. This is because for a multi-homed host the address that will be used for the socket is unknown unless the socket is connected.

Return Value If no error occurs, **getsockname()** 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 *namelen* argument is not large enough, or the *name* or *namelen* argument is not part of the user address space.
WSAEINPROGRESS	A blocking Winsock call is in progress, or the service provider is still processing a callback function (see section **B.3.6.6**).
WSAENOTSOCK	The descriptor is not a socket.
WSAEINVAL	The socket has not been bound to an address with **bind().**

See Also **bind**(), **socket**(), **getpeername**().

3.14 getsockopt()

Description Retrieve a socket option.

#include <winsock.h>

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

s	A descriptor identifying a socket.
level	The level at which the option is defined; the only supported *levels* are SOL_SOCKET and SOL_PROVIDER. (SOL_PROVIDER is defined to be an alias for IPPROTO_TCP for the sake of compatibility with Windows Sockets specification 1.1.)
optname	The socket option for which the value is to be retrieved.
optval	A pointer to the buffer in which the value for the requested option is to be returned.
optlen	A pointer to the size of the *optval* buffer.

Remarks **getsockopt()** retrieves the current value for a socket option associated with a socket of any type, in any state, and stores the result in *optval*. Options may exist at multiple protocol levels, but they are always present at the uppermost "socket" level. Options affect socket operations, such as the routing of packets, out-of-band data transfer, etc.

The value associated with the selected option is returned in the buffer *optval*. The integer pointed to by *optlen* should originally contain the size of this buffer; on return, it will be set to the size of the value returned. For SO_LINGER, this will be the size of a struct linger; for most other options it will be the size of an integer.

The application is responsible for allocating any memory space pointed to directly or indirectly by any of the parameters it specified.

If the option was never set with **setsockopt(),** then **getsockopt()** returns the default value for the option.

The following options are supported for **getsockopt()**. The Type identifies the type of data addressed by *optval*.

***level* = SOL_SOCKET**		
Value	Type	Meaning
SO_ACCEPTCONN	BOOL	Socket is **listen()**ing.
SO_BROADCAST	BOOL	Socket is configured for the transmission of broadcast messages.
SO_DEBUG	BOOL	Debugging is enabled.
SO_DONTLINGER	BOOL	If true, the SO_LINGER option is disabled.
SO_DONTROUTE	BOOL	Routing is disabled.
SO_ERROR	int	Retrieve error status and clear.
SO_QOS	QOS	The flow spec of this socket.

SO_GROUP_QOS	QOS	The flow spec of the socket group to which this socket belongs.
SO_GROUP_ID	GROUP	The identifier of the group to which this socket belongs.
SO_GROUP_PRIORITY	int	The relative priority for sockets that are part of a socket group.
SO_KEEPALIVE	BOOL	Keepalives are being sent.
SO_LINGER	struct linger	Returns the current linger options.
SO_MAX_MSG_SIZE	unsigned int	Maximum size of a message for message-oriented socket types (e.g., SO_DGRAM). Has no meaning for stream-oriented sockets.
SO_OOBINLINE	BOOL	Out-of-band data is being received in the normal data stream.
SO_PROTOCOL_INFO	PROTOCOL _INFO	Description of protocol info for protocol that is bound to this socket.
SO_RCVBUF	int	Buffer size for receives
SO_REUSEADDR	BOOL	The socket may be bound to an address which is already in use.
SO_SNDBUF	int	Buffer size for sends
SO_TYPE	int	The type of the socket (e.g. SOCK_STREAM).

***level* = SOL_PROVIDER (also aliased to IPPROTO_TCP)**		
PVD_CONFIG	Service Provider Dependent	An "opaque" data structure object from the service provider associated with socket *s*. This object stores the current configuration information of the service provider. The exact format of this data structure is service provider specific.
TCP_NODELAY	BOOL	Disables the Nagle algorithm for send coalescing.

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

Value	Type	Meaning
SO_RCVLOWAT	int	Receive low water mark
SO_RCVTIMEO	int	Receive timeout
SO_SNDLOWAT	int	Send low water mark
SO_SNDTIMEO	int	Send timeout
IP_OPTIONS		Get options in IP header.
TCP_MAXSEG	int	Get TCP maximum segment size.

Calling **getsockopt()** with an unsupported option will result in an error code of WSAENOPROTOOPT being returned from **WSAGetLastError()**.

SO_DEBUG

Winsock service providers are encouraged (but not required) to supply output debug information if the SO_DEBUG option is set by an application. The mechanism for

generating the debug information and the form it takes are beyond the scope of this specification.

SO_QOS

This is a get-only socket option which indicates the QOS of the socket. The default QOS defined in Sec. 2.8.3 will be returned before the application QOS for this socket. The WSAENOPROTOOPT error code is indicated for service providers which do not support the QOS option. See **WSAConnect()** about how to set the flow spec.

SO_GROUP_QOS

This is a get-only socket option which indicates the QOS of the group this socket belongs to. The default QOS defined in Sec. 2.8.3 will be returned before the application sets the QOS for this socket group. The WSAENOPROTOOPT error code is indicated for service providers which do not support the group QOS option. If this socket does not belong to an appropriate socket group, the *ForwardFP* and *BackwardFP* fields of the returned QOS struct are set to NULL. See **WSAConnect()** about how to set the flow spec.

SO_GROUP_ID

This is a get-only socket option which indicates the identifier of the group this socket belongs to. If this socket is not a group socket, the value is NULL.

SO_GROUP_PRIORITY

Group priority indicates the relative priority of the specified socket relative to other sockets within the socket group. Values are non-negative integers, with zero corresponding to the highest priority. Priority values represent a hint to the underlying service provider about how potentially scarce resources should be allocated. For example, whenever two or more sockets are both ready to transmit data, the highest priority socket (lowest value for SO_GROUP_PRIORITY) should be serviced first, with the remainder serviced in turn according to their relative priorities.

The WSAENOPROTOOPT error code is indicated for non group sockets or for service providers which do not support group sockets.

SO_KEEPALIVE

An application may request that a TCP/IP service provider enable the use of "keep-alive" packets on TCP-connections by turning on the SO_KEEPALIVE socket option. A Winsock provider need not support the use of keep-alives: if it does, the precise semantics are implementation-specific but should conform to section 4.2.3.6 of RFC 1122: *Requirements for Internet Hosts -- Communication Layers*. If a connection is dropped as the result of "keep-alives" the error code WSAENETRESET is returned to any calls in progress on the socket, and any subsequent calls will fail with WSAENOTCONN.

SO_LINGER

SO_LINGER controls the action taken when unsent data is queued on a socket and a **closesocket()** is performed. See **closesocket()** for a description of the way in which the SO_LINGER settings affect the semantics of **closesocket()**. The application sets the desired behavior by creating a *struct linger* (pointed to by the *optval* argument) with the following elements:

```
struct linger {
      int   l_onoff;
      int   l_linger;
```

```
}
```

SO_MAX_MSG_SIZE

This is a get-only socket option which indicates the maximum size of a message for message-oriented socket types (e.g., SO_DGRAM) as implemented by a particular service provider. It has no meaning for byte stream oriented sockets

SO_REUSEADDR

By default, a socket may not be bound (see **bind()**) to a local address which is already in use. On occasions, however, it may be desirable to "re-use" an address in this way. Since every connection is uniquely identified by the combination of local and remote addresses, there is no problem with having two sockets bound to the same local address as long as the remote addresses are different. To inform the Winsock provider that a **bind()** on a socket should not be disallowed because the desired address is already in use by another socket, the application should set the SO_REUSEADDR socket option for the socket before issuing the **bind()**. Note that the option is interpreted only at the time of the **bind()**: it is therefore unnecessary (but harmless) to set the option on a socket which is not to be bound to an existing address, and setting or resetting the option after the **bind()** has no effect on this or any other socket.

PVD_CONFIG

This option retrieves an "opaque" data structure object from the service provider associated with socket *s*. This object stores the current configuration information of the service provider. The exact format of this data structure is service provider specific.

TCP_NODELAY

The TCP_NODELAY option disables the Nagle algorithm. The Nagle algorithm is used to reduce the number of small packets sent by a host by buffering unacknowledged send data until a full-size packet can be sent. However, for some applications this algorithm can impede performance, and TCP_NODELAY may be used to turn it off. Application writers should not set TCP_NODELAY unless the impact of doing so is well-understood and desired, since setting TCP_NODELAY can have a significant negative impact of network performance.

Return Value

If no error occurs, **getsockopt()** 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 *optlen* argument was invalid.
WSAEINPROGRESS	A blocking Winsock call is in progress, or the service provider is still processing a callback function (see section **B.3.6.6**).
WSAEINVAL	No value available for *optname* at the moment.
WSAENOPROTOOPT	The option is unknown or unsupported by the indicated protocol family.

WSAENOTSOCK	The descriptor is not a socket.

See Also **setsockopt**(), **socket(), WSAAsyncSelect(), WSAConnect**().

3.15 htonl()

Description Convert a **u_long** from host to network byte order.

#include <winsock.h>

u_long WSAAPI htonl (u_long *hostlong* **);**

hostlong A 32-bit number in host byte order.

Remarks This routine takes a 32-bit number in host byte order and returns a 32-bit number in network byte order.

Return Value **htonl()** returns the value in network byte order.

See Also **htons**(), **ntohl**(), **ntohs**().

3.16 htons()

Description Convert a **u_short** from host to network byte order.

#include <winsock.h>

u_short WSAAPI htons (u_short *hostshort* **);**

hostshort A 16-bit number in host byte order.

Remarks This routine takes a 16-bit number in host byte order and returns a 16-bit number in network byte order.

Return Value **htons()** returns the value in network byte order.

See Also **htonl()**, **ntohl()**, **ntohs()**.

3.17 inet_addr()

Description Convert a string containing a dotted address into an **in_addr**.

#include <winsock.h>

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

cp	A character string representing a number expressed in the Internet standard "." notation.

Remarks This function interprets the character string specified by the *cp* parameter. This string represents a numeric Internet address expressed in the Internet standard "." notation. The value returned is a number suitable for use as an Internet address. All Internet addresses are returned in network order (bytes ordered from left to right).

Internet Addresses

Values specified using the "." notation take one of the following forms:

a.b.c.d a.b.c a.b a

When four parts are specified, each is interpreted as a byte of data and assigned, from left to right, to the four bytes of an Internet address. Note that when an Internet address is viewed as a 32-bit integer quantity on the Intel architecture, the bytes referred to above appear as "d.c.b.a". That is, the bytes on an Intel processor are ordered from right to left.

Note: The following notations are only used by Berkeley, and nowhere else on the Internet. In the interests of compatibility with their software, they are supported as specified.

When a three part address is specified, the last part is interpreted as a 16-bit quantity and placed in the right most two bytes of the network address. This makes the three part address format convenient for specifying Class B network addresses as "128.net.host".

When a two part address is specified, the last part is interpreted as a 24-bit quantity and placed in the right most three bytes of the network address. This makes the two part address format convenient for specifying Class A network addresses as "net.host".

When only one part is given, the value is stored directly in the network address without any byte rearrangement.

Return Value If no error occurs, **inet_addr()** returns an unsigned long containing a suitable binary representation of the Internet address given. If the passed-in string does not contain a legitimate Internet address, for example if a portion of an "a.b.c.d" address exceeds 255, **inet_addr()** returns the value INADDR_NONE.

See Also **inet_ntoa()**

3.18 inet_ntoa()

Description Convert a network address into a string in dotted format.

#include <winsock.h>

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

in A structure which represents an Internet host address.

Remarks This function takes an Internet address structure specified by the *in* parameter. It returns an ASCII string representing the address in "." notation as "a.b.c.d". Note that the string returned by **inet_ntoa()** resides in memory which is allocated by Winsock . The application should not make any assumptions about the way in which the memory is allocated. The data is guaranteed to be valid until the next Winsock API call within the same thread, but no longer.

Return Value If no error occurs, **inet_ntoa()** returns a char pointer to a static buffer containing the text address in standard "." notation. Otherwise, it returns NULL. The data should be copied before another Winsock call is made.

See Also **inet_addr()**.

3.19 ioctlsocket()

Description Control the mode of a socket.

#include <winsock.h>

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

s	A descriptor identifying a socket.
cmd	The command to perform on the socket *s*.
argp	A pointer to a parameter for *cmd*.

Remarks This routine may be used on any socket in any state. It is used to get or retrieve operating parameters associated with the socket, independent of the protocol and communications subsystem. The following commands are supported:

Command	Semantics
FIONBIO	Enable or disable non-blocking mode on socket *s*. *argp* points at an **unsigned long**, which is non-zero if non-blocking mode is to be enabled and zero if it is to be disabled. When a socket is created, it operates in blocking mode (i.e. non-blocking mode is disabled). This is consistent with BSD sockets. The **WSAAsyncSelect()** or **WSAEventSelect()** routine automatically sets a socket to nonblocking mode. If **WSAAsyncSelect()** or **WSAEventSelect()** has been issued on a socket, then any attempt to use **ioctlsocket()** to set the socket back to blocking mode will fail with WSAEINVAL. To set the socket back to blocking mode, an application must first disable **WSAAsyncSelect()** by calling **WSAAsyncSelect()** with the *lEvent* parameter equal to 0, or disable **WSAEventSelect()** by calling **WSAEventSelect()** with the *lNetworkEvents* parameter equal to 0.
FIONREAD	Determine the amount of data which can be read atomically from socket *s*. *argp* points at an **unsigned long** in which **ioctlsocket()** stores the result. If *s* is stream-oriented (e.g., type SOCK_STREAM), FIONREAD returns the total amount of data which may be read in a single **recv()**; this is normally the same as the total amount of data queued on the socket. If *s* is message-oriented (e.g., type SOCK_DGRAM), FIONREAD returns the size of the first datagram (message) queued on the socket.
SIOCATMARK	Determine whether or not all out-of-band data has been read. This applies only to a socket of stream style (e.g., type SOCK_STREAM) which has been configured for in-line reception of any out-of-band data (SO_OOBINLINE). If no out-of-band data is waiting to be read, the operation returns TRUE. Otherwise it returns FALSE, and the next **recv()** or **recvfrom()** performed on the socket will retrieve some or all of the data preceding the "mark"; the application should use the

SIOCATMARK operation to determine whether any remains. If there is any normal data preceding the "urgent" (out of band) data, it will be received in order. (Note that a **recv()** or **recvfrom()** will never mix out-of-band and normal data in the same call.) *argp* points at a **BOOL** in which **ioctlsocket()** stores the result.

Compatibility This function is a subset of **ioctl()** as used in Berkeley sockets. In particular, there is no command which is equivalent to FIOASYNC, while SIOCATMARK is the only socket-level command which is supported.

Return Value Upon successful completion, the **ioctlsocket()** 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.
WSAEINVAL	*cmd* is not a valid command, or *argp* is not an acceptable parameter for *cmd*, or the command is not applicable to the type of socket supplied
WSAEINPROGRESS	A blocking Winsock call is in progress, or the service provider is still processing a callback function (see section **B.3.6.6**).
WSAENOTSOCK	The descriptor *s* is not a socket.

See Also **socket()**, **setsockopt()**, **getsockopt()**, **WSAAsyncSelect(), WSAEventSelect().**

3.20 listen()

Description Establish a socket to listen for incoming connection.

#include <winsock.h>

int WSAAPI listen (SOCKET *s***, int** *backlog* **);**

s	A descriptor identifying a bound, unconnected socket.
backlog	The maximum length to which the queue of pending connections may grow. If this value is SOMAXCONN, then the underlying service provider responsible for socket *s* will set the backlog to a maximum "reasonable" value.

Remarks To accept connections, a socket is first created with **socket**(), a backlog for incoming connections is specified with **listen**(), and then the connections are accepted with **accept**(). **listen**() applies only to sockets that are connection-oriented , e.g., those of type SOCK_STREAM. The socket *s* is put into "passive" mode where incoming connections are acknowledged and queued pending acceptance by the process.

This function is typically used by servers that could have more than one connection request at a time: if a connection request arrives with the queue full, the client will receive an error with an indication of WSAECONNREFUSED.

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

Return Value If no error occurs, **listen()** 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.
WSAEADDRINUSE	An attempt has been made to **listen()** on an address in use.
WSAEINPROGRESS	A blocking Winsock call is in progress, or the service provider is still processing a callback function (see section **B.3.6.6**).
WSAEINVAL	The socket has not been bound with **bind()**.
WSAEISCONN	The socket is already connected.
WSAEMFILE	No more socket descriptors are available.
WSAENOBUFS	No buffer space is available.
WSAENOTSOCK	The descriptor is not a socket.

WSAEOPNOTSUPP	The referenced socket is not of a type that supports the **listen()** operation.

See Also **accept()**, **connect()**, **socket()**.

3.21 ntohl()

Description Convert a **u_long** from network to host byte order.

#include <winsock.h>

u_long WSAAPI ntohl (u_long *netlong* **);**

netlong A 32-bit number in network byte order.

Remarks This routine takes a 32-bit number in network byte order and returns a 32-bit number in host byte order.

Return Value **ntohl()** returns the value in host byte order.

See Also **htonl**(), **htons**(), **ntohs**().

3.22 ntohs()

Description Convert a **u_short** from network to host byte order.

#include <winsock.h>

u_short WSAAPI ntohs (u_short *netshort* **);**

netshort A 16-bit number in network byte order.

Remarks This routine takes a 16-bit number in network byte order and returns a 16-bit number in host byte order.

Return Value **ntohs()** returns the value in host byte order.

See Also **htonl()**, **htons()**, **ntohl()**.

3.23 recv()

Description Receive data from a socket.

#include <winsock.h>

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

s	A descriptor identifying a connected socket.
buf	A buffer for the incoming data.
len	The length of *buf*.
flags	Specifies the way in which the call is made.

Remarks This function is used on connected connectionless or connection-oriented sockets specified by the *s* parameter and is used to read incoming data.

For sockets of stream style (e.g., type SOCK_STREAM), as much information as is currently available up to the size of the buffer supplied is returned. If the socket has been configured for in-line reception of out-of-band data (socket option SO_OOBINLINE) and out-of-band data is unread, only out-of-band data will be returned. The application may use the **ioctlsocket()** SIOCATMARK to determine whether any more out-of-band data remains to be read.

For message-oriented sockets (e.g., type SOCK_DGRAM), data is extracted from the first enqueued datagram (message), up to the size of the buffer supplied. If the datagram is larger than the buffer supplied, the buffer is filled with the first part of the datagram, the excess data is lost, and **recv()** returns the error WSAEMSGSIZE.

If no incoming data is available at the socket, the **recv()** call waits for data to arrive unless the socket is non-blocking. In this case a value of SOCKET_ERROR is returned with the error code set to WSAEWOULDBLOCK. The **select**(), **WSAAsyncSelect**(), or **WSAEventSelect**() calls may be used to determine when more data arrives.

