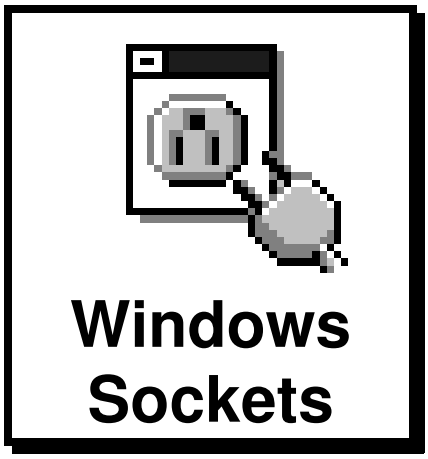

Windows Sockets

An Open Interface for Network Programming under Microsoft® Windows™

Version 1.1

20 January 1993

Martin Hall
Mark Towfiq
Geoff Arnold
David Treadwell
Henry Sanders

Copyright © 1992 by Martin Hall, Mark Towfiq
Geoff Arnold, David Treadwell and Henry Sanders

All rights reserved.

This document may be freely redistributed in any form, electronic or otherwise, provided that it is distributed in its entirety and that the copyright and this notice are included. Comments or questions may be submitted via electronic mail to **winsock@microdyne.com**. Requests to be added to the Windows Sockets mailing list should be addressed to **winsock-request@microdyne.com**. This specification, archives of the mailing list, and other information on Windows Sockets are available via anonymous FTP from the host microdyne.com, directory /pub/winsock. Questions about products conforming to this specification should be addressed to the vendors of the products.

Portions of the Windows Sockets specification are derived from material which is Copyright (c) 1982-1986 by the Regents of the University of California. All rights are reserved. The Berkeley Software License Agreement specifies the terms and conditions for redistribution.

Revision history:

1.0 Rev.A	June 11, 1992
1.0 Rev.B	June 16, 1992
1.0 Rev. C	October 12, 1992
1.1	January, 1993

Windows Sockets
Version 1.1

TABLE OF CONTENTS

TABLE OF CONTENTS iii
ACKNOWLEDGMENTS vi
1. INTRODUCTION 1
    1.1 What is Windows Sockets? 1
    1.2 Berkeley Sockets 1
    1.3 Microsoft Windows and Windows-specific extensions 1
    1.4 The Status of this Specification 2
    1.5 Revision History 2
        1.5.1 Windows Sockets Version 1.0 2
        1.5.2 Windows Sockets Version 1.1 2
2. PROGRAMMING WITH SOCKETS 4
    2.1 Windows Sockets Stack Installation Checking 4
    2.2 Sockets 4
        2.2.1 Basic concepts 4
        2.2.2 Client-server model 4
        2.2.3 Out-of-band data 5
        2.2.4 Broadcasting 5
    2.3 Byte Ordering 6
    2.4 Socket Options 6
    2.5 Database Files 7
    2.6 Deviation from Berkeley Sockets 7
        2.6.1 socket data type and error values 8
        2.6.2 select() and FD_* 8
        2.6.3 Error codes - errno, h_errno & WSAGetLastError() 8
        2.6.4 Pointers 9
        2.6.5 Renamed functions 9
            2.6.5.1 close() & closesocket() 9
            2.6.5.1 ioctl() & ioctlsocket() 9
        2.6.6 Blocking routines & EINPROGRESS 9
        2.6.7 Maximum number of sockets supported 9
        2.6.8 Include files 10
        2.6.9 Return values on API failure 10
        2.6.10 Raw Sockets 10
    2.7 Windows Sockets in Multithreaded Versions of Windows 10
3. SOCKET LIBRARY OVERVIEW 12
    3.1 Socket Functions 12
        3.1.1 Blocking/Non blocking & Data Volatility 12
    3.2 Database Functions 13
    3.3 Microsoft Windows-specific Extension Functions 14
        3.3.1 Asynchronous select() Mechanism 15
        3.3.2 Asynchronous Support Routines 15
        3.3.3 Hooking Blocking Methods 15
        3.3.4 Error Handling 16
        3.3.5 Accessing a Windows Sockets DLL from an Intermediate DLL 16
        3.3.6 Internal use of Messages by Windows Sockets Implementations 16
        3.3.7 Private API Interfaces 17
4. SOCKET LIBRARY REFERENCE 18

4.1 Socket Routines 18
4.1.1 accept() 19
4.1.2 bind() 21
4.1.3 closesocket() 23
4.1.4 connect() 25
4.1.5 getpeername() 27
4.1.6 getsockname() 28
4.1.7 getsockopt() 29
4.1.8 htonl() 31
4.1.9 htons() 32
4.1.10 inet_addr() 33
4.1.11 inet_ntoa() 34
4.1.12 ioctlsocket() 35
4.1.13 listen() 37
4.1.14 ntohl() 39
4.1.15 ntohs() 40
4.1.16 recv() 41
4.1.17 recvfrom() 43
4.1.18 select() 46
4.1.19 send() 48
4.1.20 sendto() 50
4.1.21 setsockopt() 53
4.1.22 shutdown() 56
4.1.23 socket() 58
4.2 Database Routines 60
4.2.1 gethostbyaddr() 60
4.2.2 gethostbyname() 62
4.2.3 gethostname() 63
4.2.4 getprotobyname() 64
4.2.5 getprotobynumber() 66
4.2.6 getservbyname() 67
4.2.7 getservbyport() 69
4.3 Microsoft Windows-specific Extensions 70
4.3.1 WSAAsyncGetHostByAddr() 70
4.3.2 WSAAsyncGetHostByName() 73
4.3.3 WSAAsyncGetProtoByName() 76
4.3.4 WSAAsyncGetProtoByNumber() 79
4.3.5 WSAAsyncGetServByName() 82
4.3.6 WSAAsyncGetServByPort() 85
4.3.7 WSAAsyncSelect() 88
4.3.8 WSACancelAsyncRequest() 94
4.3.9 WSACancelBlockingCall() 96
4.3.10 WSACleanup() 98
4.3.11 WSAGetLastError() 100
4.3.12 WSAIsBlocking() 101
4.3.13 WSASetBlockingHook() 102
4.3.14 WSASetLastError() 104
4.3.15 WSAStartup() 105
4.3.16 WSAUnhookBlockingHook() 109
Appendix A. Error Codes and Header Files 110
A.1 Error Codes 110
A.2 Header Files 112
A.2.1 Berkeley Header Files 112
A.2.2 Windows Sockets Header File - winsock.h 113

Appendix B. Notes for Windows Sockets Suppliers 125
B.1 Introduction 125
B.2 Windows Sockets Components 125
B.2.1 Development Components 125
B.2.2 Run Time Components 125
B.3 Multithreadedness and blocking routines. 125
B.4 Database Files 126
B.5 FD_ISSET 126
B.6 Error Codes 126
B.7 DLL Ordinal Numbers 126
B.8 Validation Suite 127
Appendix C. For Further Reference 129
Appendix D. Background Information 130
D.1 Legal Status of Windows Sockets 130
D.2 The Story Behind the Windows Sockets Icon 130

ACKNOWLEDGMENTS

The authors would like to thank their companies for allowing them the time and resources to make this specification possible: JSB Corporation, Microdyne Corporation, FTP Software, Sun Microsystems, and Microsoft Corporation.

Special thanks should also be extended to the other efforts contributing to the success of Windows Sockets. The original draft was heavily influenced by existing specifications offered and detailed by JSB Corporation and Net Manage, Inc. The "version 1.0 debate" hosted by Microsoft in Seattle allowed many of the members of the working group to hash out final details for 1.0 vis-a-vis.

Sun Microsystems was kind enough to allow first time implementors to "plug and play" beta software during the first Windows Sock-A-Thon of Windows Sockets applications and implementations at Interop Fall '92. Microsoft has shared WSAT (the Windows Sockets API Tester) with other Windows Sockets implementors as a standard Windows Sockets test suite to aid in testing their implementations. Finally, Sun Microsystems and FTP Software plan to host the Windows Sock-A-Thon II in Boston February '93.

Without the contributions of the individuals and corporations involved in the working group, Windows Sockets would never have been as thoroughly reviewed and completed as quickly. In just one year, several competitors in the networking business developed a useful specification with something to show for it! Many thanks to all which participated, either in person or on e-mail to the Windows Sockets effort. The authors would like to thank everyone who participated in any way, and apologize in advance for anyone we have omitted.

List of contributors:

Martin Hall	(Chairman)	JSB Corporation	martinh@jsbus.com
Mark Towfiq	(Coordinator)	Microdyne Corporation	towfiq@microdyne.com
Geoff Arnold	(Editor 1.0)	Sun Microsystems	geoff@east.sun.com
David Treadwell	(Editor 1.1)	Microsoft Corporation	davidtr@microsoft.com
Henry Sanders		Microsoft Corporation	henrysa@microsoft.com
J. Allard		Microsoft Corporation	jallard@microsoft.com
Chris Arap-Bologna		Distinct	chris@distinct.com
Larry Backman		FTP Software	backman@ftp.com
Alistair Banks		Microsoft Corporation	alistair@microsoft.com
Rob Barrow		JSB Corporation	robb@jsb.co.uk
Carl Beame		Beame & Whiteside	beame@mcmaster,ca
Dave Beaver		Microsoft Corporation	dbeaver@microsoft.com
Amatzia BenArtzi		NetManage, Inc.	amatzia@netmanage.com
Mark Beyer		Ungermann-Bass	mbeyer@ub.com
Nelson Bolyard		Silicon Graphics, Inc.	nelson@sgi.com
Pat Bonner		Hewlett-Packard	p_bonner@cnd.hp.com
Derek Brown		FTP Software	db@wco.ftp.com
Malcolm Butler		ICL	mcab@oasis.icl.co.uk
Mike Calbaum		Fronteir Technologies	mike@frontiertech.com
Isaac Chan		Microsoft Corporation	isaacc@microsoft.com
Khoji Darbani		Informix	khoji@informix.com
Nestor Fesas		Hughes LAN Systems	nestor@hls.com
Karanja Gakio		FTP Software	karanja@ftp.com
Vikas Garg		Distinct	vikas@distinct.com
Gary Gere		Gupta	ggere@gupta.com

Jim Gilroy	Microsoft Corporation	jamesg@microsoft.com
Bill Hayes	Hewlett-Packard	billh@hpchdpc.cnd.hp.com
Paul Hill	MIT	pbh@athena.mit.edu
Tmima Koren	Net Manage, Inc.	tmima@netmanage.com
Hoek Law	Citicorp	law@dcc.tti.com
Graeme Le Roux	Moresdawn P/L	-
Kevin Lewis	Novell	kevinl@novell.com
Roger Lin	3Com	roger_lin@3mail.3com.com
Terry Lister	Hewlett-Packard	tel@cnd.hp.com
Jeng Long Jiang	Wollongong	long@twg.com
Lee Murach	Network Research	lee@nrc.com
Pete Ostenson	Microsoft Corporation	peteo@microsoft.com
David Pool	Spry, Inc.	dave@spry.com
Bob Quinn	FTP Software	rcq@ftp.com
Glenn Reitsma	Hughes LAN Systems	glennr@hls.com
Brad Rice	Age	rice@age.com
Allen Rochkind	3Com	-
Jonathan Rosen	IBM	jrosen@vnet.ibm.com
Steve Stokes	Novell	stoke@novell.com
Joseph Tsai	3Com	joe_tsai@3mail.3com.com
James Van Bokkelen	FTP Software	jbvb@ftp.com
Miles Wu	Wollongong	wu@twg.com
Boris Yanovsky	NetManage, Inc.	boris@netmanage.com

1. INTRODUCTION

1.1 What is Windows Sockets?

The Windows Sockets specification defines a network programming interface for Microsoft Windows[1] which is based on the "socket" paradigm popularized in the Berkeley Software Distribution (BSD) from the University of California at Berkeley. It encompasses both familiar Berkeley socket style routines and a set of Windows-specific extensions designed to allow the programmer to take advantage of the message-driven nature of Windows.