If the socket is connection-oriented and the remote side has shut down the connection gracefully, a **recv()** will complete immediately with 0 bytes received. If the connection has been reset, a **recv()** will fail with the error WSAECONNRESET.

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_PEEK	Peek at the incoming data. The data is copied into the buffer but is not removed from the input queue.
MSG_OOB	Process out-of-band data (See section B.3.2.3 for a discussion of this topic.)

Return Value If no error occurs, **recv()** returns the number of bytes received. If the connection has been gracefully closed, the return value is 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 *buf* argument is not in a valid part of the user address space.
WSAENOTCONN	The socket is not connected.
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 **B.3.6.6**).
WSAENETRESET	The connection must be reset because the Winsock implementation 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 **recv()** on a socket after **shutdown()** has been invoked with *how* set to SD_RECEIVE or SD_BOTH.
WSAEWOULDBLOCK	The socket is marked as non-blocking and the receive operation would block.
WSAEMSGSIZE	The message was too large to fit into the specified buffer and was truncated.
WSAEINVAL	The socket has not been bound with **bind().**
WSAECONNABORTED	The virtual circuit was aborted due to timeout or other failure. . The application should close the socket as it is no longer useable.
WSAECONNRESET	The virtual circuit was reset by the remote side executing a “hard” or “abortive” close. The application should close the socket as it is no longer useable.

See Also **recvfrom**(), **send**(), **select**(), **socket**(), **WSAAsyncSelect**().

3.24 recvfrom()

Description Receive a datagram and store the source address.

#include <winsock.h>

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

s	A descriptor identifying a bound socket.
buf	A buffer for the incoming data.
len	The length of *buf*.
flags	Specifies the way in which the call is made.
from	An optional pointer to a buffer which will hold the source address upon return.
fromlen	An optional pointer to the size of the *from* buffer.

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

For stream-oriented sockets such as those of type SOCK_STREAM, as much information as is currently available up to the size of the buffer supplied is returned. If the socket has been configured for in-line reception of out-of-band data (socket option SO_OOBINLINE) and out-of-band data is unread, only out-of-band data will be returned. The application may use the **ioctlsocket**() SIOCATMARK to determine whether any more out-of-band data remains to be read. The from and fromlen parameters are ignored for connection-oriented sockets.

For message-oriented sockets, data is extracted from the first enqueued message, up to the size of the buffer supplied. If the message is larger than the buffer supplied, the buffer is filled with the first part of the message, the excess data is lost, and **recvfrom()** returns the error code WSAEMSGSIZE.

If *from* is non-zero, and the socket is not connection-oriented (e.g., type SOCK_DGRAM), the network address of the peer which sent the data is copied to the corresponding struct sockaddr. The value pointed to by *fromlen* is initialized to the size of this structure, and is modified on return to indicate the actual size of the address stored there.

If no incoming data is available at the socket, the **recvfrom()** call waits for data to arrive unless the socket is non-blocking. In this case a value of SOCKET_ERROR is returned with the error code set to WSAEWOULDBLOCK. The **select**(), **WSAAsyncSelect**(), or **WSAEventSelect**() may be used to determine when more data arrives.

If the socket is connection-oriented and the remote side has shut down the connection gracefully, a **recvfrom()** will complete immediately with 0 bytes received. If the connection has been reset **recvfrom()** will fail with the error WSAECONNRESET.

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_PEEK	Peek at the incoming data. The data is copied into the buffer but is not removed from the input queue.
MSG_OOB	Process out-of-band data (See section B.3.2.3 for a discussion of this topic.)

Return Value

If no error occurs, **recvfrom()** returns the number of bytes received. If the connection has been gracefully closed, the return value is 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 *buf* or *from* parameters are not part of the user address space, or the *fromlen* argument is too small to accommodate the peer address.
WSAEINTR	The (blocking) call was canceled via **WSACancelBlockingCall().**
WSAEINPROGRESS	A blocking Winsock call is in progress, or the service provider is still processing a callback function (see section **B.3.6.6**).
WSAEINVAL	The socket has not been bound with **bind()**.
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 **recvfrom()** on a socket after **shutdown()** has been invoked with *how* set to SD_RECEIVE or SD_BOTH.
WSAEWOULDBLOCK	The socket is marked as non-blocking and the **recvfrom()** operation would block.
WSAEMSGSIZE	The message was too large to fit into the specified buffer and was truncated.
WSAECONNABORTED	The virtual circuit was aborted due to timeout or other failure. The application should close the socket as it is no longer useable.
WSAECONNRESET	The virtual circuit was reset by the remote side executing a “hard” or “abortive” close. The application should close the socket as it is no longer useable.

See Also **recv**(), **send**(), **socket**(), **WSAAsyncSelect(), WSAEventSelect().**

3.25 select()

Description Determine the status of one or more sockets, waiting if necessary.

#include <winsock.h>

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

nfds	This argument is ignored and included only for the sake of compatibility.
readfds	An optional pointer to a set of sockets to be checked for readability.
writefds	An optional pointer to a set of sockets to be checked for writability
exceptfds	An optional pointer to a set of sockets to be checked for errors.
timeout	The maximum time for **select()** to wait, or NULL for blocking operation.

Remarks This function is used to determine the status of one or more sockets. For each socket, the caller may request information on read, write or error status. The set of sockets for which a given status is requested is indicated by an fd_set structure. The sockets contained within the fd_set structures must be associated with a single service provider. Upon return, the structure is updated to reflect the subset of these sockets which meet the specified condition, and **select()** returns the number of sockets meeting the conditions. A set of macros is provided for manipulating an fd_set. These macros are compatible with those used in the Berkeley software, but the underlying representation is completely different.

The parameter *readfds* identifies those sockets which are to be checked for readability. If the socket is currently **listen()**ing, it will be marked as readable if an incoming connection request has been received, so that an **accept()** is guaranteed to complete without blocking. For other sockets, readability means that queued data is available for reading or, for connection-oriented sockets, that the virtual circuit corresponding to the socket has been closed, so that a **recv()** or **recvfrom()** is guaranteed to complete without blocking. If the virtual circuit was closed gracefully, then a **recv()** will return immediately with 0 bytes read; if the virtual circuit was reset, then a **recv()** will complete immediately with the error code WSAECONNRESET. The presence of out-of-band data will be checked if the socket option SO_OOBINLINE has been enabled (see **setsockopt()**).

The parameter *writefds* identifies those sockets which are to be checked for writability. If a socket is **connect()**ing (non-blocking), writability means that the connection establishment successfully completed. If the socket is not in the process of **connect()**ing, writability means that a **send()** or **sendto()** will complete without blocking. [It is not specified how long this guarantee can be assumed to be valid, particularly in a multithreaded environment.]

The parameter *exceptfds* identifies those sockets which are to be checked for the presence of out-of-band data or any exceptional error conditions. Note that out-of-band data will

only be reported in this way if the option SO_OOBINLINE is FALSE. For a connection-oriented socket, the breaking of the connection by the peer or due to KEEPALIVE failure will be indicated as an exception. This specification does not define which other errors will be included. If a socket is **connect**()ing (non-blocking), failure of the connect attempt is indicated in *exceptfds.*

Any of *readfds*, *writefds*, or *exceptfds* may be given as NULL if no descriptors are of interest.

Four macros are defined in the header file **winsock.h** for manipulating the descriptor sets. The variable FD_SETSIZE determines the maximum number of descriptors in a set. (The default value of FD_SETSIZE is 64, which may be modified by #defining FD_SETSIZE to another value before #including **winsock.h**.) Internally, an fd_set is represented as an array of SOCKETs; the last valid entry is followed by an element set to INVALID_SOCKET. The macros are:

FD_CLR(*s***, ****set***)**	Removes the descriptor *s* from *set*.
FD_ISSET(*s***, ****set***)**	Nonzero if *s* is a member of the *set*, zero otherwise.
FD_SET(*s***, ****set***)**	Adds descriptor *s* to *set*.
FD_ZERO(**set***)**	Initializes the *set* to the NULL set.

The parameter *timeout* controls how long the **select()** may take to complete. If *timeout* is a null pointer, **select()** will block indefinitely until at least one descriptor meets the specified criteria. Otherwise, *timeout* points to a struct timeval which specifies the maximum time that **select()** should wait before returning. If the timeval is initialized to {0, 0}, **select()** will return immediately; this is used to "poll" the state of the selected sockets. If this is the case, then the **select()** call is considered nonblocking and the standard assumptions for nonblocking calls apply. For example, the blocking hook will not be called, and Winsock will not yield.

Return Value

select() returns the total number of descriptors which are ready and contained in the fd_set structures, 0 if the time limit expired, or SOCKET_ERROR if an error occurred. If the return value is SOCKET_ERROR, **WSAGetLastError**() may be used to retrieve a specific error code.

Comments

select() has no effect on the persistance of socket events registered with **WSAAsyncSelect**() or **WSAEventSelect**().

Error Codes

WSANOTINITIALISED	A successful **WSAStartup**() must occur before using this API.
WSAEFAULT	The Winsock implementation was unable to allocated needed resources for its internal operations, or the *readfds*, *writefds*, or *exceptfds* parameters are not part of the user address space.
WSAENETDOWN	The network subsystem has failed.
WSAEINVAL	The *timeout* value is not valid.

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 **B.3.6.6**).
WSAENOTSOCK	One of the descriptor sets contains an entry which is not a socket.

See Also **accept()**, **connect()**, **recv()**, **recvfrom()**, **send(), WSAAsyncSelect(), WSAEventSelect()**

3.26 send()

Description Send data on a connected socket.

#include <winsock.h>

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

s	A descriptor identifying a connected socket.
buf	A buffer containing the data to be transmitted.
len	The length of the data in *buf*.
flags	Specifies the way in which the call is made.

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

Note that the successful completion of a **send()** does not indicate that the data was successfully delivered.

If no buffer space is available within the transport system to hold the data to be transmitted, **send**() will block unless the socket has been placed in a non-blocking I/O mode. On non-blocking stream-oriented sockets, the number of bytes written may be between 1 and the requested length, depending on buffer availability on both the local and foreign hosts. The **select**(), **WSAAsyncSelect**() or **WSAEventSelect**() call may be used to determine when it is possible to send more data.

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 B.3.4 .
MSG_OOB	Send out-of-band data (stream style socket such as SOCK_STREAM only; see also section B.3.2.3)

Return Value If no error occurs, **send()** returns the total number of characters sent. (Note that this may be less than the number indicated by *len*.) 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.
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 **B.3.6.6**).
WSAEFAULT	The *buf* 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 **send()** on a socket after **shutdown()** has been invoked with how set to SD_SEND or SD_BOTH.
WSAEWOULDBLOCK	The socket is marked as non-blocking and the requested operation would block.
WSAEMSGSIZE	The socket is message-oriented, and the message is larger than the maximum supported by the underlying transport.
WSAEINVAL	The socket has not been bound with **bind().**
WSAECONNABORTED	The virtual circuit was aborted due to timeout or other failure. . The application should close the socket as it is no longer useable.

WSAECONNRESET	The virtual circuit was reset by the remote side executing a “hard” or “abortive” close. The application should close the socket as it is no longer useable.

See Also **recv()**, **recvfrom()**, **select()**, **socket()**, **sendto()**, **WSAAsyncSelect()**, **WSAEventSelect()**.

3.27 sendto()

Description Send data to a specific destination.

#include <winsock.h>

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

s	A descriptor identifying a socket.
buf	A buffer containing the data to be transmitted.
len	The length of the data in *buf*.
flags	Specifies the way in which the call is made.
to	An optional pointer to the address of the target socket.
tolen	The size of the address in *to*.

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

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

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

For sockets using IP:
To send a broadcast (on a SOCK_DGRAM only), the address in the *to* parameter should be constructed using the special IP address INADDR_BROADCAST (defined in **winsock.h**) together with the intended port number. It is generally inadvisable for a broadcast datagram to exceed the size at which fragmentation may occur, which implies that the data portion of the datagram (excluding headers) should not exceed 512 bytes.

If no buffer space is available within the transport system to hold the data to be transmitted, **sendto**() will block unless the socket has been placed in a non-blocking I/O mode. On non-blocking stream-oriented sockets, the number of bytes written may be between 1 and the requested length, depending on buffer availability on both the local and foreign hosts. The **select**(), **WSAAsyncSelect**() or **WSAEventSelect**() call may be used to determine when it is possible to send more data.

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 B.3.4 .
MSG_OOB	Send out-of-band data (stream style socket such as SOCK_STREAM only; see also section B.3.2.3)

Return Value If no error occurs, **sendto()** returns the total number of characters sent. (Note that this may be less than the number indicated by *len*.) 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.
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 **B.3.6.6**).
WSAEFAULT	The *buf* or *to* parameters are not part of the user address space, or the *tolen* 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 **sendto()** on a socket after **shutdown()** has been invoked with how set to SD_SEND or SD_BOTH.

WSAEWOULDBLOCK	The socket is marked as non-blocking and the requested operation would block.
WSAEMSGSIZE	The socket is message-oriented, and the message is larger than the maximum supported by the underlying transport.
WSAECONNABORTED	The virtual circuit was aborted due to timeout or other failure. . The application should close the socket as it is no longer useable.
WSAECONNRESET	The virtual circuit was reset by the remote side executing a “hard” or “abortive” close. The application should close the socket as it is no longer useable.
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 **recv()**, **recvfrom()**, **select()**, **socket()**, **send()**, **WSAAsyncSelect(), WSAEventSelect()**.

3.28 setsockopt()

Description Set a socket option.

#include <winsock.h>

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

s	A descriptor identifying a socket.
level	The level at which the option is defined; the only supported *level*s are SOL_SOCKET and SOL_PROVIDER. (SOL_PROVIDER is defined to be an alias for IPPROTO_TCP for the sake of compatibility with Windows Sockets specification 1.1.)
optname	The socket option for which the value is to be set.
optval	A pointer to the buffer in which the value for the requested option is supplied.
optlen	The size of the *optval* buffer.

Remarks **setsockopt()** sets the current value for a socket option associated with a socket of any type, in any state. Although options may exist at multiple protocol levels, they are always present at the uppermost "socket" level. Options affect socket operations, such as whether expedited data is received in the normal data stream, whether broadcast messages may be sent on the socket, etc.

There are two types of socket options: Boolean options that enable or disable a feature or behavior, and options which require an integer value or structure. To enable a Boolean option, *optval* points to a nonzero integer. To disable the option *optval* points to an integer equal to zero. *optlen* should be equal to sizeof(int) for Boolean options. For other options, *optval* points to the an integer or structure that contains the desired value for the option, and *optlen* is the length of the integer or structure.

The following options are supported for **setsockopt()**. The Type identifies the type of data addressed by *optval*.

***level* = SOL_SOCKET**		
Value	Type	Meaning
SO_BROADCAST	BOOL	Allow transmission of broadcast messages on the socket.
SO_DEBUG	BOOL	Record debugging information.
SO_DONTLINGER	BOOL	Don't block close waiting for unsent data to be sent. Setting this option is equivalent to setting SO_LINGER with *l_onoff* set to zero.
SO_DONTROUTE	BOOL	Don't route: send directly to interface.
SO_GROUP_PRIORITY	int	Specify the relative priority to be established for sockets that are part of a socket group.
SO_KEEPALIVE	BOOL	Send keepalives
SO_LINGER	struct linger	Linger on close if unsent data is present

SO_OOBINLINE	BOOL	Receive out-of-band data in the normal data stream.
SO_RCVBUF	int	Specify buffer size for receives
SO_REUSEADDR	BOOL	Allow the socket to be bound to an address which is already in use. (See **bind()**.)
SO_SNDBUF	int	Specify buffer size for sends.

***level* = SOL_PROVIDER (aliased to IPPROTO_TCP)**		
PVD_CONFIG	Service Provider Dependent	This object stores the configuration information for the service provider associated with socket *s*. The exact format of this data structure is service provider specific.
TCP_NODELAY	BOOL	Disables the Nagle algorithm for send coalescing.

BSD options not supported for **setsockopt()** are:

Value	Type	Meaning
SO_ACCEPTCONN	BOOL	Socket is listening
SO_RCVLOWAT	int	Receive low water mark
SO_RCVTIMEO	int	Receive timeout
SO_SNDLOWAT	int	Send low water mark
SO_SNDTIMEO	int	Send timeout
SO_TYPE	int	Type of the socket
IP_OPTIONS		Set options field in IP header.

SO_DEBUG

Winsock service providers are encouraged (but not required) to supply output debug information if the SO_DEBUG option is set by an application. The mechanism for generating the debug information and the form it takes are beyond the scope of this specification.

SO_GROUP_PRIORITY

Group priority indicates the relative priority of the specified socket relative to other sockets within the socket group. Values are non-negative integers, with zero corresponding to the highest priority. Priority values represent a hint to the underlying service provider about how potentially scarce resources should be allocated. For example, whenever two or more sockets are both ready to transmit data, the highest priority socket (lowest value for SO_GROUP_PRIORITY) should be serviced first, with the remainder serviced in turn according to their relative priorities.

The WSAENOPROTOOPT error is indicated for non group sockets or for service providers which do not support group sockets.

SO_KEEPALIVE

An application may request that a TCP/IP provider enable the use of "keep-alive" packets on TCP-connections by turning on the SO_KEEPALIVE socket option. A Winsock provider need not support the use of keep-alives: if it does, the precise semantics are implementation-specific but should conform to section 4.2.3.6 of RFC 1122: *Requirements for Internet Hosts -- Communication Layers*. If a connection is dropped as the result of "keep-alives" the error code WSAENETRESET is returned to any calls in progress on the socket, and any subsequent calls will fail with WSAENOTCONN.

SO_LINGER

SO_LINGER controls the action taken when unsent data is queued on a socket and a **closesocket()** is performed. See **closesocket()** for a description of the way in which the SO_LINGER settings affect the semantics of **closesocket()**. The application sets the desired behavior by creating a *struct linger* (pointed to by the *optval* argument) with the following elements:

```
struct linger {
      int   l_onoff;
      int   l_linger;
}
```

To enable SO_LINGER, the application should set *l_onoff* to a non-zero value, set *l_linger* to 0 or the desired timeout (in seconds), and call **setsockopt()**. To enable SO_DONTLINGER (i.e. disable SO_LINGER) *l_onoff* should be set to zero and **setsockopt()** should be called.

SO_REUSEADDR

By default, a socket may not be bound (see **bind()**) to a local address which is already in use. On occasions, however, it may be desirable to "re-use" an address in this way. Since every connection is uniquely identified by the combination of local and remote addresses, there is no problem with having two sockets bound to the same local address as long as the remote addresses are different. To inform the Winsock provider that a **bind()** on a socket should not be disallowed because the desired address is already in use by another socket, the application should set the SO_REUSEADDR socket option for the socket before issuing the **bind()**. Note that the option is interpreted only at the time of the **bind()**: it is therefore unnecessary (but harmless) to set the option on a socket which is not to be bound to an existing address, and setting or resetting the option after the **bind()** has no effect on this or any other socket.

PVD_CONFIG

This object stores the configuration information for the service provider associated with socket *s*. The exact format of this data structure is service provider specific.

TCP_NODELAY

The TCP_NODELAY option is specific to TCP/IP service providers. It is used to disable the Nagle algorithm. The Nagle algorithm is used to reduce the number of small packets sent by a host by buffering unacknowledged send data until a full-size packet can be sent. However, for some applications this algorithm can impede performance, and TCP_NODELAY may be used to turn it off. Application writers should not set TCP_NODELAY unless the impact of doing so is well-understood and desired, since setting TCP_NODELAY can have a significant negative impact of network performance.

Return Value

If no error occurs, **setsockopt()** 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	*optval* is not in a valid part of the process address space.
WSAEINPROGRESS	A blocking Winsock call is in progress, or the service provider is still processing a callback function (see section **B.3.6.6**).
WSAEINVAL	*level* is not valid, or the information in *optval* is not valid.
WSAENETRESET	Connection has timed out when SO_KEEPALIVE is set.
WSAENOPROTOOPT	The option is unknown or unsupported for the specified provider.
WSAENOTCONN	Connection has been reset when SO_KEEPALIVE is set.
WSAENOTSOCK	The descriptor is not a socket.

See Also **bind**(), **getsockopt**(), **ioctlsocket**(), **socket**(), **WSAAsyncSelect**(), **WSAEventSelect**().

3.29 shutdown()

Description Disable sends and/or receives on a socket.

#include <winsock.h>

int WSAAPI shutdown (SOCKET *s*, **int** *how* **);**

s	A descriptor identifying a socket.
how	A flag that describes what types of operation will no longer be allowed.

Remarks **shutdown**() is used on all types of sockets to disable reception, transmission, or both.

If *how* is SD_RECEIVE, subsequent receives on the socket will be disallowed. This has no effect on the lower protocol layers. For TCP, the TCP window is not changed and incoming data will be accepted (but not acknowledged) until the window is exhausted. For UDP, incoming datagrams are accepted and queued. In no case will an ICMP error packet be generated.

If *how* is SD_SEND, subsequent sends are disallowed. For TCP sockets, a FIN will be sent.

Setting *how* to SD_BOTH disables both sends and receives as described above.

Note that **shutdown()** does not close the socket, and resources attached to the socket will not be freed until **closesocket()** is invoked.

Comments **shutdown()** does not block regardless of the SO_LINGER setting on the socket.

An application should not rely on being able to re-use a socket after it has been shut down. In particular, a Winsock provider is not required to support the use of **connect**() on such a socket.

Return Value If no error occurs, **shutdown()** 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.
WSAEINVAL	*how* is not valid, or is not consistent with the socket type, e.g., SD_SEND is used with a UNI_RECV socket type.
WSAEINPROGRESS	A blocking Winsock call is in progress, or the service provider is still processing a callback function (see section **B.3.6.6**).

WSAENOTCONN	The socket is not connected (connection-oriented sockets only).
WSAENOTSOCK	The descriptor is not a socket.

See Also **connect()**, **socket()**.

3.30 socket()

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

#include <winsock.h>

SOCKET WSAAPI socket (int *af***, int** *type***, int** *protocol* **);**

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.

Remarks **socket()** causes a socket descriptor and any related resources to be allocated and bound to a specific transport service provider. Winsock will utilize the first available service provider that supports the requested combination of address family, socket type and protocol parameters.

If a protocol is not specified (i.e., equal to 0), 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 following are the only two *type* specifications supported for Winsock 1.1:

Type	Explanation
SOCK_STREAM	Provides sequenced, reliable, two-way, connection-based byte streams with an out-of-band data transmission mechanism. Uses TCP for the Internet address family.
SOCK_DGRAM	Supports datagrams, which are connectionless, unreliable buffers of a fixed (typically small) maximum length. Uses UDP for the Internet address family.

In Winsock 2 many new socket types will be introduced. However, since an application can dynamically discover the attributes of each available transport protocol via the **WSAEnumProtocols**() function, the various socket types need not be called out in the API specification. Socket type definitions will appear in Winsock.h which will be periodically updated as new socket types, address families and protocols are defined.

Connection-oriented sockets such as SOCK_STREAM provide full-duplex connections, and must be in a connected state before any data 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**() and **recv**() calls. When a session has been completed, a **closesocket**() must be performed.

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

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

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

Return Value

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

Error Codes

WSANOTINITIALISED	A successful **WSAStartup()** must occur before using this API.
WSAENETDOWN	The network subsystem or the associated service provider 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 **B.3.6.6**).
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**().

3.31 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 <winsock.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 CALLBACK ConditionFunc(
        WSABUF CallerId,
        LPWSABUF lpCallerData,
        LPQOS lpCallerSQOS,
        LPQOS lpCallerGQOS,
        WSABUF CalleeId,
        LPWSABUF lpCalleeData,
        GROUP FAR * g,
        DWORD dwCallbackData );
```

LPWSABUF and **LPQOS** are defined in winsock.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           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.

lpCallerSQOS contains two blocks of memory containing the flow specs for socket *s*, one for each direction, specified by the caller. The forward or backward flow spec values will be set to NULL as appropriate for any unidirectional sockets. The first part of each memory block contains the flow parameters common to all providers, optionally followed by any provider specific portion started at *ProviderSpecificParams*. These is no provider specific portion if *SizePSP* is 0. A NULL value for *lpCallerSQOS* indicates no caller supplied QOS.

lpCallerGQOS specifies two blocks of memory containing the flow specs for the socket group the caller is to create, one for each direction. The first part of each memory block contains the flow parameters common to all providers, optionally followed by any provider specific portion started at *ProviderSpecificParams*. These is no provider specific portion if *SizePSP* is 0. A NULL value for *lpCallerGQOS* indicates no caller-supplied group QOS.

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().**

3.32 WSAAsyncGetHostByAddr()

Description Get host information corresponding to an address - asynchronous version.

#include <winsock.h>

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

hWnd	The handle of the window which should receive a message when the asynchronous request completes.
wMsg	The message to be received when the asynchronous request completes.
addr	A pointer to the network address for the host. Host addresses are stored in network byte order.
len	The length of the address.
type	The type of the address
buf	A pointer to the data area to receive the hostent data. Note that this must be larger than the size of a hostent structure. This is because the data area supplied is used by Winsock to contain not only a hostent structure but any and all of the data which is referenced by members of the hostent structure. It is recommended that you supply a buffer of MAXGETHOSTSTRUCT bytes.
buflen	The size of data area *buf* above.

Remarks This function is an asynchronous version of **gethostbyaddr()**, and is used to retrieve host name and address information corresponding to a network address. Winsock initiates the operation and returns to the caller immediately, passing back an asynchronous task handle which the application may use to identify the operation. When the operation is completed, the results (if any) are copied into the buffer provided by the caller and a message is sent to the application's window.

When the asynchronous operation is complete the application's window *hWnd* receives message *wMsg*. The *wParam* argument contains the asynchronous task handle as returned by the original function call. The high 16 bits of *lParam* contain any error code. The error code may be any error as defined in **winsock.h**. An error code of zero indicates successful completion of the asynchronous operation. On successful completion, the buffer supplied to the original function call contains a hostent structure. To access the elements of this structure, the original buffer address should be cast to a hostent structure pointer and accessed as appropriate.

Note that if the error code is WSAENOBUFS, it indicates that the size of the buffer specified by *buflen* in the original call was too small to contain all the resultant information. In this case, the low 16 bits of *lParam* contain the size of buffer required to supply ALL the requisite information. If the application decides that the partial data is inadequate, it may reissue the **WSAAsyncGetHostByAddr()** function call with a buffer

large enough to receive all the desired information (i.e. no smaller than the low 16 bits of *lParam*).

The error code and buffer length should be extracted from the *lParam* using the macros WSAGETASYNCERROR and WSAGETASYNCBUFLEN, defined in **winsock.h** as:

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

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

Return Value

The return value specifies whether or not the asynchronous operation was successfully initiated. Note that it does not imply success or failure of the operation itself.

If the operation was successfully initiated, **WSAAsyncGetHostByAddr()** returns a nonzero value of type HANDLE which is the asynchronous task handle for the request. This value can be used in two ways. It can be used to cancel the operation using **WSACancelAsyncRequest()**. It can also be used to match up asynchronous operations and completion messages, by examining the *wParam* message argument.

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

Comments

The buffer supplied to this function is used by Winsock to construct a hostent structure together with the contents of data areas referenced by members of the same hostent structure. To avoid the WSAENOBUFS error noted above, the application should provide a buffer of at least MAXGETHOSTSTRUCT bytes (as defined in **winsock.h**).

Error Codes

The following error codes may be set when an application window receives a message. As described above, they may be extracted from the *lParam* in the reply message using the WSAGETASYNCERROR macro.

WSAENETDOWN	The network subsystem has failed.
WSAENOBUFS	No/insufficient buffer space is available
WSAHOST_NOT_FOUND	Authoritative Answer Host not found.
WSATRY_AGAIN	Non-Authoritative Host not found, or SERVERFAIL.
WSANO_RECOVERY	Non recoverable errors, FORMERR, REFUSED, NOTIMP.
WSANO_DATA	Valid name, no data record of requested type.

The following errors may occur at the time of the function call, and indicate that the asynchronous operation could not be initiated.

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

WSAENETDOWN	The network subsystem has failed.
WSAEINPROGRESS	A blocking Winsock call is in progress, or the service provider is still processing a callback function (see section **B.3.6.6**).
WSAEWOULDBLOCK	The asynchronous operation cannot be scheduled at this time due to resource or other constraints within the Winsock implementation.

See Also **gethostbyaddr()**, **WSACancelAsyncRequest()**

3.33 WSAAsyncGetHostByName()

Description Get host information corresponding to a hostname - asynchronous version.

#include <winsock.h>

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

hWnd	The handle of the window which should receive a message when the asynchronous request completes.
wMsg	The message to be received when the asynchronous request completes.
name	A pointer to the name of the host.
buf	A pointer to the data area to receive the hostent data. Note that this must be larger than the size of a hostent structure. This is because the data area supplied is used by Winsock to contain not only a hostent structure but any and all of the data which is referenced by members of the hostent structure. It is recommended that you supply a buffer of MAXGETHOSTSTRUCT bytes.
buflen	The size of data area *buf* above.

Remarks This function is an asynchronous version of **gethostbyname()**, and is used to retrieve host name and address information corresponding to a hostname. Winsock initiates the operation and returns to the caller immediately, passing back an asynchronous task handle which the application may use to identify the operation. When the operation is completed, the results (if any) are copied into the buffer provided by the caller and a message is sent to the application's window.

When the asynchronous operation is complete the application's window *hWnd* receives message *wMsg*. The *wParam* argument contains the asynchronous task handle as returned by the original function call. The high 16 bits of *lParam* contain any error code. The error code may be any error as defined in **winsock.h**. An error code of zero indicates successful completion of the asynchronous operation. On successful completion, the buffer supplied to the original function call contains a hostent structure. To access the elements of this structure, the original buffer address should be cast to a hostent structure pointer and accessed as appropriate.

Note that if the error code is WSAENOBUFS, it indicates that the size of the buffer specified by *buflen* in the original call was too small to contain all the resultant information. In this case, the low 16 bits of *lParam* contain the size of buffer required to supply ALL the requisite information. If the application decides that the partial data is inadequate, it may reissue the **WSAAsyncGetHostByName()** function call with a buffer large enough to receive all the desired information (i.e. no smaller than the low 16 bits of *lParam*).

The error code and buffer length should be extracted from the *lParam* using the macros WSAGETASYNCERROR and WSAGETASYNCBUFLEN, defined in **winsock.h** as:

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

```
#define WSAGETASYNCBUFLEN(lParam)           LOWORD(lParam)
```

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

Return Value

The return value specifies whether or not the asynchronous operation was successfully initiated. Note that it does not imply success or failure of the operation itself.

If the operation was successfully initiated, **WSAAsyncGetHostByName()** returns a nonzero value of type HANDLE which is the asynchronous task handle for the request. This value can be used in two ways. It can be used to cancel the operation using **WSACancelAsyncRequest()**. It can also be used to match up asynchronous operations and completion messages, by examining the *wParam* message argument.

If the asynchronous operation could not be initiated, **WSAAsyncGetHostByName()** returns a zero value, and a specific error number may be retrieved by calling **WSAGetLastError()**.

Comments

The buffer supplied to this function is used by Winsock to construct a hostent structure together with the contents of data areas referenced by members of the same hostent structure. To avoid the WSAENOBUFS error noted above, the application should provide a buffer of at least MAXGETHOSTSTRUCT bytes (as defined in **winsock.h**).

Error Codes

The following error codes may be set when an application window receives a message. As described above, they may be extracted from the *lParam* in the reply message using the WSAGETASYNCERROR macro.

WSAENETDOWN	The network subsystem has failed.
WSAENOBUFS	No/insufficient buffer space is available
WSAHOST_NOT_FOUND	Authoritative Answer Host not found.
WSATRY_AGAIN	Non-Authoritative Host not found, or SERVERFAIL.
WSANO_RECOVERY	Non recoverable errors, FORMERR, REFUSED, NOTIMP.
WSANO_DATA	Valid name, no data record of requested type.

The following errors may occur at the time of the function call, and indicate that the asynchronous operation could not be initiated.

WSANOTINITIALISED	A successful **WSAStartup()** must occur before using this API.
WSAENETDOWN	The network subsystem has failed.
WSAEINPROGRESS	A blocking Winsock call is in progress, or the service provider is still processing a callback function (see section **B.3.6.6**).

WSAEWOULDBLOCK	The asynchronous operation cannot be scheduled at this time due to resource or other constraints within the Winsock implementation.

See Also **gethostbyname()**, **WSACancelAsyncRequest()**

3.34 WSAAsyncGetProtoByName()

Description Get protocol information corresponding to a protocol name - asynchronous version.

#include <winsock.h>

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

hWnd	The handle of the window which should receive a message when the asynchronous request completes.
wMsg	The message to be received when the asynchronous request completes.
name	A pointer to the protocol name to be resolved.
buf	A pointer to the data area to receive the protoent data. Note that this must be larger than the size of a protoent structure. This is because the data area supplied is used by Winsock to contain not only a protoent structure but any and all of the data which is referenced by members of the protoent structure. It is recommended that you supply a buffer of MAXGETHOSTSTRUCT bytes.
buflen	The size of data area *buf* above.

Remarks This function is an asynchronous version of **getprotobyname()**, and is used to retrieve the protocol name and number corresponding to a protocol name. Winsock initiates the operation and returns to the caller immediately, passing back an asynchronous task handle which the application may use to identify the operation. When the operation is completed, the results (if any) are copied into the buffer provided by the caller and a message is sent to the application's window.

When the asynchronous operation is complete the application's window *hWnd* receives message *wMsg*. The *wParam* argument contains the asynchronous task handle as returned by the original function call. The high 16 bits of *lParam* contain any error code. The error code may be any error as defined in **winsock.h**. An error code of zero indicates successful completion of the asynchronous operation. On successful completion, the buffer supplied to the original function call contains a protoent structure. To access the elements of this structure, the original buffer address should be cast to a protoent structure pointer and accessed as appropriate.

Note that if the error code is WSAENOBUFS, it indicates that the size of the buffer specified by *buflen* in the original call was too small to contain all the resultant information. In this case, the low 16 bits of *lParam* contain the size of buffer required to supply ALL the requisite information. If the application decides that the partial data is inadequate, it may reissue the **WSAAsyncGetProtoByName()** function call with a buffer large enough to receive all the desired information (i.e. no smaller than the low 16 bits of *lParam*).

The error code and buffer length should be extracted from the *lParam* using the macros WSAGETASYNCERROR and WSAGETASYNCBUFLEN, defined in **winsock.h** as:

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

```
#define WSAGETASYNCBUFLEN(lParam)           LOWORD(lParam)
```

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

Return Value

The return value specifies whether or not the asynchronous operation was successfully initiated. Note that it does not imply success or failure of the operation itself.

If the operation was successfully initiated, **WSAAsyncGetProtoByName()** returns a nonzero value of type HANDLE which is the asynchronous task handle for the request. This value can be used in two ways. It can be used to cancel the operation using **WSACancelAsyncRequest()**. It can also be used to match up asynchronous operations and completion messages, by examining the *wParam* message argument.

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

Comments

The buffer supplied to this function is used by Winsock to construct a protoent structure together with the contents of data areas referenced by members of the same protoent structure. To avoid the WSAENOBUFS error noted above, the application should provide a buffer of at least MAXGETHOSTSTRUCT bytes (as defined in **winsock.h**).

Error Codes

The following error codes may be set when an application window receives a message. As described above, they may be extracted from the *lParam* in the reply message using the WSAGETASYNCERROR macro.

WSAENETDOWN	The network subsystem has failed.
WSAENOBUFS	No/insufficient buffer space is available
WSAHOST_NOT_FOUND	Authoritative Answer Host not found.
WSATRY_AGAIN	Non-Authoritative Host not found, or SERVERFAIL.
WSANO_RECOVERY	Non recoverable errors, FORMERR, REFUSED, NOTIMP.
WSANO_DATA	Valid name, no data record of requested type.

The following errors may occur at the time of the function call, and indicate that the asynchronous operation could not be initiated.

WSANOTINITIALISED	A successful **WSAStartup()** must occur before using this API.
WSAENETDOWN	The network subsystem has failed.
WSAEINPROGRESS	A blocking Winsock call is in progress, or the service provider is still processing a callback function (see section **B.3.6.6**).

WSAEWOULDBLOCK	The asynchronous operation cannot be scheduled at this time due to resource or other constraints within the Winsock implementation.

See Also **getprotobyname()**, **WSACancelAsyncRequest()**

3.35 WSAAsyncGetProtoByNumber()

Description Get protocol information corresponding to a protocol number - asynchronous version.

#include <winsock.h>

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

hWnd	The handle of the window which should receive a message when the asynchronous request completes.
wMsg	The message to be received when the asynchronous request completes.
number	The protocol number to be resolved, in host byte order.
buf	A pointer to the data area to receive the protoent data. Note that this must be larger than the size of a protoent structure. This is because the data area supplied is used by Winsock to contain not only a protoent structure but any and all of the data which is referenced by members of the protoent structure. It is recommended that you supply a buffer of MAXGETHOSTSTRUCT bytes.
buflen	The size of data area *buf* above.

Remarks This function is an asynchronous version of **getprotobynumber()**, and is used to retrieve the protocol name and number corresponding to a protocol number. Winsock initiates the operation and returns to the caller immediately, passing back an asynchronous task handle which the application may use to identify the operation. When the operation is completed, the results (if any) are copied into the buffer provided by the caller and a message is sent to the application's window.

When the asynchronous operation is complete the application's window *hWnd* receives message *wMsg*. The *wParam* argument contains the asynchronous task handle as returned by the original function call. The high 16 bits of *lParam* contain any error code. The error code may be any error as defined in **winsock.h**. An error code of zero indicates successful completion of the asynchronous operation. On successful completion, the buffer supplied to the original function call contains a protoent structure. To access the elements of this structure, the original buffer address should be cast to a protoent structure pointer and accessed as appropriate.

Note that if the error code is WSAENOBUFS, it indicates that the size of the buffer specified by *buflen* in the original call was too small to contain all the resultant information. In this case, the low 16 bits of *lParam* contain the size of buffer required to supply ALL the requisite information. If the application decides that the partial data is inadequate, it may reissue the **WSAAsyncGetProtoByNumber()** function call with a buffer large enough to receive all the desired information (i.e. no smaller than the low 16 bits of *lParam*).

The error code and buffer length should be extracted from the *lParam* using the macros WSAGETASYNCERROR and WSAGETASYNCBUFLEN, defined in **winsock.h** as:

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