The Windows Sockets Specification is intended to provide a single API to which application developers can program and multiple network software vendors can conform. Furthermore, in the context of a particular version of Microsoft Windows, it defines a binary interface (ABI) such that an application written to the Windows Sockets API can work with a conformant protocol implementation from any network software vendor. This specification thus defines the library calls and associated semantics to which an application developer can program and which a network software vendor can implement.

Network software which conforms to this Windows Sockets specification will be considered "Windows Sockets Compliant". Suppliers of interfaces which are "Windows Sockets Compliant" shall be referred to as "Windows Sockets Suppliers". To be Windows Sockets Compliant, a vendor must implement 100% of this Windows Sockets specification.

Applications which are capable of operating with any "Windows Sockets Compliant" protocol implementation will be considered as having a "Windows Sockets Interface" and will be referred to as "Windows Sockets Applications".

This version of the Windows Sockets specification defines and documents the use of the API in conjunction with the Internet Protocol Suite (IPS, generally referred to as TCP/IP). Specifically, all Windows Sockets implementations support both stream (TCP) and datagram (UDP) sockets.

While the use of this API with alternative protocol stacks is not precluded (and is expected to be the subject of future revisions of the specification), such usage is beyond the scope of this version of the specification.

1.2 Berkeley Sockets

The Windows Sockets Specification has been built upon 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[2] and other environments, and to simplify the task of porting existing sockets-based source code. The Windows Sockets API is consistent with release 4.3 of the Berkeley Software Distribution (4.3BSD).

Portions of the Windows Sockets specification are derived from material which is Copyright (c) 1982-1986 by the Regents of the University of California. All rights are reserved. The Berkeley Software License Agreement specifies the terms and conditions for redistribution.

1.3 Microsoft Windows and Windows-specific extensions

---

[1] Windows is a trademark of Microsoft Corporation.
[2] UNIX is a trademark of Unix System Laboratories, Inc.

This 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 Windows Sockets implementations and Windows Sockets applications in both 16 and 32 bit operating environments.

Windows Sockets 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.

1.4 The Status of this Specification

Windows Sockets is an independent specification which was created and exists for the benefit of application developers and network vendors and, indirectly, computer users. Each published (non-draft) version of this specification represents a fully workable API for implementation by network vendors and programming use by application developers. Discussion of this specification and suggested improvements continue and are welcomed. Such discussion occurs mainly via the Internet electronic mail forum winsock@microdyne.com. Meetings of interested parties occur on an irregular basis. Details of these meetings are publicized to the electronic mail forum.

1.5 Revision History

1.5.1 Windows Sockets Version 1.0

Windows Sockets Version 1.0 represented the results of considerable work within the vendor and user community as discussed in Appendix C. This version of the specification was released in order that network software suppliers and application developers could begin to construct implementations and applications which conformed to the Windows Sockets standard.

1.5.2 Windows Sockets Version 1.1

Windows Sockets Version 1.1 follows the guidelines and structure laid out by version 1.0, making changes only where absolutely necessary as indicated by the experiences of a number of companies that created Windows Sockets implementations based on the version 1.0 specification. Version 1.1 contains several clarifications and minor fixes to version 1.0. Additionally, the following more significant changes were incorporated into version 1.1:

o Inclusion of the **gethostname()** routine to simplify retrieval of the host's name and address.

o Definition of DLL ordinal values below 1000 as reserved for Windows Sockets and ordinals above 1000 as unrestricted. This allows Windows Sockets vendors to include private interfaces to their DLLs without risking that the ordinals chosen will conflict with a future version of Windows Sockets.

o Addition of a reference count to **WSAStartup()** and **WSACleanup()**, requiring correspondences between the calls. This allows applications and third-party DLLs to make use of a Windows Sockets implementation without being concerned about the calls to these APIs made by the other.

o Change of return type of **inet_addr()** from **struct in_addr** to **unsigned long**. This was required due to different handling of four-byte structure returns between the Microsoft and Borland C compilers.

o Change of **WSAAsyncSelect()** semantics from "edge-triggered" to "level-triggered". The level-triggered semantics significantly simplify an application's use of this routine.

o Change the **ioctlsocket()** FIONBIO semantics to fail if a **WSAAsyncSelect()** call is outstanding on the socket.

o Addition of the TCP_NODELAY socket option for RFC 1122 conformance.

All changes between the 1.0 and 1.1 specifications are flagged with change bars at the left of the page.

2. PROGRAMMING WITH SOCKETS

2.1 Windows Sockets Stack Installation Checking

To detect the presence of one (or many) Windows Sockets implementations on a system, an application which has been linked with the Windows Sockets 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 Windows Sockets implementations (WINSOCK.DLL). For each instance it can issue a **LoadLibrary()** call and use the **WSAStartup()** routine to discover implementation specific data.

This version of the Windows Sockets specification does not attempt to address explicitly the issue of multiple concurrent Windows Sockets implementations. Nothing in the specification should be interpreted as restricting multiple Windows Sockets DLLs from being present and used concurrently by one or more Windows Sockets applications.

For further details of where to obtain Windows Sockets components, see Appendix B.2.

2.2 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.

2.2.1 Basic concepts

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 in the same domain (it may be possible to cross domain boundaries, but only if some translation process is performed). The Windows Sockets facilities support a single communication domain: the Internet domain, which is used by processes which communicate using the Internet Protocol Suite. (Future versions of this specification may include additional domains.)

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.

Two types of sockets currently are available to a user. A stream socket provides for the bi-directional, reliable, sequenced, and unduplicated flow of data without record boundaries.

A datagram socket supports bi-directional flow of data which is not promised to be sequenced, reliable, or unduplicated. That is, a process receiving messages on a datagram socket may find messages duplicated, and, possibly, in an order different from the order in which it was sent. An important characteristic of a datagram socket is that record boundaries in data are preserved. Datagram sockets closely model the facilities found in many contemporary packet switched networks such as Ethernet.

2.2.2 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-based services are the norm, some services are based on the use of datagram sockets.

2.2.3 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. Windows Sockets 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.

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 4.1.21, **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 Windows Sockets implementation will maintain a logical "mark" in the data stream indicate the point at which the out-of-band data was sent. An application can use the SIOCATMARK **ioctlsocket()** command (see section 4.1.12) 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()** routine is particularly well suited to handling notification of the presence of out-of-band-data.

2.2.4 Broadcasting

By using a datagram socket, it is possible to send broadcast packets on many networks supported by the system. The network itself must support broadcast: the system 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: it is desired to find a resource on a local network without prior knowledge of its address, or 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, 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 Windows Sockets specification does not provide any mechanism whereby an application can determine the type of underlying network, nor any way to control the semantics of broadcasting.

2.3 Byte Ordering

The Intel byte ordering is like that of the DEC VAX[3], and therefore differs from the Internet and 68000[4]-type processor byte ordering. Thus care must be taken to ensure correct orientation.

Any reference to IP addresses or port numbers passed to or from a Windows Sockets routine must be in network order. 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 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 the Windows Sockets API rather than writing their own conversion code, since future implementations of Windows Sockets are likely to run on systems for which the host order is identical to the network byte order. Only applications which use the standard conversion functions are likely to be portable.

2.4 Socket Options

The socket options supported by Windows Sockets are listed in the pages describing **setsockopt()** and **getsockopt()**. A Windows Sockets implementation must recognize all of these options, and (for **getsockopt()**) return plausible values for each. The default value for each option is shown in the following table.

---

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

Value	Type	Meaning	Default	Note
SO_ACCEPTCONN	BOOL	Socket is **listen()**ing.	FALSE unless a **listen()** has been performed	
SO_BROADCAST	BOOL	Socket is configured for the transmission of broadcast messages.	FALSE	
SO_DEBUG	BOOL	Debugging is enabled.	FALSE	(i)
SO_DONTLINGER	BOOL	If true, the SO_LINGER option is disabled.	TRUE	
SO_DONTROUTE	BOOL	Routing is disabled.	FALSE	(i)
SO_ERROR	int	Retrieve error status and clear.	0	
SO_KEEPALIVE	BOOL	Keepalives are being sent.	FALSE	
SO_LINGER	struct linger FAR *	Returns the current linger options.	*l_onoff* is 0	
SO_OOBINLINE	BOOL	Out-of-band data is being received in the normal data stream.	FALSE	
SO_RCVBUF	int	Buffer size for receives	Implementation dependent	(i)
SO_REUSEADDR	BOOL	The address to which this socket is bound can be used by others.	FALSE	
SO_SNDBUF	int	Buffer size for sends	Implementation dependent	(i)
SO_TYPE	int	The type of the socket (e.g. SOCK_STREAM).	As created via **socket()**	
TCP_NODELAY	BOOL	Disables the Nagle algorithm for send coalescing.	Implementation dependent	

Notes:

(i) An implementation may silently ignore this option on **setsockopt()** and return a constant value for **getsockopt()**, or it may accept a value for **setsockopt()** and return the corresponding value in **getsockopt()** without using the value in any way.

2.5 Database Files

The **getXbyY()**[5] 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 Windows Sockets implementation in different ways, a Windows Sockets application requests such information in a consistent manner through either the **getXbyY()** or the **WSAAsyncGetXByY()** class of routines.

2.6 Deviation from Berkeley Sockets

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

---

[5] This specification uses the function name **getXbyY()** to represent the set of routines **gethostbyaddr()**, **gethostbyname()**, etc. Similarly **WSAAsyncGetXByY()** represents **WSAAsyncGetHostByAddr()**, etc.

2.6.1 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 Windows Sockets specification, such as being able to use sockets as file handles in Windows NT[6]. 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. Windows Sockets handles have no restrictions, other than that the value INVALID_SOCKET is not a valid socket. Socket handles may take any value in the range 0 to INVALID_SOCKET-1.

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

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

TYPICAL BSD STYLE:

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

PREFERRED STYLE:

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

2.6.2 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 the Windows Sockets API. 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.