```
#define WSAGETASYNCBUFLEN(lParam)           LOWORD(lParam)
```

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

Return Value

The return value specifies whether or not the asynchronous operation was successfully initiated. Note that it does not imply success or failure of the operation itself.

If the operation was successfully initiated, **WSAAsyncGetProtoByNumber()** returns a nonzero value of type HANDLE which is the asynchronous task handle for the request. This value can be used in two ways. It can be used to cancel the operation using **WSACancelAsyncRequest()**. It can also be used to match up asynchronous operations and completion messages, by examining the *wParam* message argument.

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

Comments

The buffer supplied to this function is used by Winsock to construct a protoent structure together with the contents of data areas referenced by members of the same protoent structure. To avoid the WSAENOBUFS error noted above, the application should provide a buffer of at least MAXGETHOSTSTRUCT bytes (as defined in **winsock.h**).

Error Codes

The following error codes may be set when an application window receives a message. As described above, they may be extracted from the *lParam* in the reply message using the WSAGETASYNCERROR macro.

WSAENETDOWN	The network subsystem has failed.
WSAENOBUFS	No/insufficient buffer space is available
WSAHOST_NOT_FOUND	Authoritative Answer Host not found.
WSATRY_AGAIN	Non-Authoritative Host not found, or SERVERFAIL.
WSANO_RECOVERY	Non recoverable errors, FORMERR, REFUSED, NOTIMP.
WSANO_DATA	Valid name, no data record of requested type.

The following errors may occur at the time of the function call, and indicate that the asynchronous operation could not be initiated.

WSANOTINITIALISED	A successful **WSAStartup()** must occur before using this API.
WSAENETDOWN	The network subsystem has failed.
WSAEINPROGRESS	A blocking Winsock call is in progress, or the service provider is still processing a callback function (see section **B.3.6.6**).

WSAEWOULDBLOCK	The asynchronous operation cannot be scheduled at this time due to resource or other constraints within the Winsock implementation.

See Also **getprotobynumber()**, **WSACancelAsyncRequest()**

3.36 WSAAsyncGetServByName()

Description Get service information corresponding to a service name and port - asynchronous version.

#include <winsock.h>

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

hWnd	The handle of the window which should receive a message when the asynchronous request completes.
wMsg	The message to be received when the asynchronous request completes.
name	A pointer to a service name.
proto	A pointer to a protocol name. This may be NULL, in which case **WSAAsyncGetServByName()** will search for the first service entry for which *s_name* or one of the *s_aliases* matches the given *name*. Otherwise **WSAAsyncGetServByName()** matches both *name* and *proto*.
buf	A pointer to the data area to receive the servent data. Note that this must be larger than the size of a servent structure. This is because the data area supplied is used by Winsock to contain not only a servent structure but any and all of the data which is referenced by members of the servent structure. It is recommended that you supply a buffer of MAXGETHOSTSTRUCT bytes.
buflen	The size of data area *buf* above.

Remarks This function is an asynchronous version of **getservbyname()**, and is used to retrieve service information corresponding to a service name. Winsock initiates the operation and returns to the caller immediately, passing back an asynchronous task handle which the application may use to identify the operation. When the operation is completed, the results (if any) are copied into the buffer provided by the caller and a message is sent to the application's window.

When the asynchronous operation is complete the application's window *hWnd* receives message *wMsg*. The *wParam* argument contains the asynchronous task handle as returned by the original function call. The high 16 bits of *lParam* contain any error code. The error code may be any error as defined in **winsock.h**. An error code of zero indicates successful completion of the asynchronous operation. On successful completion, the buffer supplied to the original function call contains a servent structure. To access the elements of this structure, the original buffer address should be cast to a servent structure pointer and accessed as appropriate.

Note that if the error code is WSAENOBUFS, it indicates that the size of the buffer specified by *buflen* in the original call was too small to contain all the resultant information. In this case, the low 16 bits of *lParam* contain the size of buffer required to

supply ALL the requisite information. If the application decides that the partial data is inadequate, it may reissue the **WSAAsyncGetServByName()** function call with a buffer large enough to receive all the desired information (i.e. no smaller than the low 16 bits of *lParam*).

The error code and buffer length should be extracted from the *lParam* using the macros WSAGETASYNCERROR and WSAGETASYNCBUFLEN, defined in **winsock.h** as:

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

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

Return Value

The return value specifies whether or not the asynchronous operation was successfully initiated. Note that it does not imply success or failure of the operation itself.

If the operation was successfully initiated, **WSAAsyncGetServByName()** returns a nonzero value of type HANDLE which is the asynchronous task handle for the request. This value can be used in two ways. It can be used to cancel the operation using **WSACancelAsyncRequest()**. It can also be used to match up asynchronous operations and completion messages, by examining the *wParam* message argument.

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

Comments

The buffer supplied to this function is used by Winsock to construct a servent structure together with the contents of data areas referenced by members of the same servent structure. To avoid the WSAENOBUFS error noted above, the application should provide a buffer of at least MAXGETHOSTSTRUCT bytes (as defined in **winsock.h**).

Error Codes

The following error codes may be set when an application window receives a message. As described above, they may be extracted from the *lParam* in the reply message using the WSAGETASYNCERROR macro.

WSAENETDOWN	The network subsystem has failed.
WSAENOBUFS	No/insufficient buffer space is available
WSAHOST_NOT_FOUND	Authoritative Answer Host not found.
WSATRY_AGAIN	Non-Authoritative Host not found, or SERVERFAIL.
WSANO_RECOVERY	Non recoverable errors, FORMERR, REFUSED, NOTIMP.
WSANO_DATA	Valid name, no data record of requested type.

The following errors may occur at the time of the function call, and indicate that the asynchronous operation could not be initiated.

WSANOTINITIALISED	A successful **WSAStartup()** must occur before using this API.
WSAENETDOWN	The network subsystem has failed.
WSAEINPROGRESS	A blocking Winsock call is in progress, or the service provider is still processing a callback function (see section **B.3.6.6**).
WSAEWOULDBLOCK	The asynchronous operation cannot be scheduled at this time due to resource or other constraints within the Winsock implementation.

See Also **getservbyname()**, **WSACancelAsyncRequest()**

3.37 WSAAsyncGetServByPort()

Description Get service information corresponding to a port and protocol - asynchronous version.

#include <winsock.h>

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

hWnd	The handle of the window which should receive a message when the asynchronous request completes.
wMsg	The message to be received when the asynchronous request completes.
port	The port for the service, in network byte order.
proto	A pointer to a protocol name. This may be NULL, in which case **WSAAsyncGetServByPort()** will search for the first service entry for which *s_port* match the given *port*. Otherwise **WSAAsyncGetServByPort()** matches both *port* and *proto*.
buf	A pointer to the data area to receive the servent data. Note that this must be larger than the size of a servent structure. This is because the data area supplied is used by Winsock to contain not only a servent structure but any and all of the data which is referenced by members of the servent structure. It is recommended that you supply a buffer of MAXGETHOSTSTRUCT bytes.
buflen	The size of data area *buf* above.

Remarks This function is an asynchronous version of **getservbyport()**, and is used to retrieve service information corresponding to a port number. Winsock initiates the operation and returns to the caller immediately, passing back an asynchronous task handle which the application may use to identify the operation. When the operation is completed, the results (if any) are copied into the buffer provided by the caller and a message is sent to the application's window.

When the asynchronous operation is complete the application's window *hWnd* receives message *wMsg*. The *wParam* argument contains the asynchronous task handle as returned by the original function call. The high 16 bits of *lParam* contain any error code. The error code may be any error as defined in **winsock.h**. An error code of zero indicates successful completion of the asynchronous operation. On successful completion, the buffer supplied to the original function call contains a servent structure. To access the elements of this structure, the original buffer address should be cast to a servent structure pointer and accessed as appropriate.

Note that if the error code is WSAENOBUFS, it indicates that the size of the buffer specified by *buflen* in the original call was too small to contain all the resultant information. In this case, the low 16 bits of *lParam* contain the size of buffer required to supply ALL the requisite information. If the application decides that the partial data is inadequate, it may reissue the **WSAAsyncGetServByPort()** function call with a buffer

large enough to receive all the desired information (i.e. no smaller than the low 16 bits of *lParam*).

The error code and buffer length should be extracted from the *lParam* using the macros WSAGETASYNCERROR and WSAGETASYNCBUFLEN, defined in **winsock.h** as:

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

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

Return Value

The return value specifies whether or not the asynchronous operation was successfully initiated. Note that it does not imply success or failure of the operation itself.

If the operation was successfully initiated, **WSAAsyncGetServByPort()** returns a nonzero value of type HANDLE which is the asynchronous task handle for the request. This value can be used in two ways. It can be used to cancel the operation using **WSACancelAsyncRequest()**. It can also be used to match up asynchronous operations and completion messages, by examining the *wParam* message argument.

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

Comments

The buffer supplied to this function is used by Winsock to construct a servent structure together with the contents of data areas referenced by members of the same servent structure. To avoid the WSAENOBUFS error noted above, the application should provide a buffer of at least MAXGETHOSTSTRUCT bytes (as defined in **winsock.h**).

Error Codes

The following error codes may be set when an application window receives a message. As described above, they may be extracted from the *lParam* in the reply message using the WSAGETASYNCERROR macro.

WSAENETDOWN	The network subsystem has failed.
WSAENOBUFS	No/insufficient buffer space is available
WSAHOST_NOT_FOUND	Authoritative Answer Host not found.
WSATRY_AGAIN	Non-Authoritative Host not found, or SERVERFAIL.
WSANO_RECOVERY	Non recoverable errors, FORMERR, REFUSED, NOTIMP.
WSANO_DATA	Valid name, no data record of requested type.

The following errors may occur at the time of the function call, and indicate that the asynchronous operation could not be initiated.

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

WSAENETDOWN	The network subsystem has failed.
WSAEINPROGRESS	A blocking Winsock call is in progress, or the service provider is still processing a callback function (see section **B.3.6.6**).
WSAEWOULDBLOCK	The asynchronous operation cannot be scheduled at this time due to resource or other constraints within the Winsock implementation.

See Also **getservbyport()**, **WSACancelAsyncRequest()**

3.38 WSAAsyncSelect()

Description Request event notification for a socket.

#include < winsock.h >

int WSAAPI 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 Winsock 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 non-overlapped socket back to blocking mode.

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

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

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

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

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

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

To cancel all notification – i.e., to indicate that Winsock 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 **winsock.h**.

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

TheWinsock 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_QOS**
FD_GROUP_QOS	**getsockopt()** with option **SO_GROUP_QOS**

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()/WSAConnect()** or accepted with **accept()/WSAAccept()**, and then after a send operation fails with WSAEWOULDBLOCK and buffer space becomes available. Therefore, an application can assume that sends are possible starting from the first FD_WRITE message and lasting until a send returns WSAEWOULDBLOCK. After such a failure the application will be notified that sends are again possible with an FD_WRITE message.

The FD_OOB event is used only when a socket is configured to receive out-of-band data separately. 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 **getsockopt()** with option **SO_QOS** or **SO_GROUP_QOS** to get the current QOS for socket *s* or for the socket group *s* belongs to, respectively.

Error Codes

WSANOTINITIALISED	A successful **WSAStartup()** must occur before using this 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 **B.3.6.6**).
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(), WSAEventSelect()

3.39 WSACancelAsyncRequest()

Description Cancel an incomplete asynchronous operation.

#include <winsock.h>

int WSAAPI WSACancelAsyncRequest (HANDLE *hAsyncTaskHandle* **);**

hAsyncTaskHandle Specifies the asynchronous operation to be canceled.

Remarks The **WSACancelAsyncRequest()** function is used to cancel an asynchronous operation which was initiated by one of the **WSAAsyncGetXByY()** functions such as **WSAAsyncGetHostByName()**. The operation to be canceled is identified by the *hAsyncTaskHandle* parameter, which should be set to the asynchronous task handle as returned by the initiating function.

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

Comments An attempt to cancel an existing asynchronous **WSAAsyncGetXByY()** operation can fail with an error code of WSAEALREADY for two reasons. First, the original operation has already completed and the application has dealt with the resultant message. Second, the original operation has already completed but the resultant message is still waiting in the application window queue.

Note It is unclear whether the application can usefully distinguish between WSAEINVAL and WSAEALREADY, since in both cases the error indicates that there is no asynchronous operation in progress with the indicated handle. [Trivial exception: 0 is always an invalid asynchronous task handle.] The Winsock specification does not prescribe how a conformant Winsock provider should distinguish between the two cases. For maximum portability, a Winsock application should treat the two errors as equivalent.

Error Codes

WSANOTINITIALISED	A successful **WSAStartup()** must occur before using this API.
WSAENETDOWN	The network subsystem has failed.
WSAEINVAL	Indicates that the specified asynchronous task handle was invalid
WSAEINPROGRESS	A blocking Winsock call is in progress, or the service provider is still processing a callback function (see section **B.3.6.6**).
WSAEALREADY	The asynchronous routine being canceled has already completed.

See Also **WSAAsyncGetHostByAddr**(), **WSAAsyncGetHostByName**(), **WSAAsyncGetProtoByNumber**(), **WSAAsyncGetProtoByName**(), **WSAAsyncGetServByPort**(), **WSAAsyncGetServByName**().

3.40 WSACancelBlockingCall()

Description Cancel a blocking call which is currently in progress.

#include <winsock.h>

int WSAAPI WSACancelBlockingCall (void);

Remarks This function cancels any outstanding blocking operation for this task. It is normally used in two situations:

(1) An application is processing a message which has been received while a blocking call is in progress. In this case, **WSAIsBlocking()** will be true.

(2) A blocking call is in progress, and Winsock has called back to the application's "blocking hook" function (as established by **WSASetBlockingHook()**).

In each case, the original blocking call will terminate as soon as possible with the error WSAEINTR. (In (1), the termination will not take place until Windows message scheduling has caused control to revert to the blocking routine in Winsock. In (2), the blocking call will be terminated as soon as the blocking hook function completes.)

In the case of a blocking **connect()** operation, Winsock will terminate the blocking call as soon as possible, but it may not be possible for the socket resources to be released until the connection has completed (and then been reset) or timed out. This is likely to be noticeable only if the application immediately tries to open a new socket (if no sockets are available), or to **connect()** to the same peer.

Canceling an **accept()** or a **select()** call does not adversely impact the sockets passed to these calls. Only the particular call fails; any operation that was legal before the cancel is legal after the cancel, and the state of the socket is not affected in any way.

Canceling any operation other than **accept()** and **select()** can leave the socket in an indeterminate state. If an application cancels a blocking operation on a socket, the only operation that the application can depend on being able to perform on the socket is a call to **closesocket()**, although other operations may work on some Winsock implementations. If an application desires maximum portability, it must be careful not to depend on performing operations after a cancel. An application may reset the connection by setting the timeout on SO_LINGER to 0.

If a cancel operation compromised the integrity of a SOCK_STREAM's data stream in any way, the Winsock provider must reset the connection and fail all future operations other than **closesocket()** with WSAECONNABORTED.

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

Comments Note that it is possible that the network operation completes before the **WSACancelBlockingCall()** is processed, for example if data is received into the user buffer at interrupt time while the application is in a blocking hook. In this case, the blocking operation will return successfully as if **WSACancelBlockingCall()** had never been called. Note that the **WSACancelBlockingCall()** still succeeds in this case; the

only way to know with certainty that an operation was actually canceled is to check for a return code of WSAEINTR from the blocking call.

Error Codes

WSANOTINITIALISED	A successful **WSAStartup()** must occur before using this API.
WSAENETDOWN	The network subsystem has failed.
WSAEINVAL	Indicates that there is no outstanding blocking call.

3.41 WSACleanup()

Description Terminate use of the Winsock DLL.

#include <winsock.h>

int WSAAPI WSACleanup (void);

Remarks An application or DLL is required to perform a (successful) **WSAStartup()** call before it can use Winsock services. When it has completed the use of Winsock, the application or DLL must call **WSACleanup()** to deregister itself from a Winsock implementation and allow the implementation to free any resources allocated on behalf of the application or DLL. Any open connection-oriented sockets that are connected when **WSACleanup()** is called are reset; sockets which have been closed with **closesocket()** but which still have pending data to be sent are not affected--the pending data is still sent.

There must be a call to **WSACleanup()** for every call to **WSAStartup()** made by a task. Only the final **WSACleanup()** for that task does the actual cleanup; the preceding calls simply decrement an internal reference count in the Winsock DLL. A naive application may ensure that **WSACleanup()** was called enough times by calling **WSACleanup()** in a loop until it returns WSANOTINITIALISED.

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()**.

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

In a multithreaded environment, **WSACleanup()** terminates Winsock operations for all threads.

Error Codes

WSANOTINITIALISED	A successful **WSAStartup()** must occur before using this API.
WSAENETDOWN	The network subsystem has failed.
WSAEINPROGRESS	A blocking Winsock call is in progress, or the service provider is still processing a callback function (see section **B.3.6.6**).

See Also **WSAStartup()**

3.42 WSACloseEvent()

Description Closes an open event object handle.

#include <winsock.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().**

3.43 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 <winsock.h>

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

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.
lpSQOS	A pointer to the flow specs for socket *s*, one for each direction.
lpGQOS	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 the Winsock 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 winsock.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          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. A socket group and its associated ID remain valid until the last socket belonging to this socket group is closed.

lpSQOS 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 flow spec values will be ignored, respectively, for any unidirectional sockets. The first part of each memory block contains the flow parameters common to all providers, optionally followed by any provider specific portion started at *ProviderSpecificParams*. These is no provider specific portion if *SizePSP* is 0. A NULL value for *lpSQOS* indicates no application supplied QOS.

lpGQOS 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 contains the flow parameters common to all providers, optionally followed by any provider specific portion started at *ProviderSpecificParams*. These is no provider specific portion if *SizePSP* is 0. A NULL value for *lpGQOS* indicates no application-supplied group QOS.

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 (including overlapped sockets), 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 **B.3.6.6**).
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*, *lpSQOS*, and *lpGQOS* 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 *lpSQOS* and *lpGQOS* cannot be satisfied.
WSAEPROTONOSUPPORT	The *lpCallerData* augment is not supported by the service provider.
WSAETIMEDOUT	Attempt to connect timed out without establishing a connection

WSAEWOULDBLOCK	The socket is marked as 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().**

3.44 WSACreateEvent()

Description Creates a new event object.

#include <winsock.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().**

3.45 WSADuplicateSocket()

Description Create a shared socket for a specified task.

#include <winsock.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 shared socket 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 **Note: The notification semantics for shared sockets remain an open issue. The text which follows assumes the independent notification proposal.**
Shared sockets may be used in all places where regular sockets are used and are, in fact, indistinguishable from them. Socket descriptors referencing a common underlying socket share all aspects of the underlying 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'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.

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 **B.3.6.6**).
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

3.46 WSAEnumNetworkEvents()

Description Discover occurrences of network events for the indicated socket.

#include <winsock.h>

int WSAAPI WSAEnumNetworkEvents (SOCKET *s*, **WSAEVENT** *hEventObject*, **LPWSANETWORKEVENTS** *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 WSANETWORKEVENTS 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 Winsock 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 **B.3.6.6**).
WSAENOBUFS	The supplied buffer is too small.

See Also **WSAEventSelect()**

3.47 WSAEnumProtocols()

Description Retrieve information about available transport protocols.

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

Remarks This function is used to discover information about the collection of transport protocols 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 **WSASend()/WSASendto()** and **WSARecv()/WSARecvfrom()** to be invoked in a preemptive VMM context, sometimes referred to as 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**()**/WSASocket**() 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 *bNetworkByteOrder* - A flag to indicate whether the protocol's network byte order is "big-endian" or "little-endian".

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.

LPSTR *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().**

Error Codes

WSANOTINITIALISED	A successful **WSAStartup()** must occur before using this API.
WSAENETDOWN	The network subsystem has failed.
WSAEINPROGRESS	A blocking Winsock call is in progress.
WSAEINVAL	Indicates that one of the specified parameters was invalid.
WSAENOBUFS	The buffer length was too small to receive all the relevant PROTOCOL_INFO structures and associated information. Pass in a buffer at least as large as the value returned in *lpdwBufferLength*.

3.48 WSAEventSelect()

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

#include <winsock.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 non-overlapped 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()**.

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

[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.

call which returns WSAEWOULDBLOCK immediately. For example, the following sequence of operations is possible:

(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_QOS**
FD_GROUP_QOS	**getsockopt()** with option **SO_GROUP_QOS**

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 Winsock 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()/WSAConnect()** or accepted with **accept()/WSAAccept()**, and then after a send fails with WSAEWOULDBLOCK and buffer space becomes available. Therefore, an application can assume that sends are possible starting from the first FD_WRITE network event 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_QOS or SO_GROUP_QOS to get the current QOS for socket *s* or for the socket group *s* belongs to, respectively.

Error Codes

WSANOTINITIALISED	A successful **WSAStartup()** must occur before using this 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 **B.3.6.6**).
WSAENOTSOCK	The descriptor is not a socket.

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

3.49 WSAGetLastError()

Description Get the error status for the last operation which failed.

#include <winsock.h>

int WSAAPI WSAGetLastError (void);

Remarks This function returns the last network error that occurred. When a particular Winsock function indicates that an error has occurred, this function should be called to retrieve the appropriate error code.

Return Value The return value indicates the error code for the last Winsock routine performed by this thread.

See Also **WSASetLastError()**

3.50 WSAGetOverlappedResult()

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

#include <winsock.h>

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

s	Identifies the socket. This is the same socket that was specified when the overlapped operation was started by a call to **WSARecv(), WSARecvfrom(), WSASend(),** or **WSASendto()**.
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

WSAENOTSOCK	The descriptor is not a socket.
WSA_INVALID_HANDLE	The *hEvent* field of the WSAOVERLAPPED structure does not contain a valid event object handle.
WSA_INVALID_PARAMETER	One of the parameters is unacceptable.
WSA_IO_INCOMPLETE	*fWait* is FALSE and the I/O operation has not yet completed.

See Also **WSACreateEvent(), WSAWaitForMultipleEvents(), WSARecv(), WSARecvfrom(), WSASend(), WSASendto(), WSAConnect(), WSAAccept().**

3.51 WSAGetQoSByName()

Description Initializes the QoS based on a template.

#include <winsock.h>

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

s	A descriptor identifying a socket.
lpQOSName	Specifies the QoS template name.
lpQOS	A pointer to the QOS structure to be filled.

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 WSA_INVALID_PARAMETER The specified QoS template is invalid.

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

3.52 WSAHtonl()

Description Convert a **u_long** from a specified host byte order to network byte order.

#include <winsock.h>

u_long WSAAPI WSAHtonl (u_long *hostlong* **);**

netOrder A boolean indicating whether network byte order should be considered "big-endian"(*netOrder = 0),* or "little-endian" (*netOrder = 1).*

hostlong A 32-bit number in host byte order.

Remarks This routine takes a 32-bit number in the specified host byte order and returns a 32-bit number in network byte order.

Return Value **WSAHtonl()** returns the value in network byte order.

See Also **htonl**(), **htons**(), **ntohs(), ntohl(), WSAHtons(), WSANtohl(), WSANtohs**().

3.53 WSAHtons()

Description Convert a **u_short** from a specified host byte order to network byte order.

#include <winsock.h>

u_short WSAAPI WSAHtons (u_short *hostshort* **);**

netOrder A boolean indicating whether network byte order should be considered "big-endian"(*netOrder = 0),* or "little-endian" (*netOrder = 1).*

hostshort A 16-bit number in host byte order.

Remarks This routine takes a 16-bit number in the specified host byte order and returns a 16-bit number in network byte order.

Return Value **WSAHtons()** returns the value in network byte order.

See Also **htonl**(), **htons**(), **ntohs(), ntohl(), WSAHtonl(), WSANtohl(), WSANtohs**().

3.54 WASIsBlocking()

Description Determine if a blocking call is in progress.

#include <winsock.h>

BOOL WSAAPI WSAIsBlocking (void);

Remarks This function allows a task to determine if it is executing while waiting for a previous blocking call to complete.

Return Value The return value is TRUE if there is an outstanding blocking function awaiting completion. Otherwise, it is FALSE.

Comments In Win16 environments, although a call issued on a blocking socket appears to an application program as though it "blocks", the Winsock DLL has to relinquish the processor to allow other applications to run. This means that it is possible for the application which issued the blocking call to be re-entered, depending on the message(s) it receives. In this instance, the **WSAIsBlocking()** function can be used to ascertain whether the task has been re-entered while waiting for an outstanding blocking call to complete. Note that Winsock prohibits more than one outstanding call per thread.

3.55 WSANtohl()

Description Convert a **u_long** from a specified network byte order to host byte order.

#include <winsock.h>

u_long WSAAPI WSANtohl (bool *netOrder*, **u_long** *netlong* **);**

netOrder A boolean indicating whether network byte order should be considered "big-endian"(*netOrder = 0),* or "little-endian" (*netOrder = 1).*

netlong A 32-bit number in network byte order.

Remarks This routine takes a 32-bit number in the specified network byte order and returns a 32-bit number in host byte order.

Return Value **WSANtohl()** returns the value in host byte order.

See Also **ntohl(), htonl**(), **htons**(), **ntohs(), WSAHtonl(), WSAHtons(), WSANtohs**().

3.56 WSANtohs()

Description Convert a **u_short** from a specified network byte order to host byte order.

#include <winsock.h>

u_short WSAAPI WSANtohs (bool *netOrder*, **u_short** *netshort* **);**

netOrder A boolean indicating whether network byte order should be considered "big-endian"(*netOrder = 0),* or "little-endian" (*netOrder = 1).*

netshort A 16-bit number in network byte order.

Remarks This routine takes a 16-bit number in the specified network byte order and returns a 16-bit number in host byte order.

Return Value **WSANtohs()** returns the value in host byte order.

See Also **htonl**(), **htons**(), **ntohs(), ntohl(), WSAHtonl(), WSAHtons(), WSANtohl**().

3.57 WSARecv()

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

#include <winsock.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.

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

Value	Meaning

MSG_PEEK Peek at the incoming data. The data is copied into the buffer but is not removed from the input queue. {Hmmm, "input queue" implies receive buffering is occurring, which sort of defeats the purpose of overlapped I/O. Should this be a valid flag for overlapped I/O?}

MSG_OOB Process out-of-band data (See section B.3.2.3 for a discussion of this topic.)

Upon the completion of an 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*.

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 a preemptive VMM 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, FAR *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, callback functions may be invoked in a preemptive VMM context which is sometimes referred to as an interrupt context. Applications must be aware that in this special context a very limited set of runtime and Windows library functions can be safely made. As a rule, an application should confine itself to the same set of runtime functions that the Windows documentation indicates may safely be called during a multimedia timer callback function.

In Windows 95 and NT, the completion function follows the same rules as stipulated for Win32 file I/O completion routines. The completion function will not be invoked until the thread is in an alertable wait state such as can occur when the function **WSAWaitForMultipleEvents()** is invoked.

In all environments, transports do allow an application to invoke send and receive operations from within the context of the socket I/O completion function, and guarantee

that, for a given socket, I/O completion functions will not be nested. This permits time-sensitive data transmissions to occur entirely within a preemptive context.

The prototype of the completion routine is as follows:

VOID CALLBACK CompletionRoutine(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

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()**

3.58 WSARecvfrom()

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

#include <winsock.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. 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. 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.

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

Value	Meaning
MSG_PEEK	Peek at the incoming data. The data is copied into the buffer but is not removed from the input queue.
MSG_OOB	Process out-of-band data (See section B.3.2.3 for a discussion of this topic.)

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*.

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 a preemptive VMM 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, FAR * 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, callback functions may be invoked in a preemptive VMM context which is sometimes referred to as an interrupt context. Applications must be aware that in this special context a very limited set of runtime and Windows library functions can be safely made. As a rule, an application should confine itself to the same set of runtime functions that the Windows documentation indicates may safely be called during a multimedia timer callback function.

In Windows 95 and NT, the completion function follows the same rules as stipulated for Win32 file I/O completion routines. The completion function will not be invoked until the thread is in an alertable wait state such as can occur when the function **WSAWaitForMultipleEvents()** is invoked.

In all environments, transports do allow an application to invoke send and receive operations from within the context of the socket I/O completion function, and guarantee that, for a given socket, I/O completion functions will not be nested. This permits time-sensitive data transmissions to occur entirely within a preemptive context.

The prototype of the completion routine is as follows:

VOID CALLBACK CompletionRoutine(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

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()**

3.59 WSAResetEvent()

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

#include <winsock.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 WSA_INVALID_HANDLE *hEvent* is not a valid event object handle.

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

3.60 WSASend()

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

#include <winsock.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.
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 (ignored for non-overlapped sockets).
lpCompletionRoutine	A pointer to the completion routine called when the send operation has been completed (ignored for non-overlapped sockets).

Remarks **WSASend**() is used to write outgoing data on a connected socket. For overlapped sockets (created using **WSASocket()** with flag WSA_FLAG_OVERLAPPED) this will occur using overlapped I/O. If the socket, *s*, is not an overlapped socket, the last two parameters (*lpOverlapped, lpCompletionRoutine*) are ignored and **WSASend()** behaves exactly like **send()**.

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.

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 B.3.4 .

MSG_OOB Send out-of-band data (stream style socket such as SOCK_STREAM only; see also section B.3.2.3)

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.

Overlapped socket I/O:

Upon completion of an 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 a preemptive VMM 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, FAR * 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, callback functions may be invoked in a preemptive VMM context which is sometimes referred to as an interrupt context. Applications must be aware that in this special context a very limited set of runtime and Windows library functions can be safely made. As a rule, an application should confine itself to the same set of runtime functions that the Windows documentation indicates may safely be called during a multimedia timer callback function.

In Windows 95 and NT, the completion function follows the same rules as stipulated for Win32 file I/O completion routines. The completion function will not be invoked until the thread is in an alertable wait state such as can occur when the function **WSAWaitForMultipleEvents()** is invoked.

In all environments, transports do allow an application to invoke send and receive operations from within the context of the socket I/O completion function, and guarantee that, for a given socket, I/O completion functions will not be nested. This permits time-sensitive data transmissions to occur entirely within a preemptive context.

The prototype of the completion routine is as follows:

VOID CALLBACK CompletionRoutine(**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

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()**

3.61 WSASendto()

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

#include <winsock.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 (ignored for non-overlapped sockets).
lpCompletionRoutine	A pointer to the completion routine called when the send operation has been completed (ignored for non-overlapped sockets).

Remarks **WSASendto**() is used to write outgoing data on a socket asynchronously. For overlapped sockets (created using **WSASocket()** with flag WSA_FLAG_OVERLAPPED) this will occur using overlapped I/O. If the socket, *s*, is not an overlapped socket, the last two parameters (*lpOverlapped, lpCompletionRoutine*) are ignored and **WSASend()** behaves exactly like **send()**.

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.

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 B.3.4 .

MSG_OOB
: Send out-of-band data (stream style socket such as SOCK_STREAM only; see also section B.3.2.3)

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.

Overlapped socket I/O:

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 a preemptive VMM 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, FAR * 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, callback functions may be invoked in a preemptive VMM context which is sometimes referred to as an interrupt context. Applications must be aware that in this special context a very limited set of runtime and Windows library functions can be safely made. As a rule, an application should confine itself to the same set of runtime functions that the Windows documentation indicates may safely be called during a multimedia timer callback function.

In Windows 95 and NT, the completion function follows the same rules as stipulated for Win32 file I/O completion routines. The completion function will not be invoked until the thread is in an alertable wait state such as can occur when the function **WSAWaitForMultipleEvents()** is invoked.

In all environments, transports do allow an application to invoke send and receive operations from within the context of the socket I/O completion function, and guarantee that, for a given socket, I/O completion functions will not be nested. This permits time-sensitive data transmissions to occur entirely within a preemptive context.

The prototype of the completion routine is as follows:

VOID CALLBACK CompletionRoutine(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

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()**

3.62 WSASetBlockingHook()

Description Establish an application-specific blocking hook function.

#include <winsock.h>

FARPROC WSAAPI WSASetBlockingHook (FARPROC *lpBlockFunc* **);**

lpBlockFunc A pointer to the procedure instance address of the blocking function to be installed.

Remarks This function installs a new function which a Winsock implementation should use to implement blocking socket function calls.

A Winsock implementation includes a default mechanism by which blocking socket functions are implemented. The function **WSASetBlockingHook()** gives the application the ability to execute its own function at "blocking" time in place of the default function.

When an application invokes a blocking Winsock operation, Winsock initiates the operation and then enters a loop which is similar to the following pseudocode:

```
for(;;) {
     /* flush messages for good user response */
     while(BlockingHook())
          ;
     /* check for WSACancelBlockingCall() */
     if(operation_cancelled())
          break;
     /* check to see if operation completed */
     if(operation_complete())
          break;     /* normal completion */
}
```

Note that Winsock implementations may perform the above steps in a different order; for example, the check for operation complete may occur before calling the blocking hook. The default **BlockingHook()** function is equivalent to:

```
BOOL DefaultBlockingHook(void) {
     MSG msg;
     BOOL ret;
     /* get the next message if any */
     ret = (BOOL)PeekMessage(&msg,NULL,0,0,PM_REMOVE);
     /* if we got one, process it */
     if (ret) {
          TranslateMessage(&msg);
          DispatchMessage(&msg);
     }
     /* TRUE if we got a message */
     return ret;
}
```

The **WSASetBlockingHook()** function is provided to support those applications which require more complex message processing - for example, those employing the MDI (multiple document interface) model. It is not intended as a mechanism for performing general applications functions. In particular, the only Winsock function which may be

issued from a custom blocking hook function is **WSACancelBlockingCall()**, which will cause the blocking loop to terminate.

This function must be implemented on a per-task basis for non-multithreaded versions of Windows and on a per-thread basis for multithreaded versions of Windows such as Windows NT. It thus provides for a particular task or thread to replace the blocking mechanism without affecting other tasks or threads.

In multithreaded versions of Windows, there is no default blocking hook--blocking calls block the thread that makes the call. However, an application may install a specific blocking hook by calling **WSASetBlockingHook().**
This allows easy portability of applications that depend on the blocking hook behavior.

Return Value

The return value is a pointer to the procedure-instance of the previously installed blocking function. The application or library that calls the **WSASetBlockingHook ()** function should save this return value so that it can be restored if necessary. (If "nesting" is not important, the application may simply discard the value returned by **WSASetBlockingHook()** and eventually use **WSAUnhookBlockingHook()** to restore the default mechanism.) If the operation fails, a NULL pointer 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.
WSAEINPROGRESS	A blocking Winsock call is in progress, or the service provider is still processing a callback function (see section **B.3.6.6**).

See Also

WSAUnhookBlockingHook()

3.63 WSASetEvent()

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

#include <winsock.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 WSA_INVALID_HANDLE *hEvent* is not a valid event object handle.

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

3.64 WSASetLastError()

Description Set the error code which can be retrieved by **WSAGetLastError**().

#include <winsock.h>

void WSAAPI WSASetLastError (int *iError* **);**

iError	Specifies the error code to be returned by a subsequent **WSAGetLastError()** call.

Remarks This function allows an application to set the error code to be returned by a subsequent **WSAGetLastError()** call for the current thread. Note that any subsequent Winsock routine called by the application will override the error code as set by this routine.

Return Value None.

Error Codes

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

See Also **WSAGetLastError**()

3.65 WSASocket()

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

#include <winsock.h>

SOCKET WSAAPI WSASocket (int *af***, int** *type***,** int protocol, **LPROTOCOL_INFO** *lpProtocolInfo*, **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.
lpProtocolInfo	A pointer to a PROTOCOL_INFO struct that defines the characteristics of the socket to be created. If this parameter is not NULL, the first three parameters (*af, type, protocol*) are ignored.
iFlags	The socket attribute specification.

Remarks **WSASocket()** causes a socket descriptor and any related resources to be allocated and associated with a transport service provider. This association will occur in one of two ways, depending on whether the *lpProtocolInfo* parameter is specified as a NULL pointer. If *lpProtocolInfo* is non-NULL, the first three parameters (*af, type, protocol*) will be ignored, and the transport provider indicated in the referenced PROTOCOL_INFO struct will be used. Applications may obtain PROTOCOL_INFO structs by using **WSAEnumProtocols()**.

If *lpProtocolInfo* is NULL, the first three parameters (*af, type, protocol*) will determine which service provider is used. The Winsock DLL will select the first transport provider able to support the stipulated address family, socket type and protocol values.
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 may utilize **WSASend()**, **WSASendto()**, **WSARecv()**, **WSARecvfrom()** for overlapped I/O operations, which allows multiple of these operations to be initiated and in progress simultaneously.

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**()**/WSAConnect**() call. Once connected, data may be transferred using **send**()**/WSASend**() and **recv**()**/WSARecv**() calls. When a session has been completed, a **closesocket**() must be performed.

The communications protocols used to implement a reliable, connection-oriented socket ensure that data is not lost or duplicated. If data for which the peer protocol has buffer space cannot be successfully transmitted within a reasonable length of time, the connection is considered broken and subsequent calls will fail with the error code set to WSAETIMEDOUT.

Connectionless, message-oriented sockets allow sending and receiving of datagrams to and from arbitrary peers using **sendto()/WSASendto()** and **recvfrom()/WSARecvFrom()**. If such a socket is **connect()**ed to a specific peer, datagrams may be 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 **B.3.6.6**).
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**().

3.66 WSAStartup()

Description

#include <winsock.h>

int WSAAPI WSAStartup (WORD *wVersionRequested***,**
LPWSADATA *lpWSAData* **);**

wVersionRequested	The highest version of Winsock support that the caller can use. The high order byte specifies the minor version (revision) number; the low-order byte specifies the major version number.
lpWSAData	A pointer to the **WSADATA** data structure that is to receive details of the Winsock implementation.

Remarks

This function **MUST** be the first Winsock function called by an application or DLL. It allows an application or DLL to specify the version of Winsock required and to retrieve details of the specific Winsock implementation. The application or DLL may only issue further Winsock functions after a successful **WSAStartup()** invocation.

In order to support future Winsock implementations and applications which may have functionality differences from current version of Winsock, a negotiation takes place in **WSAStartup()**. The caller of **WSAStartup()** and the Winsock DLL indicate to each other the highest version that they can support, and each confirms that the other's highest version is acceptable. Upon entry to **WSAStartup()**, the Winsock DLL examines the version requested by the application. If this version is higher than the lowest version supported by the DLL, the call succeeds and the DLL returns in *wHighVersion* the highest version it supports and in *wVersion* the minimum of its high version and *wVersionRequested.* The Winsock DLL then assumes that the application will use *wVersion.* If the *wVersion* field of the **WSADATA** structure is unacceptable to the caller, it should call **WSACleanup()** and either search for another Winsock DLL or fail to initialize.

This negotiation allows both a Winsock DLL and a Winsock application to support a range of Winsock versions. An application can successfully utilize a Winsock DLL if there is any overlap in the version ranges. The following chart gives examples of how **WSAStartup()** works in conjunction with different application and Winsock DLL versions:

App versions	DLL Versions	*wVersionRequested*	*wVersion*	*wHighVersion*	End Result
1.1	1.1	1.1	1.1	1.1	use 1.1
1.0 1.1	1.0	1.1	1.0	1.0	use 1.0
1.0	1.0 1.1	1.0	1.0	1.1	use 1.0
1.1	1.0 1.1	1.1	1.1	1.1	use 1.1
1.1	1.0	1.1	1.0	1.0	Application fails
1.0	1.1	1.0	---	---	WSAVERNOTSUPPORTED
1.0 1.1	1.0 1.1	1.1	1.1	1.1	use 1.1
1.1 2.0	1.1	2.0	1.1	1.1	use 1.1
2.0	1.1	2.0	1.1	1.1	Application fails

The following code fragment demonstrates how an application which supports only version 1.1 of Winsock makes a **WSAStartup()** call:

```
WORD wVersionRequested;
WSADATA wsaData;
int err;

wVersionRequested = MAKEWORD( 1, 1 );

err = WSAStartup( wVersionRequested, &wsaData );
if ( err != 0 ) {
    /* Tell the user that we couldn't find a useable */
    /* Winsock DLL.                                  */
    return;
}

/* Confirm that the Winsock DLL supports 1.1.*/
/* Note that if the DLL supports versions greater    */
/* than 1.1 in addition to 1.1, it will still return */
/* 1.1 in wVersion since that is the version we      */
/* requested.                                        */

if ( LOBYTE( wsaData.wVersion ) != 1 ||
         HIBYTE( wsaData.wVersion ) != 1 ) {
    /* Tell the user that we couldn't find a useable */
    /* Winsock DLL.                                  */
    WSACleanup( );
    return;
}

/* The Winsock DLL is acceptable.  Proceed.  */
```

And this code fragment demonstrates how a Winsock DLL which supports only version 1.1 performs the **WSAStartup()** negotiation:

```
/* Make sure that the version requested is >= 1.1.   */
/* The low byte is the major version and the high    */
/* byte is the minor version.                        */

if ( LOBYTE( wVersionRequested ) < 1 ||
     ( LOBYTE( wVersionRequested ) == 1 &&
       HIBYTE( wVersionRequested ) < 1 ) {
    return WSAVERNOTSUPPORTED;
}

/* Since we only support 1.1, set both wVersion and  */
/* wHighVersion to 1.1.                              */

lpWsaData->wVersion = MAKEWORD( 1, 1 );
lpWsaData->wHighVersion = MAKEWORD( 1, 1 );
```

Once an application or DLL has made a successful **WSAStartup()** call, it may proceed to make other Winsock calls as needed. When it has finished using the services of the Winsock DLL, the application or DLL must call **WSACleanup()** in order to allow the Winsock DLL to free any resources for the application.

Details of the actual Winsock implementation are described in the WSAData structure defined as follows:

```
struct WSAData {
      WORD              wVersion;
      WORD              wHighVersion;
      char              szDescription[WSADESCRIPTION_LEN+1];
      char              szSystemStatus[WSASYSSTATUS_LEN+1];
      unsigned short    iMaxSockets;
      unsigned short    iMaxUdpDg;
      char FAR *        lpVendorInfo;
};
```

The members of this structure are:

Element	Usage
wVersion	The version of the Winsock specification that the Winsock DLL expects the caller to use.
wHighVersion	The highest version of the Winsock specification that this DLL can support (also encoded as above). Normally this will be the same as *wVersion*.
szDescription	A null-terminated ASCII string into which the Winsock DLL copies a description of the Winsock implementation. The text (up to 256 characters in length) may contain any characters except control and formatting characters: the most likely use that an application will put this to is to display it (possibly truncated) in a status message.
szSystemStatus	A null-terminated ASCII string into which the Winsock DLL copies relevant status or configuration information. The Winsock DLL should use this field only if the information might be useful to the user or support staff: it should not be considered as an extension of the *szDescription* field.
iMaxSockets	The maximum number of sockets which a single process can potentially open. Application writers may use this number as a crude indication of whether the Winsock implementation is usable by the application. For example, an X Windows server might check *iMaxSockets* when first started: if it is less than 8, the application would display an error message instructing the user to reconfigure the networking software. (This is a situation in which the *szSystemStatus* text might be used.) Obviously there is no guarantee that a particular application can actually allocate *iMaxSockets* sockets, since there may be other Winsock applications in use.
iMaxUdpDg	This value is retained for compatibility with Windows Sockets specification 1.1, but should not be used when developing new applications. The value supplied here will be obtained from the first UDP protocol provider found. For the actual maximum message size specific to a particular Winsock service provider and socket type, applications should use **getsockopt()** to retrieve the value of option SO_MAX_MSG_SIZE after a socket has been created.
lpVendorInfo	This value should be ignored. It is retained for compatibility with Windows Sockets specification 1.1. The value supplied here will be obtained from the first UDP protocol provider found . Applications needing to access vendor-specific configuration information should use **getsockopt()** to retrieve the value of option PVD_CONFIG. The definition of this value (if utilized) is beyond the scope of this specification.

Note that an application should ignore the *iMaxsockets*, *iMaxUdpDg*, and *lpVendorInfo* fields in WSAData if the value in *wVersion* after a successful call to **WSAStartup**() is at

least 2.0. This is because the architecture of Winsock has been changed in version 2.0 to support multiple providers, and WSAData no longer applies to a single vendor's stack. Two new socket options are introduced to supply provider-specific information: SO_MAX_MSG_SIZE (replaces the *iMaxUdpDg* element) and PVD_CONFIG (allows any other provider-specific configuration to occur).

An application or DLL may call **WSAStartup()** more than once if it needs to obtain the WSAData structure information more than once. On each such call the application may specify any version number supported by the DLL.

There must be one **WSACleanup()** call corresponding to every **WSAStartup()** call to allow third-party DLLs to make use of a Winsock DLL on behalf of an application. This means, for example, that if an application calls **WSAStartup()** three times, it must call **WSACleanup()** three times. The first two calls to **WSACleanup()** do nothing except decrement an internal counter; the final **WSACleanup()** call for the task does all necessary resource deallocation for the task.

Return Value

WSAStartup() returns zero if successful. Otherwise it returns one of the error codes listed below. Note that the normal mechanism whereby the application calls **WSAGetLastError()** to determine the error code cannot be used, since the Winsock DLL may not have established the client data area where the "last error" information is stored.

Error Codes

WSASYSNOTREADY	Indicates that the underlying network subsystem is not ready for network communication.
WSAVERNOTSUPPORTED	The version of Winsock support requested is not provided by this particular Winsock implementation.

See Also

send(), **sendto()**, **WSACleanup()**

3.67 WSAUnhookBlockingHook()

Description Restore the default blocking hook function.

#include <winsock.h>

int WSAAPI WSAUnhookBlockingHook (void);

Remarks This function removes any previous blocking hook that has been installed and reinstalls the default blocking mechanism.

WSAUnhookBlockingHook() will always install the default mechanism, not the previous mechanism. If an application wish to nest blocking hooks - i.e. to establish a temporary blocking hook function and then revert to the previous mechanism (whether the default or one established by an earlier **WSASetBlockingHook()**) - it must save and restore the value returned by **WSASetBlockingHook()**; it cannot use **WSAUnhookBlockingHook()**.

In multithreaded versions of Windows such as Windows NT, there is no default blocking hook. Calling **WSAUnhookBlockingHook()** disables any blocking hook installed by the application and any blocking calls made block the thread which made the call.

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.

See Also **WSASetBlockingHook()**

3.68 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 <winsock.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_EVENT_0 to (WSA_WAIT_EVENT_0 + cEvents - 1)
: If *fWaitAll* is TRUE, the return value indicates that the state of all specified event objects is signaled. If *fWaitAll* is FALSE, the return value minus WSA_WAIT_EVENT_0 indicates the *lphEvents* array index of the object that satisfied the wait.

WAIT_IO_COMPLETION (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

WSA_NOT_ENOUGH_MEMORY	Not enough free memory available to complete the operation
WSA_INVALID_HANDLE	One or more of the values in the *lphEvents* array is not a valid event object handle.
WSA_INVALID_PARAMETER	The *cEvents* parameter does not contain a valid handle count.

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

Appendix A. Error Codes and Header Files

A.1 Error Codes

The following is a list of possible error codes returned by the **WSAGetLastError()** call, along with their explanations. The error numbers are consistently set across all Winsock-compliant implementations.

Winsock code	Berkeley equivalent	Error	Interpretation
WSAEINTR	EINTR	10004	As in standard C
WSAEBADF	EBADF	10009	As in standard C
WSAEACCES	EACCES	10013	As in standard C
WSAEFAULT	EFAULT	10014	As in standard C
WSAEINVAL	EINVAL	10022	As in standard C
WSAEMFILE	EMFILE	10024	As in standard C
WSAEWOULDBLOCK	EWOULDBLOCK	10035	As in BSD
WSAEINPROGRESS	EINPROGRESS	10036	This error is returned if any Winsock function is called while a blocking function is in progress.
WSAEALREADY	EALREADY	10037	As in BSD
WSAENOTSOCK	ENOTSOCK	10038	As in BSD
WSAEDESTADDRREQ	EDESTADDRREQ	10039	As in BSD
WSAEMSGSIZE	EMSGSIZE	10040	As in BSD
WSAEPROTOTYPE	EPROTOTYPE	10041	As in BSD
WSAENOPROTOOPT	ENOPROTOOPT	10042	As in BSD
WSAEPROTONOSUPPORT	EPROTONOSUPPORT	10043	As in BSD
WSAESOCKTNOSUPPORT	ESOCKTNOSUPPORT	10044	As in BSD
WSAEOPNOTSUPP	EOPNOTSUPP	10045	As in BSD
WSAEPFNOSUPPORT	EPFNOSUPPORT	10046	As in BSD
WSAEAFNOSUPPORT	EAFNOSUPPORT	10047	As in BSD
WSAEADDRINUSE	EADDRINUSE	10048	As in BSD
WSAEADDRNOTAVAIL	EADDRNOTAVAIL	10049	As in BSD
WSAENETDOWN	ENETDOWN	10050	As in BSD. This error may be reported at any time if the Winsock implementation detects an underlying failure.
WSAENETUNREACH	ENETUNREACH	10051	As in BSD
WSAENETRESET	ENETRESET	10052	As in BSD
WSAECONNABORTED	ECONNABORTED	10053	As in BSD
WSAECONNRESET	ECONNRESET	10054	As in BSD
WSAENOBUFS	ENOBUFS	10055	As in BSD
WSAEISCONN	EISCONN	10056	As in BSD
WSAENOTCONN	ENOTCONN	10057	As in BSD
WSAESHUTDOWN	ESHUTDOWN	10058	As in BSD
WSAETOOMANYREFS	ETOOMANYREFS	10059	As in BSD
WSAETIMEDOUT	ETIMEDOUT	10060	As in BSD
WSAECONNREFUSED	ECONNREFUSED	10061	As in BSD
WSAELOOP	ELOOP	10062	As in BSD
WSAENAMETOOLONG	ENAMETOOLONG	10063	As in BSD
WSAEHOSTDOWN	EHOSTDOWN	10064	As in BSD
WSAEHOSTUNREACH	EHOSTUNREACH	10065	As in BSD
WSASYSNOTREADY		10091	Returned by **WSAStartup()** indicating that the network subsystem is unusable.
WSAVERNOTSUPPORTED		10092	Returned by **WSAStartup()** indicating that the Winsock DLL cannot support this app.
WSANOTINITIALISED		10093	Returned by any function except **WSAStartup()** indicating that a successful **WSAStartup()** has not yet been performed.
WSAHOST_NOT_FOUND	HOST_NOT_FOUND	11001	As in BSD.
WSATRY_AGAIN	TRY_AGAIN	11002	As in BSD
WSANO_RECOVERY	NO_RECOVERY	11003	As in BSD
WSANO_DATA	NO_DATA	11004	As in BSD

The first set of definitions is present to resolve contentions between standard C error codes which may be defined inconsistently between various C compilers.

The second set of definitions provides Winsock versions of regular Berkeley Sockets error codes.

The third set of definitions consists of extended Winsock-specific error codes.

The fourth set of errors are returned by Winsock **getXbyY()** and **WSAAsyncGetXByY()** functions, and correspond to the errors which in Berkeley software would be returned in the *h_errno* variable. They correspond to various failures which may be returned by the Domain Name Service. If the Winsock provider does not use the DNS, it will use the most appropriate code. In general, a Winsock application should interpret WSAHOST_NOT_FOUND and WSANO_DATA as indicating that the key (name, address, etc.) was not found, while WSATRY_AGAIN and WSANO_RECOVERY suggest that the name service itself is non-operational.

The error numbers are derived from the **Winsock.h** header file listed in section A.2.2 , and are based on the fact that Winsock error numbers are computed by adding 10000 to the "normal" Berkeley error number.

Note that this table does not include all of the error codes defined in **Winsock.h**. This is because it includes only errors which might reasonably be returned by a Winsock implementation: **Winsock.h**, on the other hand, includes a full set of BSD definitions to ensure compatibility with ported software.

A.2 Header Files

A.2.1 Berkeley Header Files

The Winsock SDK includes a set of vestigial header files with names that match a number of the header files in the Berkeley software distribution. These files are provided for source code compatibility only, and each consists of three lines:

```
#ifndef _WinsockAPI_
#include <Winsock.h>
#endif
```

The header files provided for compatibility are:
netdb.h
arpa/inet.h
sys/time.h
sys/socket.h
netinet/in.h

The file **Winsock.h** contains all of the type and structure definitions, constants, macros, and function prototypes used by the Winsock specification. An application writer may choose to ignore the compatibility headers and include **Winsock.h** in each source file.

A.2.2 Winsock Header File - Winsock.h

The **Winsock.h** header file includes a number of types and definitions from the standard Windows header file **windows.h**. The **windows.h** in the Windows 3.0 SDK (Software Developer's Kit) lacks a `#include` guard, so if you need to include **windows.h** as well as **Winsock.h**, you should define the symbol _INC_WINDOWS before #including **Winsock.h**, as follows:

```
#include <windows.h>
#define _INC_WINDOWS
#include <Winsock.h>
```

Users of the SDK for Windows 3.1 and later need not do this.

A Winsock service provider vendor **MUST NOT** make any modifications to this header file which could impact binary compatibility of Winsock applications. The constant values, function parameters and return codes, and the like must remain consistent across all Winsock service provider vendors.

```
/* Winsock.H--definitions to be used with the Winsock 2.0 DLL and
 *             Winsock 2.0 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.
 */

#ifndef _WINSOCK2API_
#define _WINSOCK2API_

#ifndef _WINSOCKAPI_
#include <winsock.h>
#endif

#ifdef WIN32

        #define WSAAPI                  FAR PASCAL
        #define WSATASK                 HANDLE
        #define WSAEVENT                HANDLE
        #define LPWSAEVENT              LPHANDLE

        #define WSA_INVALID_EVENT       ((WSAEVENT)NULL)
        #define WSA_MAXIMUM_WAIT_EVENTS (MAXIMUM_WAIT_OBJECTS)
        #define WSA_WAIT_FAILED         (WAIT_FAILED)
        #define WSA_WAIT_EVENT_0        (WAIT_OBJECT_0)
        #define WSA_WAIT_TIMEOUT        (WAIT_TIMEOUT)
        #define WSA_INFINITE            (INFINITE)
        #define WSA_IO_PENDING          (ERROR_IO_PENDING)
        #define WSA_IO_INCOMPLETE       (ERROR_IO_INCOMPLETE)
        #define WSA_INVALID_HANDLE      (ERROR_INVALID_HANDLE)
        #define WSA_INVALID_PARAMETER   (ERROR_INVALID_PARAMETER)
        #define WSA_NOT_ENOUGH_MEMORY   (ERROR_NOT_ENOUGH_MEMORY)

        #define WSAEDISCON               ???