2.6.3 Error codes - errno, h_errno & WSAGetLastError()

Error codes set by the Windows Sockets implementation 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 4.3.11. This function is provided in Windows Sockets as a precursor (and eventually an alias) for the Win32 function **GetLastError()**. This is intended to provide a reliable way for a thread in a multi-threaded process to obtain per-thread error information.

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

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

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

---

[6] NT and Windows/NT are trademarks of Microsoft Corporation.

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)
      {...}
```

2.6.4 Pointers

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

2.6.5 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.

2.6.5.1 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 Windows Sockets API prevents an implementation from using regular file handles to identify sockets, nothing requires it either. Socket descriptors are not presumed to correspond to regular file handles, and file operations such as **read()**, **write()**, and **close()** cannot be assumed to 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.

2.6.5.1 ioctl() & ioctlsocket()

Various C language run-time systems use the **ioctl()** routine for purposes unrelated to Windows Sockets. 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()**.

2.6.6 Blocking routines & EINPROGRESS

Although blocking operations on sockets are supported under Windows Sockets, 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 Windows Sockets. See section 3.1.1 for more details.

2.6.7 Maximum number of sockets supported

The maximum number of sockets supported by a particular Windows Sockets supplier 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 4.3.15, **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 Windows Sockets application can make use of is determined at compile time by the manifest constant FD_SETSIZE. This value is used in constructing the fd_set structures used in **select()** (see section 4.1.18). 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 Windows Sockets implementation.

2.6.8 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 Windows Sockets application source files should simply include **winsock.h**.

2.6.9 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)
      {...}
```

2.6.10 Raw Sockets

The Windows Sockets specification does not mandate that a Windows Sockets DLL support raw sockets, that is, sockets opened with SOCK_RAW. However, a Windows Sockets DLL is allowed and encouraged to supply raw socket support. A Windows Sockets-compliant application that wishes to use raw sockets should attempt to open the socket with the **socket()** call (see section 4.1.23), and if it fails either attempt to use another socket type or indicate the failure to the user.

2.7 Windows Sockets in Multithreaded Versions of Windows

The Windows Sockets 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 the Windows Sockets implementation, 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 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 nonpremptive 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.

3. SOCKET LIBRARY OVERVIEW

3.1 Socket Functions

The Windows Sockets 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 current name for the specified socket
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.

3.1.1 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. ***It is strongly recommended that programmers use the nonblocking (asynchronous) operations if at all possible, as they work significantly better within the nonpreemptive Windows environment. Use blocking***

operations only if absolutely necessary, and carefully read and understand this section if you must use blocking operations.

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 a Windows Sockets implementation, a blocking operation which cannot be completed immediately is handled as follows. The DLL 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 Windows Sockets function. If the function has completed, or if **WSACancelBlockingCall()** has been invoked, the blocking function completes with an appropriate result. Refer to section 4.3.13, **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 Windows Sockets call. Because of the difficulty of managing this condition safely, the Windows Sockets 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 Windows Sockets call is in progress. **WSACancelBlockingCall()** may be called to cancel an in-progress blocking call, if any. Any other Windows Sockets 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 Windows Sockets API includes the function **WSASetBlockingHook()**, which allows the programmer to define a special routine which will be called instead of the default message dispatch routine described above.

The Windows Sockets DLL 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()** both 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 the Windows Sockets implementation 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. A Windows Sockets implementation cannot, and will not, address these issues. The possible consequences of failing to observe these rules are beyond the scope of this specification.

3.2 Database Functions

The Windows Sockets specification defines the following "database" routines. As noted earlier, a Windows Sockets supplier 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 the Windows Sockets library. The data which is pointed to is volatile and is good only until the next Windows Sockets 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 Windows Sockets 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.

3.3 Microsoft Windows-specific Extension Functions

The Windows Sockets specification provides a number of extensions to the standard set of Berkeley Sockets routines. Principally, these extended APIs allow message-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.

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" API call
WSACleanup()	Sign off from the underlying Windows Sockets DLL.
WSAGetLastError()	Obtain details of last Windows Sockets API error
WSAIsBlocking()	Determine if the underlying Windows Sockets DLL is already blocking an existing call for this thread
WSASetBlockingHook()	"Hook" the blocking method used by the underlying Windows Sockets implementation
WSASetLastError()	Set the error to be returned by a subsequent **WSAGetLastError()**
WSAStartup()	Initialize the underlying Windows Sockets DLL.
WSAUnhookBlockingHook()	Restore the original blocking function

3.3.1 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

3.3.2 Asynchronous Support 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.

3.3.3 Hooking Blocking Methods

As noted in section 3.1.1 above, Windows Sockets 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 to substitute a named routine which the Windows Sockets implementation is to use when relinquishing the processor during a "blocking" operation.

3.3.4 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 Windows Sockets 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 Windows Sockets 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 Windows Sockets error codes will be returned in addition to those listed, and applications should be prepared to handle errors other than those enumerated under each API description. However a Windows Sockets implementation must not return any value which is not enumerated in the table of legal Windows Sockets errors given in Appendix A.1.

3.3.5 Accessing a Windows Sockets DLL from an Intermediate DLL

A Windows Sockets 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 Windows Sockets. 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 Windows Sockets calls. This is because the Windows Sockets 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 used 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.

3.3.6 Internal use of Messages by Windows Sockets Implementations

In order to implement Windows Sockets purely as a DLL, it may be necessary for the DLL to post messages internally for communication and timing. This is perfectly legal; however, a Windows Sockets DLL must not post messages to a window handle opened by a client application except for those messages requested by the application. A Windows Sockets DLL that needs to use messages for its own purposes must open a hidden window and post any necessary messages to the handle for that window.

3.3.7 Private API Interfaces

The winsock.def file in Appendix B.7 lists the ordinals defined for the Windows Sockets APIs. In addition to the ordinal values listed, all ordinals 999 and below are reserved for future Windows Sockets use. It may be convenient for a Windows Sockets implementation to export additional, private interfaces from the Windows Sockets DLL. This is perfectly acceptable, as long as the ordinals for these exports are above 1000. Note that any application that uses a particular Windows Sockets DLL's private APIs will most likely not work on any other vendor's Windows Sockets implementation. Only the APIs defined in this document are guaranteed to be present in every Windows Sockets implementation.

If an application uses private interfaces of a particular vendor's Windows Sockets DLL, it is recommended that the DLL not be statically linked with the application but rather dynamically loaded with the Windows routines **LoadLibrary()** and **GetProcAddress().** This allows the application to give an informative error message if it is run on a system with a Windows Sockets DLL that does not support the same set of extended functionality.

4. SOCKET LIBRARY REFERENCE

4.1 Socket Routines

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.

4.1.1 accept()

Description Accept a connection on a socket.

#include <winsock.h>

SOCKET PASCAL FAR 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-based 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 packet. 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 Windows Sockets implementation has detected that the network subsystem has failed.
WSAEFAULT	The *addrlen* argument is too small (less than the sizeof a struct sockaddr).

WSAEINTR	The (blocking) call was canceled via **WSACancelBlockingCall().**
WSAEINPROGRESS	A blocking Windows Sockets call is in progress.
WSAEINVAL	**listen()** was not invoked prior to **accept()**.
WSAEMFILE	The queue is 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**()

4.1.2 bind()

Description Associate a local address with a socket.

#include <winsock.h>

int PASCAL FAR 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 datagram or stream 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 (host address/port number) of the socket by assigning a local name to an unnamed socket.

In the Internet address family, a name consists of several components. For SOCK_DGRAM and SOCK_STREAM, the name consists of three parts: a host address, the protocol number (set implicitly to UDP or TCP, respectively), and a port number which identifies the application. If an application does not care what address is assigned to it, it may specify an Internet address equal to INADDR_ANY, a port equal to 0, or both. If the Internet address is equal to INADDR_ANY, any appropriate network interface will be used; this simplifies application programming in the presence of multi-homed hosts. If the port is specified as 0, the Windows Sockets implementation 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 fill in the Internet address until the socket is connected, since several Internet addresses may be valid if the host is multi-homed.

If an application desires to bind to an arbitrary port outside of the range 1024 to 5000, such as the case of rsh which must bind to any reserved port, code similar to the following may be used:

```
SOCKADDR_IN sin;
SOCKET s;
u_short alport = IPPORT_RESERVED;

sin.sin_family = AF_INET;
sin.sin_addr.s_addr = 0;
for (;;) {
    sin.sin_port = htons(alport);
    if (bind(s, (LPSOCKADDR)&sin, sizeof (sin)) == 0) {
        /* it worked */
```

```
            }
            if ( GetLastError() != WSAEADDRINUSE) {
                /* fail */
            }
            alport--;
            if (alport == IPPORT_RESERVED/2 ) {
                /* fail--all unassigned reserved ports are */
                /* in use. */
            }
        }
```

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 Windows Sockets implementation has detected that 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 (less than the size of a struct sockaddr).
WSAEINPROGRESS	A blocking Windows Sockets call is in progress.
WSAEAFNOSUPPORT	The specified address family is not supported by this protocol.
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()**.

4.1.3 closesocket()

Description Close a socket.

#include <winsock.h>

int PASCAL FAR 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 the 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 2.4, 4.1.7 and 4.1.21) 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 2.4, 4.1.7 and 4.1.21), 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 Windows Sockets implementation 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 Windows Sockets implementation has detected that the network subsystem has failed.

WSAENOTSOCK	The descriptor is not a socket.
WSAEINPROGRESS	A blocking Windows Sockets call is in progress.
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**().

4.1.4 connect()

Description Establish a connection to a peer.

#include <winsock.h>

int PASCAL FAR 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 foreign association. The parameter *s* specifies an unconnected datagram or stream socket If the socket 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.

For stream sockets (type SOCK_STREAM), an active connection is initiated to the foreign host using *name* (an address in the name space of the socket). When the socket call completes successfully, the socket is ready to send/receive data.

For a datagram socket (type SOCK_DGRAM), a default destination is set, which will be used on subsequent **send**() and **recv**() calls.

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 the message-based **WSAAsyncSelect()** to indicate interest in connection events, then your application will receive an FD_CONNECT message when the connect operation is complete.

Error Codes

WSANOTINITIALISED	A successful **WSAStartup()** must occur before using this API.
WSAENETDOWN	The Windows Sockets implementation has detected that the network subsystem has failed.
WSAEADDRINUSE	The specified address is already in use.

WSAEINTR	The (blocking) call was canceled via **WSACancelBlockingCall().**
WSAEINPROGRESS	A blocking Windows Sockets call is in progress.
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.
WSAEINVAL	The socket is not already bound to an address.
WSAEISCONN	The socket is already connected.
WSAEMFILE	No more file 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() and WSAAsyncSelect().**

4.1.5 getpeername()

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

#include <winsock.h>

int PASCAL FAR 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 datagram or stream 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 Windows Sockets implementation has detected that the network subsystem has failed.
WSAEFAULT	The namelen argument is not large enough.
WSAEINPROGRESS	A blocking Windows Sockets call is in progress.
WSAENOTCONN	The socket is not connected.
WSAENOTSOCK	The descriptor is not a socket.

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

4.1.6 getsockname()

Description Get the local name for a socket.

#include <winsock.h>

int PASCAL FAR 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 INADDR_ANY, indicating that any of the host's IP addresses should be used for the socket, **getsockname()** will not necessarily return information about the host IP address, unless the socket has been connected with **connect()** or **accept().** A Windows Sockets application must not assume that the IP address will be changed from INADDR_ANY unless the socket is connected. This is because for a multi-homed host the IP 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 Windows Sockets implementation has detected that the network subsystem has failed.
WSAEFAULT	The *namelen* argument is not large enough.
WSAEINPROGRESS	A blocking Windows Sockets operation is in progress.
WSAENOTSOCK	The descriptor is not a socket.
WSAEINVAL	The socket has not been bound to an address with **bind().**

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

4.1.7 getsockopt()

Description Retrieve a socket option.

#include <winsock.h>

int PASCAL FAR 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 IPPROTO_TCP.
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 whether an operation blocks or not, 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 all other options it will be the size of an integer.

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*. The TCP_NODELAY option uses *level* IPPROTO_TCP; all other options use *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_KEEPALIVE	BOOL	Keepalives are being sent.
SO_LINGER	struct linger FAR *	Returns the current linger options.
SO_OOBINLINE	BOOL	Out-of-band data is being received in the normal data stream.
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).
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()**.

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 Windows Sockets implementation has detected that the network subsystem has failed.
WSAEFAULT	The *optlen* argument was invalid.
WSAEINPROGRESS	A blocking Windows Sockets operation is in progress.
WSAENOPROTOOPT	The option is unknown or unsupported. In particular, SO_BROADCAST is not supported on sockets of type SOCK_STREAM, while SO_ACCEPTCONN, SO_DONTLINGER, SO_KEEPALIVE, SO_LINGER and SO_OOBINLINE are not supported on sockets of type SOCK_DGRAM.
WSAENOTSOCK	The descriptor is not a socket.

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

4.1.8 htonl()

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

```
#include <winsock.h>

u_long PASCAL FAR 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()**.

4.1.9 htons()

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

#include <winsock.h>

u_short PASCAL FAR 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**().

4.1.10 inet_addr()

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

#include <winsock.h>

unsigned long PASCAL FAR 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()**

4.1.11 inet_ntoa()

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

#include <winsock.h>

char FAR * PASCAL FAR 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 the Windows Sockets implementation. The application should not make any assumptions about the way in which the memory is allocated. The data is guaranteed to be valid until the next Windows Sockets 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 Windows Sockets call is made.

See Also **inet_addr()**.

4.1.12 ioctlsocket()

Description Control the mode of a socket.

#include <winsock.h>

int PASCAL FAR 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 the 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()** routine automatically sets a socket to nonblocking mode. If **WSAAsyncSelect()** 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.
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 of 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 of type SOCK_DGRAM, FIONREAD returns the size of the first datagram queued on the socket.
SIOCATMARK	Determine whether or not all out-of-band data has been read. This applies only to a socket of 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 Windows Sockets implementation has detected that 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 Windows Sockets operation is in progress.
WSAENOTSOCK	The descriptor *s* is not a socket.

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

4.1.13 listen()

Description Establish a socket to listen for incoming connection.

#include <winsock.h>

int PASCAL FAR 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.

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 support connections, i.e. 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.

listen() attempts to continue to function rationally when there are no available descriptors. It will accept connections until the queue is emptied. If descriptors become available, a later call to **listen**() or **accept**() will re-fill the queue to the current or most recent "backlog", if possible, and resume listening for incoming connections.

Compatibility *backlog* is currently limited (silently) to 5. As in 4.3BSD, illegal values (less than 1 or greater than 5) 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 Windows Sockets implementation has detected that the network subsystem has failed.
WSAEADDRINUSE	An attempt has been made to **listen()** on an address in use.
WSAEINPROGRESS	A blocking Windows Sockets operation is in progress.
WSAEINVAL	The socket has not been bound with **bind()** or is already connected.
WSAEISCONN	The socket is already connected.
WSAEMFILE	No more file 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**().

4.1.14 ntohl()

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

```
#include <winsock.h>

u_long PASCAL FAR 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()**.

4.1.15 ntohs()

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

#include <winsock.h>

u_short PASCAL FAR 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()**.

4.1.16 recv()

Description Receive data from a socket.

#include <winsock.h>

int PASCAL FAR 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 datagram or stream sockets specified by the *s* parameter and is used to read incoming data.

For sockets 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.

For datagram sockets, data is extracted from the first enqueued datagram, 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**() or **WSAAsyncSelect**() calls may be used to determine when more data arrives.

If the socket is of type SOCK_STREAM 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 2.2.3 for a discussion of this topic.)

Return Value If no error occurs, **recv()** returns the number of bytes received. If the connection has been closed, it 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 Windows Sockets implementation has detected that the network subsystem has failed.
WSAENOTCONN	The socket is not connected.
WSAEINTR	The (blocking) call was canceled via **WSACancelBlockingCall().**
WSAEINPROGRESS	A blocking Windows Sockets operation is in progress.
WSAENOTSOCK	The descriptor is not a socket.
WSAEOPNOTSUPP	MSG_OOB was specified, but the socket is not of type SOCK_STREAM.
WSAESHUTDOWN	The socket has been shutdown; it is not possible to **recv()** on a socket after **shutdown()** has been invoked with *how* set to 0 or 2.
WSAEWOULDBLOCK	The socket is marked as non-blocking and the receive operation would block.
WSAEMSGSIZE	The datagram 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.
WSAECONNRESET	The virtual circuit was reset by the remote side.

See Also **recvfrom**(), ~~**read(),**~~ **recv(), send**(), **select**(), **WSAAsyncSelect**(), **socket**()

4.1.17 recvfrom()

Description Receive a datagram and store the source address.

#include <winsock.h>

int PASCAL FAR 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 sockets 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 SOCK_STREAM sockets.

For datagram sockets, data is extracted from the first enqueued datagram, 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 message, the excess data is lost, and **recvfrom()** returns the error code WSAEMSGSIZE.

If *from* is non-zero, and the socket is of 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**() or **WSAAsyncSelect**() calls may be used to determine when more data arrives.

If the socket is of type SOCK_STREAM 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 **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 2.2.3 for a discussion of this topic.)

Return Value

If no error occurs, **recvfrom()** returns the number of bytes received. If the connection has been closed, it 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 Windows Sockets implementation has detected that the network subsystem has failed.
WSAEFAULT	The *fromlen* argument was invalid: the *from* buffer was too small to accommodate the peer address.
WSAEINTR	The (blocking) call was canceled via **WSACancelBlockingCall().**
WSAEINPROGRESS	A blocking Windows Sockets operation is in progress.
WSAEINVAL	The socket has not been bound with **bind()**.
WSAENOTCONN	The socket is not connected (SOCK_STREAM only).
WSAENOTSOCK	The descriptor is not a socket.
WSAEOPNOTSUPP	MSG_OOB was specified, but the socket is not of type SOCK_STREAM.
WSAESHUTDOWN	The socket has been shutdown; it is not possible to **recvfrom()** on a socket after **shutdown()** has been invoked with *how* set to 0 or 2.
WSAEWOULDBLOCK	The socket is marked as non-blocking and the **recvfrom()** operation would block.
WSAEMSGSIZE	The datagram 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.

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

4.1.18 select()

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

#include <winsock.h>

int PASCAL FAR 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. 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 sockets of type SOCK_STREAM, that the virtual socket 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 SOCK_STREAM, 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 must not be called, and the Windows Sockets implementation must 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.

Error Codes

WSANOTINITIALISED	A successful **WSAStartup()** must occur before using this API.
WSAENETDOWN	The Windows Sockets implementation has detected that the network subsystem has failed.
WSAEINVAL	The *timeout* value is not valid.
WSAEINTR	The (blocking) call was canceled via **WSACancelBlockingCall().**
WSAEINPROGRESS	A blocking Windows Sockets operation is in progress.
WSAENOTSOCK	One of the descriptor sets contains an entry which is not a socket.

See Also

WSAAsyncSelect(), **accept()**, **connect()**, **recv()**, **recvfrom()**, **send()**.

4.1.19 send()

Description Send data on a connected socket.

#include <winsock.h>

int PASCAL FAR 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 on connected datagram or stream sockets and is used to write outgoing data on a socket. For datagram sockets, care must be taken not to exceed the maximum IP packet size of the underlying subnets, which is given by the *iMaxUdpDg* element in the WSAData structure returned by **WSAStartup()**. 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 SOCK_STREAM 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**() 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 Windows Sockets supplier may choose to ignore this flag; see also the discussion of the SO_DONTROUTE option in section 2.4.
MSG_OOB	Send out-of-band data (SOCK_STREAM only; see also section 2.2.3)

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 Windows Sockets implementation has detected that 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 Windows Sockets operation is in progress.
WSAEFAULT	The *buf* argument is not in a valid part of the user address space.
WSAENETRESET	The connection must be reset because the Windows Sockets implementation dropped it.
WSAENOBUFS	The Windows Sockets implementation 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 of type SOCK_STREAM.
WSAESHUTDOWN	The socket has been shutdown; it is not possible to **send()** on a socket after **shutdown()** has been invoked with how set to 1 or 2.
WSAEWOULDBLOCK	The socket is marked as non-blocking and the requested operation would block.
WSAEMSGSIZE	The socket is of type SOCK_DGRAM, and the datagram is larger than the maximum supported by the Windows Sockets implementation.
WSAEINVAL	The socket has not been bound with **bind().**
WSAECONNABORTED	The virtual circuit was aborted due to timeout or other failure.
WSAECONNRESET	The virtual circuit was reset by the remote side.

See Also **recv()**, **recvfrom()**, **socket()**, **sendto()**, **WSAStartup()**.

4.1.20 sendto()

Description Send data to a specific destination.

#include <winsock.h>

int PASCAL FAR 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 on datagram or stream sockets and is used to write outgoing data on a socket. For datagram sockets, care must be taken not to exceed the maximum IP packet size of the underlying subnets, which is given by the *iMaxUdpDg* element in the WSAData structure returned by **WSAStartup()**. 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 SOCK_DGRAM socket to send a datagram to a specific peer socket identified by the *to* parameter. On a SOCK_STREAM socket, the *to* and *tolen* parameters are ignored; in this case the **sendto()** is equivalent to **send**().

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 SOCK_STREAM 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**() 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 Windows Sockets supplier may choose to ignore this flag; see also the discussion of the SO_DONTROUTE option in section .
MSG_OOB	Send out-of-band data (SOCK_STREAM only; see also section)

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 Windows Sockets implementation has detected that 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 Windows Sockets operation is in progress.
WSAEFAULT	The *buf* or *to* parameters are not part of the user address space, or the *to* argument is too small (less than the sizeof a struct sockaddr).
WSAENETRESET	The connection must be reset because the Windows Sockets implementation dropped it.
WSAENOBUFS	The Windows Sockets implementation reports a buffer deadlock.
WSAENOTCONN	The socket is not connected (SOCK_STREAM only).
WSAENOTSOCK	The descriptor is not a socket.
WSAEOPNOTSUPP	MSG_OOB was specified, but the socket is not of type SOCK_STREAM.
WSAESHUTDOWN	The socket has been shutdown; it is not possible to **sendto()** on a socket after **shutdown()** has been invoked with how set to 1 or 2.
WSAEWOULDBLOCK	The socket is marked as non-blocking and the requested operation would block.