        #define WSAOVERLAPPED           OVERLAPPED
        #define LPWSAOVERLAPPED         LPOVERLAPPED

#else // WIN16

        #define WSAAPI                  FAR PASCAL
        #define WSATASK                 HTASK
        typedef DWORD                   WSAEVENT, FAR * LPWSAEVENT;

        #define WSA_INVALID_EVENT       ((WSAEVENT)NULL)
        #define WSA_MAXIMUM_WAIT_EVENTS ((DWORD)64)
        #define WSA_WAIT_FAILED         ((DWORD)-1L)
        #define WSA_WAIT_EVENT_0        ((DWORD)0)
```

```
        #define WSA_WAIT_TIMEOUT        ((DWORD)0x102L)
        #define WSA_INFINITE            ((DWORD)-1L)
        #define WSA_IO_PENDING          (WSAEWOULDBLOCK)
        #define WSA_IO_INCOMPLETE       (WSAEWOULDBLOCK)
        #define WSA_INVALID_HANDLE      (WSAENOTSOCK)
        #define WSA_INVALID_PARAMETER   (WSAEINVAL)
        #define WSA_NOT_ENOUGH_MEMORY   (WSAENOBUFFS)

        #define WSAEDISCON               ???

        typedef struct _WSAOVERLAPPED
        {
            DWORD    Internal;
            DWORD    InternalHigh;
            DWORD    Offset;
            DWORD    OffsetHigh;
            WSAEVENT hEvent;
        } WSAOVERLAPPED, FAR * LPWSAOVERLAPPED;

#endif  // WIN32

/*
 * Winsock 2.0 extension -- new options
 */
#define SO_QOS            0x2001          /* QOS for sockets */
#define SO_GROUP_QOS      0x2002          /* QOS for socket groups */
#define SO_GROUP_ID       0x2003          /* sharing the same physical connection */
#define SO_GROUP_PRIORITY 0x2004          /* the relative priority with a group */
#define SO_MAX_MSG_SIZE    0x2005          /* maximum datagram size */
#define PVD_CALL_ID       0x3001          /* an opaque Winsock call ID */
#define PVD_CONFIG        0x3002          /* configuration info for service provider */
#define TAPI_DEVICE_ID    0x3003          /* a TAPI line device ID */

/*
 * Winsock 2.0 extension -- level number for (get/set)socketopt() to apply to
 *                          service provider level.
 */
#define SOL_PROVIDER    IPPROTO_TCP     /* options for service provider level */

/*
 * Winsock 2.0 extension -- WSABUF and QOS struct
 */
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 now
       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;

/*
```

```
 * Winsock 2.0 extension -- manifest constants for return values of the condition function
 */
#define CF_ACCEPT       0x0000
#define CF_REJECT       0x0001
#define CF_DEFER        0x0002

/*
 * Winsock 2.0 extension -- manifest constants for shutdown()
 */
#define SD_RECEIVE      0x00
#define SD_SEND         0x01
#define SD_BOTH         0x02

/*
 * Winsock 2.0 extension -- data type and manifest constants for socket groups
 */
typedef unsigned int            GROUP;

#define SG_UNCONSTRAINED_GROUP   0x01
#define SG_CONSTRAINED_GROUP     0x02

/*
 * Winsock 2.0 extension -- new flags for WSAAsyncSelect() and WSAEventSelect()
 */
#define FD_QOS          0x40
#define FD_GROUP_QOS    0x80

/*
 * Winsock 2.0 extension -- data type for WSAEnumNetworkEvents()
 */
typedef stuct _WSANETWORKEVENTS {
       long lNetworkEvent,
       int iErrorCode
} WSANETWORKEVENTS, FAR * LPWSANETWORKEVENTS;

/*
 * Winsock 2.0 extension -- data type for the condition function in WSAAccept()
 */
typedef int (CALLBACK * LPCONDITIONPROC) (LPWSABUF lpCallerId,
                                          LPWSABUF lpCallerData,
                                          LPQOS lpSQOS,
                                          LPQOS lpGQOS,
                                          LPWSABUF lpCalleeId,
                                          LPWSABUF lpCalleeData,
                                          GROUP FAR * g,
                                          DWORD dwCallbackData);

/* Winsock 2.0 API new function prototypes */

#ifdef __cplusplus
extern "C" {
#endif

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

BOOL WSAAPI WSACloseEvent (WSAEVENT hEvent);

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

WSAEVENT WSAAPI WSACreateEvent (VOID);

SOCKET WSAAPI WSADuplicateSocket(SOCKET s,
                                 HTASK hTargetTask);
```

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

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

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

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

BOOL WSAAPI WSAGetQOSByname (SOCKET s,
                             LPWSABUF lpQOSName,
                             LPQOS lpQOS);

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

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

BOOL WSAAPI WSAResetEvent(WSAEVENT hEvent);

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

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

BOOL WSAAPI WSASetEvent(WSAEVENT hEvent);

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

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

#ifdef __cplusplus
```

```
}
#endif


#endif  /* _WINSOCK2API_ */
```

Appendix B. Windows Sockets Introduction

The following material is targetted at those desiring a brief introduction to the socket programing paradigm in general and/or Windows Sockets.

> **Note:**
> **This material was extracted from the Winsock 1.1 specification and is not yet updated to reflect Winsock 2 extensions**.

B.1 Berkeley Sockets

The Windows Sockets interface is patternerd after the Berkeley Sockets programming model which is the de facto standard for TCP/IP networking. It is intended to provide a high degree of familiarity for programmers who are used to programming with sockets in UNIX[3] and other environments, and to simplify the task of porting existing sockets-based source code.

B.2 Microsoft Windows and Windows-specific extensions

The Winsock API is intended to be usable within all implementations and versions of Microsoft Windows from Microsoft Windows Version 3.0 onwards. It thus provides for Winsock applications in both 16 and 32 bit operating environments.

Winsock makes provisions for multithreaded Windows processes. A process contains one or more threads of execution. In the Windows 3.1 non-multithreaded world, a task corresponds to a process with a single thread. All references to threads in this document refer to actual "threads" in multithreaded Windows environments. In non multithreaded environments (such as Windows 3.0), use of the term thread refers to a Windows process.

The Microsoft Windows extensions included in Windows Sockets are provided to allow application developers to create software which conforms to the Windows programming model. It is expected that this will facilitate the creation of robust and high-performance applications, and will improve the cooperative multitasking of applications within non-preemptive versions of Windows. With the exception of **WSAStartup()** and **WSACleanup()** their use is not mandatory.

B.3 PROGRAMMING WITH SOCKETS

B.3.1 Winsock Installation Checking

To detect the presence of a Winsock implementation on a system, an application which has been linked with the Winsock Import Library may simply call the **WSAStartup()** routine. If an application wishes to be a little more sophisticated it can examine the $PATH environment variable and search for instances of Winsock.DLLs. (There may be multiple versions of Winsock.DLL in the system.) For each instance it can issue a **LoadLibrary()** call and use the **WSAStartup()** routine to discover implementation specific data.

B.3.2 Sockets

The following material is derived from the document "An Advanced 4.3BSD Interprocess Communication Tutorial" by Samuel J. Leffler, Robert S. Fabry, William N. Joy, Phil Lapsley, Steve Miller, and Chris Torek.

B.3.2.1 Socket Types

The basic building block for communication is the socket. A socket is an endpoint of communication to which a name may be bound. Each socket in use has a type and an associated process. Sockets exist within communication domains. A communication domain is an abstraction introduced to bundle common properties of threads communicating through sockets. Sockets normally exchange data only with sockets

---

[3] UNIX is a trademark of Unix System Laboratories, Inc.

in the same domain (it may be possible to cross domain boundaries, but only if some translation process is performed).

Sockets are typed according to the communication properties visible to a user. Applications are presumed to communicate only between sockets of the same type, although there is nothing that prevents communication between sockets of different types should the underlying communication protocols support this.

TCP/IP stacks support two socket types: SOCK_STREAM which provides a connection-oriented, reliable byte stream, and SOCK_DGRAM which provides connectionless, unreliable datagram service.

Raw sockets provide a mechanism to bypass the usual protocol layers and are very much dependent on the particulars of each service provider. Service providers are not required to support raw sockets.

B.3.2.2 Client-server model

The most commonly used paradigm in constructing distributed applications is the client/server model. In this scheme client applications request services from a server application. This implies an asymmetry in establishing communication between the client and server.

The client and server require a well-known set of conventions before service may be rendered (and accepted). This set of conventions comprises a protocol which must be implemented at both ends of a connection. Depending on the situation, the protocol may be symmetric or asymmetric. In a symmetric protocol, either side may play the master or slave roles. In an asymmetric protocol, one side is immutably recognized as the master, with the other as the slave. An example of a symmetric protocol is the TELNET protocol used in the Internet for remote terminal emulation. An example of an asymmetric protocol is the Internet file transfer protocol, FTP. No matter whether the specific protocol used in obtaining a service is symmetric or asymmetric, when accessing a service there is a "client process" and a "server process".

A server application normally listens at a well-known address for service requests. That is, the server process remains dormant until a connection is requested by a client's connection to the server's address. At such a time the server process "wakes up" and services the client, performing whatever appropriate actions the client requests of it. While connection-oriented services are the norm, some services are based on the use of datagram sockets.

B.3.2.3 Out-of-band data

Note: The following discussion of out-of-band data, also referred to as TCP Urgent data, follows the model used in the Berkeley software distribution. Users and implementors should be aware of the fact that there are at present two conflicting interpretations of RFC 793 (in which the concept is introduced), and that the implementation of out-of-band data in the Berkeley Software Distribution does not conform to the Host Requirements laid down in RFC 1122. To minimize interoperability problems, applications writers are advised not to use out-of-band data unless this is required in order to interoperate with an existing service. Winsock suppliers are urged to document the out-of-band semantics (BSD or RFC 1122) which their product implements. It is beyond the scope of this specification to mandate a particular set of semantics for out-of-band data handling.

Note that the out-of-band data functionality is mainly inherited from the TCP/IP world, and may not be available to other communication domains supported by this specification.

The stream socket abstraction includes the notion of "out of band" data. Out-of-band data is a logically independent transmission channel associated with each pair of connected stream sockets. Out-of-band data is delivered to the user independently of normal data. The abstraction defines that the out-of-band data

facilities must support the reliable delivery of at least one out-of-band message at a time. This message may contain at least one byte of data, and at least one message may be pending delivery to the user at any one time. For communications protocols which support only in-band signaling (i.e., the urgent data is delivered in sequence with the normal data), the system normally extracts the data from the normal data stream and stores it separately. This allows users to choose between receiving the urgent data in order and receiving it out of sequence without having to buffer all the intervening data. It is possible to "peek" at out-of-band data.

An application may prefer to process out-of-band data "in-line", as part of the normal data stream. This is achieved by setting the socket option SO_OOBINLINE (see section 3.28 , **setsockopt()**). In this case, the application may wish to determine whether any of the unread data is "urgent" (the term usually applied to in-line out-of-band data). To facilitate this, the Winsock implementation will maintain a logical "mark" in the data stream to indicate the point at which the out-of-band data was sent. An application can use the SIOCATMARK **ioctlsocket()** command (see section 3.19) to determine whether there is any unread data preceding the mark. For example, it might use this to resynchronize with its peer by ensuring that all data up to the mark in the data stream is discarded when appropriate.

The **WSAAsyncSelect()** or **WSACallbackSelect()** routine is particularly well suited to handling notification of the presence of out-of-band-data.

B.3.2.4 Broadcasting

By using a datagram socket, it is possible to send broadcast packets on many networks supported by the system. The underlying networks must themselves support broadcast: Winsock provides no simulation of broadcast in software. Broadcast messages can place a high load on a network, since they force every host on the network to service them. Consequently, the ability to send broadcast packets has been limited to sockets which are explicitly marked as allowing broadcasting. Broadcast is typically used for one of two reasons: to find a resource on a local network without prior knowledge of its address, or to implement important functions such as routing require that information be sent to all accessible neighbors.

The destination address of the message to be broadcast depends on the network(s) on which the message is to be broadcast. The Internet domain supports a shorthand notation for broadcast on the local network, namely the address INADDR_BROADCAST. Received broadcast messages contain the senders address and port, as datagram sockets must be bound before use.

Some types of network support the notion of different types of broadcast. For example, the IEEE 802.5 token ring architecture supports the use of link-level broadcast indicators, which control whether broadcasts are forwarded by bridges. The Winsock specification does not provide any way to control the semantics of broadcasting.

B.3.3 Byte Ordering

The Intel byte ordering is like that of the DEC VAX[4], and therefore differs from the Internet and 68000[5]-type processor byte ordering. Thus care must be taken to ensure correct orientation.

Any reference to addresses or port numbers passed to or from a Winsock routine must be in network order. In the case of IP, this includes the IP address and port fields of a **struct sockaddr_in** (but not the *sin_family* field).

Consider an application which normally contacts a server on the TCP port corresponding to the "time" service, but which provides a mechanism for the user to specify that an alternative port is to be used. The port number returned by **getservbyname()** is already in network order, which is the format required for

---

[4] VAX is a trademark of Digital Equipment Corporation.
[5] 68000 is a trademark of Motorola, Inc.

constructing an address, so no translation is required. However if the user elects to use a different port, entered as an integer, the application must convert this from host to network order (using the **htons()** function) before using it to construct an address. Conversely, if the application wishes to display the number of the port within an address (returned via, e.g., **getpeername()**), the port number must be converted from network to host order (using **ntohs()**) before it can be displayed.

Since the Intel and Internet byte orders are different, the conversions described above are unavoidable. Application writers are cautioned that they should use the standard conversion functions provided as part of Winsock rather than writing their own conversion code, since future implementations of Winsock are likely to run on systems for which the host order is identical to the network byte order. Only applications which use the standard conversion functions are likely to be portable.

B.3.4 Socket Options

Winsock supports a subset of the Berkley socket options. Details for these options are described in the pages describing **setsockopt()** and **getsockopt()**.

B.3.5 Database Files

The **getXbyY()**[6] and **WSAAsyncGetXBYY()** classes of routines are provided for retrieving network specific information. The **getXbyY()** routines were originally designed (in the first Berkeley UNIX releases) as mechanisms for looking up information in text databases. Although the information may be retrieved by the Winsock implementation in different ways, a Winsock application requests such information in a consistent manner through either the **getXbyY()** or the **WSAAsyncGetXBYY()** class of routines.

B.3.6 Deviation from Berkeley Sockets

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

B.3.6.1 socket data type and error values

A new data type, SOCKET, has been defined. The definition of this type was necessary for future enhancements to the Winsock specification, such as being able to use sockets as file handles in Windows NT[7]. Definition of this type also facilitates porting of applications to a Win/32 environment, as the type will automatically be promoted from 16 to 32 bits.

In UNIX, all handles, including socket handles, are small, non-negative integers, and some applications make assumptions that this will be true. Winsock handles have no restrictions, other than that the value INVALID_SOCKET is not a valid socket. Socket handles may take any value in the range 0 to INVALID_SOCKET-1.

Because the SOCKET type is unsigned, compiling existing source code from, for example, a UNIX environment may lead to compiler warnings about signed/unsigned data type mismatches.

This means, for example, that checking for errors when the **socket()** and **accept()** routines return should not be done by comparing the return value with -1, or seeing if the value is negative (both common, and legal, approaches in BSD). Instead, an application should use the manifest constant INVALID_SOCKET as defined in Winsock**.h**. For example:

---

[6] This specification uses the function name **getXbyY()** to represent the set of routines **gethostbyaddr()**, **gethostbyname()**, etc. Similarly **WSAAsyncGetXBYY()** represents **WSAAsyncGetHostByAddr()**, etc.
[7] NT and Windows/NT are trademarks of Microsoft Corporation.

TYPICAL BSD STYLE:

```
s = socket(...);
if (s == -1)       /* or s < 0 */
      {...}
```

PREFERRED STYLE:

```
s = socket(...);
if (s == INVALID_SOCKET)
      {...}
```

B.3.6.2 select() and FD_*

Because a SOCKET is no longer represented by the UNIX-style "small non-negative integer", the implementation of the **select()** function was changed in Winsock. Each set of sockets is still represented by the fd_set type, but instead of being stored as a bitmask the set is implemented as an array of SOCKETs. To avoid potential problems, applications must adhere to the use of the FD_XXX macros to set, initialize, clear, and check the fd_set structures.

B.3.6.3 Error codes - errno, h_errno & WSAGetLastError()

Error codes set by Winsock are NOT made available via the errno variable. Additionally, for the **getXbyY()** class of functions, error codes are NOT made available via the h_errno variable. Instead, error codes are accessed by using the **WSAGetLastError()** API described in section 3.49 . This function is provided in Winsock as a precursor (and eventually an alias) for the Win32 function **GetLastError()**. This is intended to provide a reliable way for a thread in a multi-threaded process to obtain per-thread error information.

For compatibility with BSD, an application may choose to include a line of the form:

```
#define errno WSAGetLastError()
```

This will allow networking code which was written to use the global errno to work correctly in a single-threaded environment. There are, obviously, some drawbacks. If a source file includes code which inspects errno for both socket and non-socket functions, this mechanism cannot be used. Furthermore, it is not possible for an application to assign a new value to errno. (In Winsock the function **WSASetLastError()** may be used for this purpose.)

TYPICAL BSD STYLE:

```
r = recv(...);
if (r == -1
    && errno == EWOULDBLOCK)
      {...}
```

PREFERRED STYLE:

```
r = recv(...);
if (r == -1       /* (but see below) */
    && WSAGetLastError() == EWOULDBLOCK)
      {...}
```

Although error constants consistent with 4.3 Berkeley Sockets are provided for compatibility purposes, applications should, where possible, use the "WSA" error code definitions. For example, a more accurate version of the above source code fragment is:

```
r = recv(...);
if (r == -1        /* (but see below) */
    && WSAGetLastError() == WSAEWOULDBLOCK)
      {...}
```

B.3.6.4 Pointers

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

B.3.6.5 Renamed functions

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

B.3.6.5.1 close() & closesocket()

In Berkeley Sockets, sockets are represented by standard file descriptors, and so the **close()** function can be used to close sockets as well as regular files. While nothing in the Winsock prevents an implementation from using regular file handles to identify sockets, nothing requires it either. For Windows 3.1, socket descriptors do not correspond to regular file handles, and file operations such as **read()**, **write()**, and **close()** will not work correctly when applied to sockets. Sockets must be closed by using the **closesocket()** routine. Using the **close()** routine to close a socket is incorrect and the effects of doing so are undefined by this specification.

B.3.6.5.2 ioctl() & ioctlsocket()

Various C language run-time systems use the **ioctl()** routine for purposes unrelated to Winsock. For this reason we have defined the routine **ioctlsocket()** which is used to handle socket functions which in the Berkeley Software Distribution are performed using **ioctl()** and **fcntl()**.

B.3.6.6 Blocking routines & EINPROGRESS

Although blocking operations on sockets are supported under Winsock, their use is strongly discouraged. Programmers who are constrained to use blocking mode – for example, as part of an existing application which is to be ported – should be aware of the semantics of blocking operations in Winsock. See section B.4.1.1 for more details.