WSAEMSGSIZE	The socket is of type SOCK_DGRAM, and the datagram is larger than the maximum supported by the Windows Sockets implementation.
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 **recv**(), **recvfrom**(), **socket**(), **send**(), **WSAStartup**().

4.1.21 setsockopt()

Description Set a socket option.

#include <winsock.h>

int PASCAL FAR 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 IPPROTO_TCP.
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, this specification only defines options that exist 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.

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.

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 Windows Sockets implementation 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.

An application may request that the Windows Sockets implementation enable the use of "keep-alive" packets on TCP connections by turning on the SO_KEEPALIVE socket option. A Windows Sockets implementation 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.

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. TCP_NODELAY is the only supported socket option which uses *level* IPPROTO_TCP; all other options use level SOL_SOCKET.

Windows Sockets suppliers 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.

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

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_KEEPALIVE	BOOL	Send keepalives
SO_LINGER	struct linger FAR *	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.
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_ERROR	int	Get error status and clear
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.

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 Windows Sockets implementation has detected that the network subsystem has failed.
WSAEFAULT	*optval* is not in a valid part of the process address space.
WSAEINPROGRESS	A blocking Windows Sockets operation is in progress.
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. In particular, SO_BROADCAST is not supported on sockets of type SOCK_STREAM, while SO_DONTLINGER, SO_KEEPALIVE, SO_LINGER and SO_OOBINLINE are not supported on sockets of type SOCK_DGRAM.
WSAENOTCONN	Connection has been reset when SO_KEEPALIVE is set.
WSAENOTSOCK	The descriptor is not a socket.

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

4.1.22 shutdown()

Description Disable sends and/or receives on a socket.

#include <winsock.h>

int PASCAL FAR 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 0, 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 1, subsequent sends are disallowed. For TCP sockets, a FIN will be sent.

Setting *how* to 2 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 Windows Sockets implementation 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 Windows Sockets implementation has detected that the network subsystem has failed.
WSAEINVAL	how is not valid.
WSAEINPROGRESS	A blocking Windows Sockets operation is in progress.
WSAENOTCONN	The socket is not connected (SOCK_STREAM only).
WSAENOTSOCK	The descriptor is not a socket.

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

4.1.23 socket()

Description Create a socket.

#include <winsock.h>

SOCKET PASCAL FAR socket (int *af***, int** *type***, int** *protocol* **);**

af	An address format specification. The only format currently supported is PF_INET, which is the ARPA Internet address format.
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()** allocates a socket descriptor of the specified address family, data type and protocol, as well as related resources. If a protocol is not specified (i.e. equal to 0), the default for the specified connection mode is used.

Only a single protocol exists to support a particular socket type using a given address format. 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 *type* specifications are supported:

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.

Sockets of type SOCK_STREAM are full-duplex byte streams. A stream socket 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. Out-of-band data may also be transmitted as described in **send**() and received as described in **recv**().

The communications protocols used to implement a SOCK_STREAM 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.

SOCK_DGRAM 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 **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 Windows Sockets implementation has detected that the network subsystem has failed.
WSAEAFNOSUPPORT	The specified address family is not supported.
WSAEINPROGRESS	A blocking Windows Sockets operation is in progress.
WSAEMFILE	No more file 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**().

4.2 Database Routines

4.2.1 gethostbyaddr()

Description Get host information corresponding to an address.

#include <winsock.h>

struct hostent FAR * PASCAL FAR 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, which must be 4 for PF_INET addresses.

type The type of the address, which must be PF_INET.

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; for Windows Sockets this is always PF_INET.
h_length	The length, in bytes, of each address; for PF_INET, this is always 4.
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 the Windows Sockets implementation. The application must never attempt to modify this structure or to free any of its components. Furthermore, only one copy of this structure is allocated per thread, and so the application should copy any information which it needs before issuing any other Windows Sockets 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 Windows Sockets implementation has detected that 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 Windows Sockets operation is in progress.
WSAEINTR	The (blocking) call was canceled via **WSACancelBlockingCall().**

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

4.2.2 gethostbyname()

Description Get host information corresponding to a hostname.

#include <winsock.h>

struct hostent FAR * PASCAL FAR 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 the Windows Sockets implementation. The application must never attempt to modify this structure or to free any of its components. Furthermore, only one copy of this structure is allocated per thread, and so the application should copy any information which it needs before issuing any other Windows Sockets 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 Windows Sockets implementation has detected that 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 Windows Sockets operation is in progress.
WSAEINTR	The (blocking) call was canceled via **WSACancelBlockingCall().**

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

4.2.3 gethostname()

Description Return the standard host name for the local machine.

#include <winsock.h>

int PASCAL FAR 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 Windows Sockets implementation--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 Windows Sockets implementation has detected that the network subsystem has failed.
WSAEINPROGRESS	A blocking Windows Sockets operation is in progress.

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

4.2.4 getprotobyname()

Description Get protocol information corresponding to a protocol name.

#include <winsock.h>

struct protoent FAR * PASCAL FAR 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 Windows Sockets library. The application must never attempt to modify this structure or to free any of its components. Furthermore only one copy of this structure is allocated per thread, and so the application should copy any information which it needs before issuing any other Windows Sockets 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 Windows Sockets implementation has detected that 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 Windows Sockets operation is in progress.
WSAEINTR	The (blocking) call was canceled via **WSACancelBlockingCall().**

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

4.2.5 getprotobynumber()

Description Get protocol information corresponding to a protocol number.

#include <winsock.h>

struct protoent FAR * PASCAL FAR 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 the Windows Sockets implementation. The application must never attempt to modify this structure or to free any of its components. Furthermore, only one copy of this structure is allocated per thread, and so the application should copy any information which it needs before issuing any other Windows Sockets 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 Windows Sockets implementation has detected that 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 Windows Sockets operation is in progress.
WSAEINTR	The (blocking) call was canceled via **WSACancelBlockingCall().**

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

4.2.6 getservbyname()

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

#include <winsock.h>

struct servent FAR * PASCAL FAR 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 Windows Sockets library. The application must never attempt to modify this structure or to free any of its components. Furthermore only one copy of this structure is allocated per thread, and so the application should copy any information which it needs before issuing any other Windows Sockets 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 Windows Sockets implementation has detected that 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 Windows Sockets operation is in progress.
WSAEINTR	The (blocking) call was canceled via **WSACancelBlockingCall().**

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

4.2.7 getservbyport()

Description Get service information corresponding to a port and protocol.

#include <winsock.h>

struct servent FAR * PASCAL FAR 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 the Windows Sockets implementation. The application must never attempt to modify this structure or to free any of its components. Furthermore, only one copy of this structure is allocated per thread, and so the application should copy any information which it needs before issuing any other Windows Sockets 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 Windows Sockets implementation has detected that 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 Windows Sockets operation is in progress.
WSAEINTR	The (blocking) call was canceled via **WSACancelBlockingCall().**

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

4.3 Microsoft Windows-specific Extensions

4.3.1 WSAAsyncGetHostByAddr()

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

#include <winsock.h>

HANDLE PASCAL FAR 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, which must be 4 for PF_INET.
type	The type of the address, which must be PF_INET.
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 the Windows Sockets implementation 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. The Windows Sockets implementation 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 the Windows Sockets implementation 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**).

Notes For Windows Sockets Suppliers It is the responsibility of the Windows Sockets implementation to ensure that messages are successfully posted to the application. If a **PostMessage()** operation fails, the Windows Sockets implementation must re-post that message as long as the window exists.

Windows Sockets suppliers should use the WSAMAKEASYNCREPLY macro when constructing the *lParam* in the message.

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 Windows Sockets implementation has detected that 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 Windows Sockets implementation has detected that the network subsystem has failed.
WSAEINPROGRESS	A blocking Windows Sockets operation is in progress.
WSAEWOULDBLOCK	The asynchronous operation cannot be scheduled at this time due to resource or other constraints within the Windows Sockets implementation.

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

4.3.2 WSAAsyncGetHostByName()

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

#include <winsock.h>

HANDLE PASCAL FAR 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 the Windows Sockets implementation 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. The Windows Sockets implementation 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 the Windows Sockets implementation 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**).

Notes For Windows Sockets Suppliers It is the responsibility of the Windows Sockets implementation to ensure that messages are successfully posted to the application. If a **PostMessage()** operation fails, the Windows Sockets implementation must re-post that message as long as the window exists.

Windows Sockets suppliers should use the WSAMAKEASYNCREPLY macro when constructing the *lParam* in the message.

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 Windows Sockets implementation has detected that 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 Windows Sockets implementation has detected that the network subsystem has failed.
WSAEINPROGRESS	A blocking Windows Sockets operation is in progress.
WSAEWOULDBLOCK	The asynchronous operation cannot be scheduled at this time due to resource or other constraints within the Windows Sockets implementation.

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

4.3.3 WSAAsyncGetProtoByName()

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

#include <winsock.h>

HANDLE PASCAL FAR 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 the Windows Sockets implementation 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. The Windows Sockets implementation 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 the Windows Sockets implementation 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**).

Notes For Windows Sockets Suppliers It is the responsibility of the Windows Sockets implementation to ensure that messages are successfully posted to the application. If a **PostMessage()** operation fails, the Windows Sockets implementation must re-post that message as long as the window exists.

Windows Sockets suppliers should use the WSAMAKEASYNCREPLY macro when constructing the *lParam* in the message.

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 Windows Sockets implementation has detected that 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 Windows Sockets implementation has detected that the network subsystem has failed.
WSAEINPROGRESS	A blocking Windows Sockets operation is in progress.
WSAEWOULDBLOCK	The asynchronous operation cannot be scheduled at this time due to resource or other constraints within the Windows Sockets implementation.

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

4.3.4 WSAAsyncGetProtoByNumber()

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

#include <winsock.h>

HANDLE PASCAL FAR 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 the Windows Sockets implementation 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. The Windows Sockets implementation 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 the Windows Sockets implementation 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**).

Notes For Windows Sockets Suppliers It is the responsibility of the Windows Sockets implementation to ensure that messages are successfully posted to the application. If a **PostMessage()** operation fails, the Windows Sockets implementation must re-post that message as long as the window exists.

Windows Sockets suppliers should use the WSAMAKEASYNCREPLY macro when constructing the *lParam* in the message.

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 Windows Sockets implementation has detected that 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 Windows Sockets implementation has detected that the network subsystem has failed.
WSAEINPROGRESS	A blocking Windows Sockets operation is in progress.
WSAEWOULDBLOCK	The asynchronous operation cannot be scheduled at this time due to resource or other constraints within the Windows Sockets implementation.