An application could also receive WSAEINPROGRESS if the underlying service provider believes it is still in the process of invoking a callback function for some other socket (possibly belonging to another application) such as might occur if the callback function incorrectly yielded the processor. A possible recovery mechanism would be to yield the processor and try again later.

B.3.6.7 Maximum number of sockets supported

The maximum number of sockets supported by a particular Winsock service provider is implementation specific. An application should make no assumptions about the availability of a certain number of sockets. This topic is addressed further in section 3.66 , **WSAStartup()**. However, independent of the number of sockets supported by a particular implementation is the issue of the maximum number of sockets which an application can actually make use of.

The maximum number of sockets which a Winsock application can make use of is determined at the application's compile time by the manifest constant FD_SETSIZE. This value is used in constructing the fd_set structures used in **select()** (see section 3.25). The default value in Winsock**.h** is 64. If an application is designed to be capable of working with more than 64 sockets, the implementor should define the manifest FD_SETSIZE in every source file before including Winsock**.h**. One way of doing this may be to include the definition within the compiler options in the makefile, for example adding -DFD_SETSIZE=128 as an option to the compiler command line for Microsoft C. It must be emphasized

that defining FD_SETSIZE as a particular value has no effect on the actual number of sockets provided by a Winsock service provider.

B.3.6.8 Include files

For ease of portability of existing Berkeley sockets based source code, a number of standard Berkeley include files are supported. However, these Berkeley header files merely include the **Winsock.h** include file, and it is therefore sufficient (and recommended) that Winsock application source files simply include **Winsock.h**.

B.3.6.9 Return values on API failure

The manifest constant SOCKET_ERROR is provided for checking API failure. Although use of this constant is not mandatory, it is recommended. The following example illustrates the use of the SOCKET_ERROR constant:

TYPICAL BSD STYLE:

```
r = recv(...);
if (r == -1     /* or r < 0 */
    && errno == EWOULDBLOCK)
      {...}
```

PREFERRED STYLE:

```
r = recv(...);
if (r == SOCKET_ERROR
    && WSAGetLastError() == WSAEWOULDBLOCK)
      {...}
```

B.3.6.10 Raw Sockets

The Winsock specification does not mandate that a Winsock service provider support raw sockets, that is, sockets of type SOCK_RAW. However, service providers are allowed and encouraged to supply raw socket support. A Winsock-compliant application that wishes to use raw sockets should attempt to open the socket with the **socket()** call (see section 3.30), and if it fails either attempt to use another socket type or indicate the failure to the user.

B.3.7 Winsock in Multithreaded Versions of Windows

The Winsock interface is designed to work for both single-threaded versions of Windows (such as Windows 3.1) and preemptive multithreaded versions of Windows (such as Windows NT). In a multithreaded environment the sockets interface is basically the same, but the author of a multithreaded application must be aware that it is the responsibility of the application, not Winsock, to synchronize access to a socket between threads. This is the same rule as applies to other forms of I/O such as file I/O. Failure to synchronize calls on a socket leads to unpredictable results; for example if there are two simultaneous calls to **send(),** there is no guarantee as to the order in which the data will be sent.

Closing a socket in one thread that has an outstanding blocking call on the same socket in another thread will cause the blocking call to fail with WSAEINTR, just as if the operation were canceled. This also applies if there is a **select()** call outstanding and the application closes one of the sockets being selected.

There is no default blocking hook (see section 3.1.1 for details) installed in preemptive multithreaded versions of Windows. This is because the machine will not be blocked if a single application is waiting for an operation to complete and hence not calling **PeekMessage()** or **GetMessage()** which cause the application to yield in nonpreemptive Windows. However, for backwards compatibility the **WSASetBlockingHook()** call is implemented in multithreaded versions of Windows, and any application whose behavior depends on the default blocking hook may install their own blocking hook which duplicates the default hook's semantics, if desired.

B.4 Socket Library Overview

B.4.1 Socket Functions

The Winsock specification includes the following Berkeley-style socket routines:

accept() *	An incoming connection is acknowledged and associated with an immediately created socket. The original socket is returned to the listening state.
bind()	Assign a local name to an unnamed socket.
closesocket() *	Remove a socket from the per-process object reference table. Only blocks if SO_LINGER is set.
connect() *	Initiate a connection on the specified socket.
getpeername()	Retrieve the name of the peer connected to the specified socket.
getsockname()	Retrieve the local address to which the specified socket is bound.
getsockopt()	Retrieve options associated with the specified socket.
htonl()	Convert a 32-bit quantity from host byte order to network byte order.
htons()	Convert a 16-bit quantity from host byte order to network byte order.
inet_addr()	Converts a character string representing a number in the Internet standard "." notation to an Internet address value.
inet_ntoa()	Converts an Internet address value to an ASCII string in "." notation i.e. "a.b.c.d".
ioctlsocket()	Provide control for sockets.
listen()	Listen for incoming connections on a specified socket.
ntohl()	Convert a 32-bit quantity from network byte order to host byte order.
ntohs()	Convert a 16-bit quantity from network byte order to host byte order.
recv() *	Receive data from a connected socket.
recvfrom() *	Receive data from either a connected or unconnected socket.
select() *	Perform synchronous I/O multiplexing.
send() *	Send data to a connected socket.
sendto() *	Send data to either a connected or unconnected socket.
setsockopt()	Store options associated with the specified socket.
shutdown()	Shut down part of a full-duplex connection.
socket()	Create an endpoint for communication and return a socket.

* = The routine can block if acting on a blocking socket.

B.4.1.1 Blocking/Non blocking & Data Volatility

One major issue in porting applications from a Berkeley sockets environment to a Windows environment involves "blocking"; that is, invoking a function which does not return until the associated operation is completed. The problem arises when the operation may take an arbitrarily long time to complete: an obvious example is a **recv()** which may block until data has been received from the peer system. The default behavior within the Berkeley sockets model is for a socket to operate in a blocking mode unless the programmer explicitly requests that operations be treated as non-blocking. ***In Win16 environments (which are non-preemptive and have no direct operating system support for blocking operations), it is strongly recommended that programmers use the nonblocking (asynchronous) operations if at all possible. Use***

blocking operations only if absolutely necessary, and carefully read and understand this section if you must use blocking operations. These cautions and restrictions do not apply to Win32 environments which are both multi-threaded and preemptive.

Even on a blocking socket, some operations (e.g. **bind()**, **getsockopt()**, **getpeername()**) can be completed immediately. For such operations there is no difference between blocking and non-blocking operation. Other operations (e.g. **recv()**) may be completed immediately or may take an arbitrary time to complete, depending on various transport conditions. When applied to a blocking socket, these operations are referred to as blocking operations. All routines which can block are listed with an asterisk in the tables above and below.

Within Winsock, a blocking operation which cannot be completed immediately is handled as follows. The service provider initiates the operation, and then enters a loop in which it dispatches any Windows messages (yielding the processor to another thread if necessary) and then checks for the completion of the Winsock function. If the function has completed, or if **WSACancelBlockingCall()** has been invoked, the blocking function completes with an appropriate result. Refer to section 3.62 , **WSASetBlockingHook()**, for a complete description of this mechanism, including pseudocode for the various functions.

If a Windows message is received for a process for which a blocking operation is in progress, there is a risk that the application will attempt to issue another Winsock call. Because of the difficulty of managing this condition safely, the Winsock specification does not support such application behavior. Two functions are provided to assist the programmer in this situation. **WSAIsBlocking()** may be called to determine whether or not a blocking Winsock call is in progress. **WSACancelBlockingCall()** may be called to cancel an in-progress blocking call, if any. Except where otherwise noted, any other Winsock function which is called in this situation will fail with the error WSAEINPROGRESS. It should be emphasized that this restriction applies to both blocking and non-blocking operations.

Although this mechanism is sufficient for simple applications, it cannot support the complex message-dispatching requirements of more advanced applications (for example, those using the MDI model). For such applications, the Winsock API includes the function **WSASetBlockingHook()**, which allows the application to specify a special routine which will be called instead of the default message dispatch routine described above.

The Winsock provider calls the blocking hook only if all of the following are true: the routine is one which is defined as being able to block, the specified socket is a blocking socket, and the request cannot be completed immediately. (A socket is set to blocking by default, but the IOCTL FIONBIO, and **WSAAsyncSelect()** set a socket to nonblocking mode.) If an application uses only non-blocking sockets and uses the **WSAAsyncSelect()** and/or the **WSAAsyncGetXByY()** routines instead of **select()** and the **getXbyY()** routines, then the blocking hook will never be called and the application does not need to be concerned with the reentrancy issues the blocking hook can introduce.

If an application invokes an asynchronous or non-blocking operation which takes a pointer to a memory object (e.g. a buffer, or a global variable) as an argument, it is the responsibility of the application to ensure that the object is available to Winsock throughout the operation. The application must not invoke any Windows function which might affect the mapping or addressability of the memory involved. In a multithreaded system, the application is also responsible for coordinating access to the object using appropriate synchronization mechanisms. Winsock cannot, and will not, address these issues. The possible consequences of failing to observe these rules are beyond the scope of this specification.

B.4.2 Database Functions

The Winsock specification defines the following "database" routines. As noted earlier, a Winsock service provider may choose to implement these in a manner which does not depend on local database files. The pointer returned by certain database routines such as **gethostbyname()** points to a structure which is

allocated by Winsock or its service providers. The data which is pointed to is volatile and is good only until the next Winsock API call from that thread. Additionally, the application must never attempt to modify this structure or to free any of its components. Only one copy of this structure is allocated for a thread, and so the application should copy any information which it needs before issuing any other Winsock API calls.

gethostbyaddr() *	Retrieve the name(s) and address corresponding to a network address.
gethostbyname() *	Retrieve the name(s) and address corresponding to a host name.
gethostname()	Retrieve the name of the local host.
getprotobyname() *	Retrieve the protocol name and number corresponding to a protocol name.
getprotobynumber() *	Retrieve the protocol name and number corresponding to a protocol number.
getservbyname() *	Retrieve the service name and port corresponding to a service name.
getservbyport() *	Retrieve the service name and port corresponding to a port.

*** =** The routine can block under some circumstances.

B.4.3 Microsoft Windows-specific Extension Functions

The Winsock specification provides a number of extensions to the standard set of Berkeley Sockets routines. Principally, these extended functions allow message or function-based, asynchronous access to network events. While use of this extended API set is not mandatory for socket-based programming (with the exception of **WSAStartup()** and **WSACleanup()**), it is recommended for conformance with the Microsoft Windows programming paradigm.

WSAAccept() *	An extended version of accept() which allows for conditional acceptance and socket grouping.
WSAAsyncGetHostByAddr()	A set of functions which provide asynchronous versions of the standard Berkeley **getXbyY()** functions. For example, the **WSAAsyncGetHostByName()** function provides an asynchronous message based implementation of the standard Berkeley **gethostbyname()** function.
WSAAsyncGetHostByName()	
WSAAsyncGetProtoByName()	
WSAAsyncGetProtoByNumber()	
WSAAsyncGetServByName()	
WSAAsyncGetServByPort()	
WSAAsyncSelect()	Perform asynchronous version of **select()**
WSACancelAsyncRequest()	Cancel an outstanding instance of a **WSAAsyncGetXByY()** function.
WSACancelBlockingCall()	Cancel an outstanding "blocking" Winsock call
WSACleanup()	Sign off from the underlying Winsock DLL.
WSACloseEvent()	Destroys an event object.
WSAConnect()	An extended version of connect() which allows for socket grouping and QOS specification.
WSACreateEvent()	Creates an event object.
WSADuplicateSocket()	Allow an underlying socket to be shared by creating a virtual socket.
WSAEnumNetworkEvents()	Discover occurrences of network events.
WSAEnumProtocols()	Retrieve information about each available protocol.
WSAEventSelect()	Associate network events with an event object.
WSAGetLastError()	Obtain details of last Winsock error
WSAGetOverlappedResult()	Get completion status of overlapped operation.
WSAGetQOSByName()	Supply QOS parameters based on a well-known service name.
WSAHtonl()	Extended version of htonl()
WSAHtons()	Extended version of htons()
WSAIsBlocking()	Determine if the underlying Winsock DLL is already blocking an existing call for this thread
WSANtohl()	Extended version of ntohl()
WSANtohs()	Extended version of ntohs()
WSAProviderSelection()	Set the Winsock service provider to be used for upcoming Winsock calls.
WSARecv()	An extended version of recv() which accommodates overlapped sockets and provides the *flags* parameter as IN OUT
WSARecvFrom()	An extended version of recvfrom() which accommodates overlapped sockets and provides the *flags* parameter as IN OUT
WSAResetEvent()	Resets an event object.
WSASend()	An extended version of send() which accommodates overlapped sockets
WSASendTo()	An extended version of sendto() which accommodates overlapped sockets
WSASetBlockingHook()	"Hook" the blocking method used by the underlying Winsock implementation
WSASetEvent()	Sets an event object.
WSASetLastError()	Set the error to be returned by a subsequent **WSAGetLastError()**

WSASocket()	An extended version of socket() which takes a PROTOCOL_INFO struct as input and allows overlapped sockets to be created.
WSAStartup()	Initialize the underlying Winsock DLL.
WSAUnhookBlockingHook()	Restore the original blocking function
WSAWaitForMultipleEvents()	Blocks on multiple event objects.

B.4.3.1 Asynchronous select() Mechanism

The **WSAAsyncSelect()** API allows an application to register an interest in one or many network events. This API is provided to supersede the need to do polled network I/O. Any situation in which **select()** or non-blocking I/O routines (such as **send()** and **recv()**) are either already used or are being considered is usually a candidate for the **WSAAsyncSelect()** API. When declaring interest in such condition(s), you supply a window handle to be used for notification. The corresponding window then receives message-based notification of the conditions in which you declared an interest.

WSAAsyncSelect() allows interest to be declared in the following conditions for a particular socket:

Socket readiness for reading
Socket readiness for writing
Out-of-band data ready for reading
Socket readiness for accepting incoming connection
Completion of non-blocking **connect()**
Connection closure
Change in quality of service parameters

B.4.3.2 Asynchronous Database Routines

The asynchronous "database" functions allow applications to request information in an asynchronous manner. Some network implementations and/or configurations perform network based operations to resolve such requests. The **WSAAsyncGetXByY()** functions allow application developers to request services which would otherwise block the operation of the whole Windows environment if the standard Berkeley function were used. The **WSACancelAsyncRequest()** function allows an application to cancel any outstanding asynchronous request.

B.4.3.3 Hooking Blocking Methods

As noted in section B.4.1.1 above, Winsock implements blocking operations in such a way that Windows message processing can continue, which may result in the application which issued the call receiving a Windows message. In certain situations an application may want to influence or change the way in which this pseudo-blocking process is implemented. The **WSASetBlockingHook()** provides the ability for an application to substitute a named routine to be used when Winsock relinquishes the processor during a "blocking" operation.

B.4.3.4 Error Handling

For compatibility with thread-based environments, details of API errors are obtained through the **WSAGetLastError()** API. Although the accepted "Berkeley-Style" mechanism for obtaining socket-based network errors is via "errno", this mechanism cannot guarantee the integrity of an error ID in a multi-threaded environment. **WSAGetLastError()** allows you to retrieve an error code on a per thread basis.

WSAGetLastError() returns error codes which avoid conflict with standard Microsoft C error codes. Certain error codes returned by certain Winsock routines fall into the standard range of error codes as defined by Microsoft C. If you are NOT using an application development environment which defines error codes consistent with Microsoft C, you are advised to use the Winsock error codes prefixed by "WSA" to ensure accurate error code detection.

Note that this specification defines a recommended set of error codes, and lists the possible errors which may be returned as a result of each function. It may be the case in some implementations that other Winsock error codes will be returned in addition to those listed, and applications should be prepared to handle errors other than those enumerated under each API description. However Winsock will not return any value which is not enumerated in the table of legal Winsock errors given in Appendix A.1.

B.4.3.5 Accessing Winsock DLL from an Intermediate DLL

The Winsock DLL may be accessed both directly from an application and through an "intermediate" DLL. An example of such an intermediate DLL would be a virtual network API layer that supports generalized network functionality for applications and uses Winsock. Such a DLL could be used by several applications simultaneously, and the DLL must take special precautions with respect to the **WSAStartup()** and **WSACleanup()** calls to ensure that these routines are called in the context of each task that will make Winsock calls. This is because the Winsock DLL will need a call to **WSAStartup()** for each task in order to set up task-specific data structures, and a call to **WSACleanup()** to free any resources allocated for the task.

There are (at least) two ways to accomplish this. The simplest method is for the intermediate DLL to have calls similar to **WSAStartup()** and **WSACleanup()** that applications call as appropriate. The DLL would then call **WSAStartup()** or **WSACleanup()** from within these routines. Another mechanism is for the intermediate DLL to build a table of task handles, which are obtained from the **GetCurrentTask()** Windows API, and at each entry point into the intermediate DLL check whether **WSAStartup()** has been called for the current task, then call **WSAStartup()** if necessary.

If a DLL makes a blocking call and does not install its own blocking hook, then the DLL author must be aware that control may be returned to the application either by an application-installed blocking hook or by the default blocking hook. Thus, it is possible that the application will cancel the DLL's blocking operation via **WSACancelBlockingCall()**. If this occurs, the DLL's blocking operation will fail with the error code WSAEINTR, and the DLL must return control to the calling task as quickly as possible, as the user has likely pressed a cancel or close button and the task has requested control of the CPU. It is recommended that DLLs which make blocking calls install their own blocking hooks with **WSASetBlockingHook()** to prevent unforeseen interactions between the application and the DLL.

Note that this is not necessary for DLLs in Windows NT because of its different process and DLL structure. Under Windows NT, the intermediate DLL could simply call **WSAStartup()** in its DLL initialization routine, which is called whenever a new process which uses the DLL starts.

Appendix C. Open Issues

The following issues are not yet resolved (or sometimes simply not addressed yet due to lack of time)

1. Make sure that 16 <-> 32 bit conversion of error codes happens correctly.

2. Resolve concerns regarding availability of source code for Winsock 2 DLL's, and policies for how and when bugs will be fixed.

3. Specify the debugging support that will be built into Winsock.DLL

4. The QOS section needs to be revised and updated.

5. Is there a need for a generic, asynchrouous (overlapped) ioctl/setsockopt function?

6. Define the notification mechansim for shared sockets. Proposals include:
   - totally independent notification on each descriptor
   - each event may be sent to one and only one descriptor

7. If an app asks for connect data but the protocol does not support it, does the request fail or does the connect data get ignored? (general concensus was to fail)

8. Is the output flags param of WSARecv used to indicate MSG_OOB as well as MSG_PARTIAL?

9. Should winsock2 define a generic configuration mechanism?

10. How does the user or an application determine the order in which providers are considered when Winsock.DLL must select a provider on the basis of a tripple (address family, socket type, protocol)? Is there a control panel applet or somesuch that controls this?

11. How will scatter/gather semantics be supported?