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

4.3.5 WSAAsyncGetServByName()

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

#include <winsock.h>

HANDLE PASCAL FAR 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 the Windows Sockets implementation 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. The Windows Sockets implementation 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 **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 the Windows Sockets implementation 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**).

Notes For Windows Sockets Suppliers It is the responsibility of the Windows Sockets implementation to ensure that messages are successfully posted to the application. If a **PostMessage()** operation fails, the Windows Sockets implementation must re-post that message as long as the window exists.

Windows Sockets suppliers should use the WSAMAKEASYNCREPLY macro when constructing the *lParam* in the message.

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 Windows Sockets implementation has detected that 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 Windows Sockets implementation has detected that the network subsystem has failed.
WSAEINPROGRESS	A blocking Windows Sockets operation is in progress.
WSAEWOULDBLOCK	The asynchronous operation cannot be scheduled at this time due to resource or other constraints within the Windows Sockets implementation.

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

4.3.6 WSAAsyncGetServByPort()

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

#include <winsock.h>

HANDLE PASCAL FAR 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 the Windows Sockets implementation 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. The Windows Sockets implementation 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 the Windows Sockets implementation 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**).

Notes For Windows Sockets Suppliers It is the responsibility of the Windows Sockets implementation to ensure that messages are successfully posted to the application. If a **PostMessage()** operation fails, the Windows Sockets implementation must re-post that message as long as the window exists.

Windows Sockets suppliers should use the WSAMAKEASYNCREPLY macro when constructing the *lParam* in the message.

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 Windows Sockets implementation has detected that 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 Windows Sockets implementation has detected that the network subsystem has failed.
WSAEINPROGRESS	A blocking Windows Sockets operation is in progress.
WSAEWOULDBLOCK	The asynchronous operation cannot be scheduled at this time due to resource or other constraints within the Windows Sockets implementation.

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

4.3.7 WSAAsyncSelect()

Description Request event notification for a socket.

#include <winsock.h>

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

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

Remarks This function is used to request that the Windows Sockets 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.

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

Issuing a **WSAAsyncSelect()** for a socket cancels any previous **WSAAsyncSelect()** 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 the Windows Sockets implementation 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.[7]

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*.
FD_ACCEPT	Socket *s* ready for accepting a new incoming connection
FD_CONNECT	Connection on socket *s* completed
FD_CLOSE	Connection identified by socket *s* has been closed

[7]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.

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 Windows Sockets API call which returns WSAEWOULDBLOCK immediately. For example, the following sequence of events is possible:

(i) data arrives on socket **s**; Windows Sockets 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.

The Windows Sockets 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()**
FD_CONNECT	NONE
FD_CLOSE	NONE

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, and FD_ACCEPT 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 concern itself with the amount of data that arrives at any one time. Consider the following sequence:

(i) Windows Sockets DLL receives 100 bytes of data on socket **s** and posts an FD_READ message.
(ii) The application issues **recv(s, buffptr, 50, 0)** to read 50 bytes.
(iii) The Windows Sockets DLL posts another FD_READ message 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 is true when the application initially calls **WSAAsyncSelect()** or when the reenabling function is called, then a message is posted as appropriate. For example, if an application calls **listen()**, a connect attempt is made, then the application calls **WSAAsyncSelect()** specifying that it wants to receive FD_ACCEPT messages for the socket, the Windows Sockets implementation posts an FD_ACCEPT message immediately.

The FD_WRITE event is handled slightly differently. An FD_WRITE message is posted when a socket is first connected with **connect()** or accepted with **accept()**, and then after a **send()** or **sendto()** fails with WSAEWOULDBLOCK and buffer space becomes available. Therefore, an application can assume that sends are possible starting from the first FD_WRITE message and lasting until a send returns WSAEWOULDBLOCK. After such a failure the application will be notified that sends are again possible with an FD_WRITE message.

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

The error code in an FD_CLOSE message indicates whether the socket close was graceful or abortive. If the error code is 0, then the close was graceful; if the error code is WSAECONNRESET, then the socket's virtual socket was reset. This only applies to sockets of type 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. It will NOT receive an FD_READ message to indicate this condition.

Error Codes

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

WSAENETDOWN	The Windows Sockets implementation has detected that the network subsystem has failed.
WSAEINVAL	Indicates that one of the specified parameters was invalid
WSAEINPROGRESS	A blocking Windows Sockets operation is in progress.

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.
WSAEDESTADDRREQ	A destination address is required.
WSAEFAULT	The namelen argument is incorrect.
WSAEINVAL	The socket is already bound to an address.
WSAEISCONN	The socket is already connected.
WSAEMFILE	No more file 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.
WSAENOTCONN	The socket is not connected.
WSAENOTSOCK	The descriptor is a file, not a socket.
WSAETIMEDOUT	Attempt to connect timed out without establishing a connection

Event: FD_CLOSE

Error Code	Meaning
WSAENETDOWN	The Windows Sockets implementation has detected that 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

Error Code	Meaning
WSAENETDOWN	The Windows Sockets implementation has detected that the network subsystem has failed.

Notes For Windows Sockets Suppliers

It is the responsibility of the Windows Sockets Supplier to ensure that messages are successfully posted to the application. If a **PostMessage()** operation fails, the Windows Sockets implementation MUST re-post that message as long as the window exists.

Windows Sockets suppliers should use the WSAMAKESELECTREPLY macro when constructing the *lParam* in the message.

When a socket is closed, the Windows Sockets Supplier should purge any messages remaining for posting to the application window. However the application must be prepared to receive, and discard, any messages which may have been posted prior to the **closesocket()**.

See Also **select()**

4.3.8 WSACancelAsyncRequest()

Description Cancel an incomplete asynchronous operation.

#include <winsock.h>

int PASCAL FAR 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.

Notes For Windows Sockets Suppliers 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 Windows Sockets specification does not prescribe how a conformant Windows Sockets implementation should distinguish between the two cases. For maximum portability, a Windows Sockets application should treat the two errors as equivalent.

Error Codes

WSANOTINITIALISED	A successful **WSAStartup()** must occur before using this API.
WSAENETDOWN	The Windows Sockets implementation has detected that the network subsystem has failed.
WSAEINVAL	Indicates that the specified asynchronous task handle was invalid
WSAEINPROGRESS	A blocking Windows Sockets operation is in progress.
WSAEALREADY	The asynchronous routine being canceled has already completed.

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

4.3.9 WSACancelBlockingCall()

Description Cancel a blocking call which is currently in progress.

#include <winsock.h>

int PASCAL FAR 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 Windows Sockets 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 Windows Sockets. In (2), the blocking call will be terminated as soon as the blocking hook function completes.)

In the case of a blocking **connect()** operation, the Windows Sockets implementation 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.

Cancelling 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.

Cancelling 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 Windows Sockets 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 Windows Sockets implementation 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 Windows Sockets implementation has detected that the network subsystem has failed.
WSAEINVAL	Indicates that there is no outstanding blocking call.

4.3.10 WSACleanup()

Description Terminate use of the Windows Sockets DLL.

#include <winsock.h>

int PASCAL FAR WSACleanup (void);

Remarks An application or DLL is required to perform a (successful) **WSAStartup()** call before it can use Windows Sockets services. When it has completed the use of Windows Sockets, the application or DLL must call **WSACleanup()** to deregister itself from a Windows Sockets implementation and allow the implementation to free any resources allocated on behalf of the application or DLL. Any open SOCK_STREAM 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 Windows Sockets 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 Windows Sockets 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.

Notes For Windows Sockets Suppliers Well-behaved Windows Sockets applications will make a **WSACleanup()** call to indicate deregistration from a Windows Sockets implementation. This function can thus, for example, be utilized to free up resources allocated to the specific application.

A Windows Sockets implementation must be prepared to deal with an application which terminates without invoking **WSACleanup()** - for example, as a result of an error.

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

A Windows Sockets implementation must ensure that **WSACleanup()** leaves things in a state in which the application can invoke **WSAStartup()** to re-establish Windows Sockets usage.

Error Codes

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

WSAENETDOWN	The Windows Sockets implementation has detected that the network subsystem has failed.
WSAEINPROGRESS	A blocking Windows Sockets operation is in progress.

See Also **WSAStartup()**

4.3.11 WSAGetLastError()

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

```
#include <winsock.h>

int PASCAL FAR WSAGetLastError ( void );
```

Remarks This function returns the last network error that occurred. When a particular Windows Sockets API 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 Windows Sockets API routine performed by this thread.

Notes For Windows Sockets Suppliers The use of the **WSAGetLastError()** function to retrieve the last error code, rather than relying on a global error variable (cf. *errno*), is required in order to provide compatibility with future multi-threaded environments.

Note that in a nonpreemptive Windows environment **WSAGetLastError()** is used to retrieve only Windows Sockets API errors. In a preemptive environment, **WSAGetLastError()** will invoke **GetLastError()**, which is used to retrieve the error status for all Win32 API functions on a per-thread basis. For portability, an application should use **WSAGetLastError()** immediately after the Windows Sockets API function which failed.

See Also **WSASetLastError()**

4.3.12 WSAIsBlocking()

Description Determine if a blocking call is in progress.

#include <winsock.h>

BOOL PASCAL FAR 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 Although a call issued on a blocking socket appears to an application program as though it "blocks", the Windows Sockets 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 Windows Sockets prohibits more than one outstanding call per thread.

Notes For Windows Sockets Suppliers A Windows Sockets implementation must prohibit more than one outstanding blocking call per thread.

4.3.13 WSASetBlockingHook()

Description Establish an application-specific blocking hook function.

#include <winsock.h>

FARPROC PASCAL FAR 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 Windows Sockets implementation should use to implement blocking socket function calls.

A Windows Sockets 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 Windows Sockets API operation, the Windows Sockets implementation 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 Windows Sockets 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 Windows Sockets API 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 Windows Sockets implementation has detected that the network subsystem has failed.
WSAEINPROGRESS	A blocking Windows Sockets operation is in progress.

See Also **WSAUnhookBlockingHook()**

4.3.14 WSASetLastError()

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

#include <winsock.h>

void PASCAL FAR WSASetLastError (int *iError* **);**

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 Windows Sockets routine called by the application will override the error code as set by this routine.

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

Notes For Windows Sockets Suppliers In a Win32 environment, this function will invoke **SetLastError()**.

Return Value None.

Error Codes

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

See Also **WSAGetLastError()**

4.3.15 WSAStartup()

Description

#include <winsock.h>

int PASCAL FAR WSAStartup (WORD *wVersionRequested***,**
LPWSADATA *lpWSAData* **);**

wVersionRequested	The highest version of Windows Sockets API 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 Windows Sockets implementation.

Remarks

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

In order to support future Windows Sockets implementations and applications which may have functionality differences from Windows Sockets 1.1, a negotiation takes place in **WSAStartup()**. The caller of **WSAStartup()** and the Windows Sockets 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 Windows Sockets 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 Windows Sockets 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 Windows Sockets DLL or fail to initialize.

This negotiation allows both a Windows Sockets DLL and a Windows Sockets application to support a range of Windows Sockets versions. An application can successfully utilize a Windows Sockets 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 Windows Sockets 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 Windows Sockets 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 Windows Sockets 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 Windows Sockets DLL is acceptable.  Proceed.  */
```

And this code fragment demonstrates how a Windows Sockets 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 Windows Sockets API calls as needed. When it has finished using the services of the Windows Sockets DLL, the application or DLL must call **WSACleanup()** in order to allow the Windows Sockets DLL to free any resources for the application.

Details of the actual Windows Sockets 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 Windows Sockets specification that the Windows Sockets DLL expects the caller to use.
wHighVersion	The highest version of the Windows Sockets specification that this DLL can support (also encoded as above). Normally this will be the same as *wVersion*.
szDescription	A null-terminated ASCII string into which the Windows Sockets DLL copies a description of the Windows Sockets implementation, including vendor identification. The text (up to 256 characters in length) may contain any characters, but vendors are cautioned against including 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 Windows Sockets DLL copies relevant status or configuration information. The Windows Sockets 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. A Windows Sockets implementation may provide a global pool of sockets for allocation to any process; alternatively it may allocate per-process resources for sockets. The number may well reflect the way in which the Windows Sockets DLL or the networking software was configured. Application writers may use this number as a crude indication of whether the Windows Sockets 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 Windows Sockets applications in use.
iMaxUdpDg	The size in bytes of the largest UDP datagram that can be sent or received by a Windows Sockets application. If the implementation imposes no limit, *iMaxUdpDg* is zero. In many implementations of Berkeley sockets, there is an implicit limit of 8192 bytes on UDP datagrams (which are fragmented if necessary). A Windows Sockets implementation may impose a limit based, for instance, on the allocation of fragment reassembly buffers. The minimum value of *iMaxUdpDg* for a compliant Windows Sockets implementation is 512.

Note that regardless of the value of *iMaxUdpDg*, it is inadvisable to attempt to send a broadcast datagram which is larger than the Maximum Transmission Unit (MTU) for the network. (The Windows Sockets API does not provide a mechanism to discover the MTU, but it must be no less than 512 bytes.)

lpVendorInfo — A far pointer to a vendor-specific data structure. The definition of this structure (if supplied) is beyond the scope of this specification.

An application or DLL may call **WSAStartup()** more than once if it needs to obtain the WSAData structure information more than once. However, the *wVersionRequired* parameter is assumed to be the same on all calls to **WSAStartup()**; that is, an application or DLL cannot change the version of Windows Sockets it expects after the initial call to **WSAStartup().**

There must be one **WSACleanup()** call corresponding to every **WSAStartup()** call to allow third-party DLLs to make use of a Windows Sockets 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 Windows Sockets DLL may not have established the client data area where the "last error" information is stored.

Notes For Windows Sockets Suppliers

Each Windows Sockets application MUST make a **WSAStartup()** call before issuing any other Windows Sockets API calls. This function can thus be utilized for initialization purposes.

Further issues are discussed in the notes for **WSACleanup()**.

Error Codes

WSASYSNOTREADY	Indicates that the underlying network subsystem is not ready for network communication.
WSAVERNOTSUPPORTED	The version of Windows Sockets API support requested is not provided by this particular Windows Sockets implementation.
WSAEINVAL	The Windows Sockets version specified by the application is not supported by this DLL.

See Also

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

4.3.16 WSAUnhookBlockingHook()

Description Restore the default blocking hook function.

#include <winsock.h>

int PASCAL FAR 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()**

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 Windows Sockets-compliant implementations.

Windows Sockets 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 Windows Sockets API 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 Windows Sockets 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 Windows Sockets 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 Windows Sockets versions of regular Berkeley Sockets error codes.

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

The fourth set of errors are returned by Windows Sockets **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 Windows Sockets implementation does not use the DNS, it will use the most appropriate code. In general, a Windows Sockets 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 Windows Sockets 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 Windows Sockets 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

A Windows Sockets supplier who provides a development kit to support the development of Windows Sockets applications must supply 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 Windows Sockets specification. An application writer may choose to ignore the compatibility headers and include **winsock.h** in each source file.

A.2.2 Windows Sockets 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 Windows Sockets DLL vendor **MUST NOT** make any modifications to this header file which could impact binary compatibility of Windows Sockets applications. The constant values, function parameters and return codes, and the like must remain consistent across all Windows Sockets DLL vendors.

```
/* WINSOCK.H--definitions to be used with the WINSOCK.DLL
 *
 * This header file corresponds to version 1.1 of the Windows Sockets 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 _WINSOCKAPI_
#define _WINSOCKAPI_

/*
 * Pull in WINDOWS.H if necessary
 */
#ifndef _INC_WINDOWS
#include <windows.h>
#endif /* _INC_WINDOWS */

/*
 * Basic system type definitions, taken from the BSD file sys/types.h.
 */
typedef unsigned char   u_char;
typedef unsigned short  u_short;
typedef unsigned int    u_int;
typedef unsigned long   u_long;

/*
 * The new type to be used in all
 * instances which refer to sockets.
 */
typedef u_int           SOCKET;

/*
 * Select uses arrays of SOCKETs.  These macros manipulate such
 * arrays.  FD_SETSIZE may be defined by the user before including
 * this file, but the default here should be >= 64.
 *
 * CAVEAT IMPLEMENTOR and USER: THESE MACROS AND TYPES MUST BE
 * INCLUDED IN WINSOCK.H EXACTLY AS SHOWN HERE.
 */
#ifndef FD_SETSIZE
#define FD_SETSIZE      64
#endif /* FD_SETSIZE */

typedef struct fd_set {
        u_short fd_count;               /* how many are SET? */
        SOCKET  fd_array[FD_SETSIZE];   /* an array of SOCKETs */
} fd_set;

extern int PASCAL FAR __WSAFDIsSet(SOCKET, fd_set FAR *);
```

```
#define FD_CLR(fd, set) do { \
    u_int __i; \
    for (__i = 0; __i < ((fd_set FAR *)(set))->fd_count ; __i++) { \
        if (((fd_set FAR *)(set))->fd_array[__i] == fd) { \
            while (__i < ((fd_set FAR *)(set))->fd_count-1) { \
                ((fd_set FAR *)(set))->fd_array[__i] = \
                    ((fd_set FAR *)(set))->fd_array[__i+1]; \
                __i++; \
            } \
            ((fd_set FAR *)(set))->fd_count--; \
            break; \
        } \
    } \
} while(0)

#define FD_SET(fd, set) do { \
    if (((fd_set FAR *)(set))->fd_count < FD_SETSIZE) \
        ((fd_set FAR *)(set))->fd_array[((fd_set FAR *)(set))->fd_count++]=fd;\
} while(0)

#define FD_ZERO(set) (((fd_set FAR *)(set))->fd_count=0)

#define FD_ISSET(fd, set) __WSAFDIsSet((SOCKET)fd, (fd_set FAR *)set)

/*
 * Structure used in select() call, taken from the BSD file sys/time.h.
 */
struct timeval {
        long    tv_sec;         /* seconds */
        long    tv_usec;        /* and microseconds */
};

/*
 * Operations on timevals.
 *
 * NB: timercmp does not work for >= or <=.
 */
#define timerisset(tvp)         ((tvp)->tv_sec || (tvp)->tv_usec)
#define timercmp(tvp, uvp, cmp) \
        ((tvp)->tv_sec cmp (uvp)->tv_sec || \
         (tvp)->tv_sec == (uvp)->tv_sec && (tvp)->tv_usec cmp (uvp)->tv_usec)
#define timerclear(tvp)         (tvp)->tv_sec = (tvp)->tv_usec = 0

/*
 * Commands for ioctlsocket(),  taken from the BSD file fcntl.h.
 *
 *
 * Ioctl's have the command encoded in the lower word,
 * and the size of any in or out parameters in the upper
 * word.  The high 2 bits of the upper word are used
 * to encode the in/out status of the parameter; for now
 * we restrict parameters to at most 128 bytes.
 */
#define IOCPARM_MASK    0x7f            /* parameters must be < 128 bytes */
#define IOC_VOID        0x20000000      /* no parameters */
#define IOC_OUT         0x40000000      /* copy out parameters */
#define IOC_IN          0x80000000      /* copy in parameters */
#define IOC_INOUT       (IOC_IN|IOC_OUT)
                                        /* 0x20000000 distinguishes new &
                                           old ioctl's */
#define _IO(x,y)        (IOC_VOID|(x<<8)|y)

#define _IOR(x,y,t)     (IOC_OUT|(((long)sizeof(t)&IOCPARM_MASK)<<16)|(x<<8)|y)

#define _IOW(x,y,t)     (IOC_IN|(((long)sizeof(t)&IOCPARM_MASK)<<16)|(x<<8)|y)

#define FIONREAD    _IOR('f', 127, u_long) /* get # bytes to read */
#define FIONBIO     _IOW('f', 126, u_long) /* set/clear non-blocking i/o */
#define FIOASYNC    _IOW('f', 125, u_long) /* set/clear async i/o */

/* Socket I/O Controls */
#define SIOCSHIWAT  _IOW('s',  0, u_long)  /* set high watermark */
#define SIOCGHIWAT  _IOR('s',  1, u_long)  /* get high watermark */
#define SIOCSLOWAT  _IOW('s',  2, u_long)  /* set low watermark */
```

```
#define SIOCGLOWAT  _IOR('s',  3, u_long)  /* get low watermark */
#define SIOCATMARK  _IOR('s',  7, u_long)  /* at oob mark? */

/*
 * Structures returned by network data base library, taken from the
 * BSD file netdb.h.  All addresses are supplied in host order, and
 * returned in network order (suitable for use in system calls).
 */

struct  hostent {
        char    FAR * h_name;           /* official name of host */
        char    FAR * FAR * h_aliases;  /* alias list */
        short   h_addrtype;             /* host address type */
        short   h_length;               /* length of address */
        char    FAR * FAR * h_addr_list; /* list of addresses */
#define h_addr  h_addr_list[0]          /* address, for backward compat */
};

/*
 * It is assumed here that a network number
 * fits in 32 bits.
 */
struct  netent {
        char    FAR * n_name;           /* official name of net */
        char    FAR * FAR * n_aliases;  /* alias list */
        short   n_addrtype;             /* net address type */
        u_long  n_net;                  /* network # */
};

struct  servent {
        char    FAR * s_name;           /* official service name */
        char    FAR * FAR * s_aliases;  /* alias list */
        short   s_port;                 /* port # */
        char    FAR * s_proto;          /* protocol to use */
};

struct  protoent {
        char    FAR * p_name;           /* official protocol name */
        char    FAR * FAR * p_aliases;  /* alias list */
        short   p_proto;                /* protocol # */
};

/*
 * Constants and structures defined by the internet system,
 * Per RFC 790, September 1981, taken from the BSD file netinet/in.h.
 */

/*
 * Protocols
 */
#define IPPROTO_IP              0               /* dummy for IP */
#define IPPROTO_ICMP            1               /* control message protocol */
#define IPPROTO_GGP             2               /* gateway^2 (deprecated) */
#define IPPROTO_TCP             6               /* tcp */
#define IPPROTO_PUP             12              /* pup */
#define IPPROTO_UDP             17              /* user datagram protocol */
#define IPPROTO_IDP             22              /* xns idp */
#define IPPROTO_ND              77              /* UNOFFICIAL net disk proto */

#define IPPROTO_RAW             255             /* raw IP packet */
#define IPPROTO_MAX             256

/*
 * Port/socket numbers: network standard functions
 */
#define IPPORT_ECHO             7
#define IPPORT_DISCARD          9
#define IPPORT_SYSTAT           11
#define IPPORT_DAYTIME          13
#define IPPORT_NETSTAT          15
#define IPPORT_FTP              21
#define IPPORT_TELNET           23
#define IPPORT_SMTP             25
#define IPPORT_TIMESERVER       37
#define IPPORT_NAMESERVER       42
```

```
#define IPPORT_WHOIS            43
#define IPPORT_MTP              57

/*
 * Port/socket numbers: host specific functions
 */
#define IPPORT_TFTP             69
#define IPPORT_RJE              77
#define IPPORT_FINGER           79
#define IPPORT_TTYLINK          87
#define IPPORT_SUPDUP           95

/*
 * UNIX TCP sockets
 */
#define IPPORT_EXECSERVER       512
#define IPPORT_LOGINSERVER      513
#define IPPORT_CMDSERVER        514
#define IPPORT_EFSSERVER        520

/*
 * UNIX UDP sockets
 */
#define IPPORT_BIFFUDP          512
#define IPPORT_WHOSERVER        513
#define IPPORT_ROUTESERVER      520
                                        /* 520+1 also used */

/*
 * Ports < IPPORT_RESERVED are reserved for
 * privileged processes (e.g. root).
 */
#define IPPORT_RESERVED         1024

/*
 * Link numbers
 */
#define IMPLINK_IP              155
#define IMPLINK_LOWEXPER        156
#define IMPLINK_HIGHEXPER       158

/*
 * Internet address (old style... should be updated)
 */
struct in_addr {
        union {
                struct { u_char s_b1,s_b2,s_b3,s_b4; } S_un_b;
                struct { u_short s_w1,s_w2; } S_un_w;
                u_long S_addr;
        } S_un;
#define s_addr  S_un.S_addr
                                /* can be used for most tcp & ip code */
#define s_host  S_un.S_un_b.s_b2
                                /* host on imp */
#define s_net   S_un.S_un_b.s_b1
                                /* network */
#define s_imp   S_un.S_un_w.s_w2
                                /* imp */
#define s_impno S_un.S_un_b.s_b4
                                /* imp # */
#define s_lh    S_un.S_un_b.s_b3
                                /* logical host */
};

/*
 * Definitions of bits in internet address integers.
 * On subnets, the decomposition of addresses to host and net parts
 * is done according to subnet mask, not the masks here.
 */
#define IN_CLASSA(i)            (((long)(i) & 0x80000000) == 0)
#define IN_CLASSA_NET           0xff000000
#define IN_CLASSA_NSHIFT        24
#define IN_CLASSA_HOST          0x00ffffff
#define IN_CLASSA_MAX           128
```

```
#define IN_CLASSB(i)            (((long)(i) & 0xc0000000) == 0x80000000)
#define IN_CLASSB_NET           0xffff0000
#define IN_CLASSB_NSHIFT        16
#define IN_CLASSB_HOST          0x0000ffff
#define IN_CLASSB_MAX           65536

#define IN_CLASSC(i)            (((long)(i) & 0xc0000000) == 0xc0000000)
#define IN_CLASSC_NET           0xffffff00
#define IN_CLASSC_NSHIFT        8
#define IN_CLASSC_HOST          0x000000ff

#define INADDR_ANY              (u_long)0x00000000
#define INADDR_LOOPBACK         0x7f000001
#define INADDR_BROADCAST        (u_long)0xffffffff
#define INADDR_NONE             0xffffffff

/*
 * Socket address, internet style.
 */
struct sockaddr_in {
        short   sin_family;
        u_short sin_port;
        struct  in_addr sin_addr;
        char    sin_zero[8];
};

#define WSADESCRIPTION_LEN      256
#define WSASYS_STATUS_LEN       128

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

typedef WSADATA FAR *LPWSADATA;

/*
 * Options for use with [gs]etsockopt at the IP level.
 */
#define IP_OPTIONS      1               /* set/get IP per-packet options */

/*
 * Definitions related to sockets: types, address families, options,
 * taken from the BSD file sys/socket.h.
 */

/*
 * This is used instead of -1, since the
 * SOCKET type is unsigned.
 */
#define INVALID_SOCKET  (SOCKET)(~0)
#define SOCKET_ERROR            (-1)

/*
 * Types
 */
#define SOCK_STREAM     1               /* stream socket */
#define SOCK_DGRAM      2               /* datagram socket */
#define SOCK_RAW        3               /* raw-protocol interface */
#define SOCK_RDM        4               /* reliably-delivered message */
#define SOCK_SEQPACKET  5               /* sequenced packet stream */

/*
 * Option flags per-socket.
 */
#define SO_DEBUG        0x0001          /* turn on debugging info recording */
#define SO_ACCEPTCONN   0x0002          /* socket has had listen() */
#define SO_REUSEADDR    0x0004          /* allow local address reuse */
#define SO_KEEPALIVE    0x0008          /* keep connections alive */
#define SO_DONTROUTE    0x0010          /* just use interface addresses */
```

```
#define SO_BROADCAST    0x0020          /* permit sending of broadcast msgs */
#define SO_USELOOPBACK  0x0040          /* bypass hardware when possible */
#define SO_LINGER       0x0080          /* linger on close if data present */
#define SO_OOBINLINE    0x0100          /* leave received OOB data in line */

#define SO_DONTLINGER   (u_int)(~SO_LINGER)

/*
 * Additional options.
 */
#define SO_SNDBUF       0x1001          /* send buffer size */
#define SO_RCVBUF       0x1002          /* receive buffer size */
#define SO_SNDLOWAT     0x1003          /* send low-water mark */
#define SO_RCVLOWAT     0x1004          /* receive low-water mark */
#define SO_SNDTIMEO     0x1005          /* send timeout */
#define SO_RCVTIMEO     0x1006          /* receive timeout */
#define SO_ERROR        0x1007          /* get error status and clear */
#define SO_TYPE         0x1008          /* get socket type */

/*
 * TCP options.
 */
#define TCP_NODELAY     0x0001

/*
 * Address families.
 */
#define AF_UNSPEC       0               /* unspecified */
#define AF_UNIX         1               /* local to host (pipes, portals) */
#define AF_INET         2               /* internetwork: UDP, TCP, etc. */
#define AF_IMPLINK      3               /* arpanet imp addresses */
#define AF_PUP          4               /* pup protocols: e.g. BSP */
#define AF_CHAOS        5               /* mit CHAOS protocols */
#define AF_NS           6               /* XEROX NS protocols */
#define AF_ISO          7               /* ISO protocols */
#define AF_OSI          AF_ISO          /* OSI is ISO */
#define AF_ECMA         8               /* european computer manufacturers */
#define AF_DATAKIT      9               /* datakit protocols */
#define AF_CCITT        10              /* CCITT protocols, X.25 etc */
#define AF_SNA          11              /* IBM SNA */
#define AF_DECnet       12              /* DECnet */
#define AF_DLI          13              /* Direct data link interface */
#define AF_LAT          14              /* LAT */
#define AF_HYLINK       15              /* NSC Hyperchannel */
#define AF_APPLETALK    16              /* AppleTalk */
#define AF_NETBIOS      17              /* NetBios-style addresses */

#define AF_MAX          18

/*
 * Structure used by kernel to store most
 * addresses.
 */
struct sockaddr {
        u_short sa_family;              /* address family */
        char    sa_data[14];            /* up to 14 bytes of direct address */
};

/*
 * Structure used by kernel to pass protocol
 * information in raw sockets.
 */
struct sockproto {
        u_short sp_family;              /* address family */
        u_short sp_protocol;            /* protocol */
};

/*
 * Protocol families, same as address families for now.
 */
#define PF_UNSPEC       AF_UNSPEC
#define PF_UNIX         AF_UNIX
#define PF_INET         AF_INET
#define PF_IMPLINK      AF_IMPLINK
#define PF_PUP          AF_PUP
```

```
#define PF_CHAOS        AF_CHAOS
#define PF_NS           AF_NS
#define PF_ISO          AF_ISO
#define PF_OSI          AF_OSI
#define PF_ECMA         AF_ECMA
#define PF_DATAKIT      AF_DATAKIT
#define PF_CCITT        AF_CCITT
#define PF_SNA          AF_SNA
#define PF_DECnet       AF_DECnet
#define PF_DLI          AF_DLI
#define PF_LAT          AF_LAT
#define PF_HYLINK       AF_HYLINK
#define PF_APPLETALK    AF_APPLETALK

#define PF_MAX          AF_MAX

/*
 * Structure used for manipulating linger option.
 */
struct  linger {
        u_short l_onoff;                /* option on/off */
        u_short l_linger;               /* linger time */
};

/*
 * Level number for (get/set)sockopt() to apply to socket itself.
 */
#define SOL_SOCKET      0xffff          /* options for socket level */

/*
 * Maximum queue length specifiable by listen.
 */
#define SOMAXCONN       5

#define MSG_OOB         0x1             /* process out-of-band data */
#define MSG_PEEK        0x2             /* peek at incoming message */
#define MSG_DONTROUTE   0x4             /* send without using routing tables */

#define MSG_MAXIOVLEN   16

/*
 * Define constant based on rfc883, used by gethostbyxxxx() calls.
 */
#define MAXGETHOSTSTRUCT        1024

/*
 * Define flags to be used with the WSAAsyncSelect() call.
 */
#define FD_READ         0x01
#define FD_WRITE        0x02
#define FD_OOB          0x04
#define FD_ACCEPT       0x08
#define FD_CONNECT      0x10
#define FD_CLOSE        0x20

/*
 * All Windows Sockets error constants are biased by WSABASEERR from
 * the "normal"
 */
#define WSABASEERR              10000
/*
 * Windows Sockets definitions of regular Microsoft C error constants
 */
#define WSAEINTR                (WSABASEERR+4)
#define WSAEBADF                (WSABASEERR+9)
#define WSAEACCES               (WSABASEERR+13)
#define WSAEFAULT               (WSABASEERR+14)
#define WSAEINVAL               (WSABASEERR+22)
#define WSAEMFILE               (WSABASEERR+24)

/*
 * Windows Sockets definitions of regular Berkeley error constants
 */
#define WSAEWOULDBLOCK          (WSABASEERR+35)
#define WSAEINPROGRESS          (WSABASEERR+36)
```

```
#define WSAEALREADY             (WSABASEERR+37)
#define WSAENOTSOCK             (WSABASEERR+38)
#define WSAEDESTADDRREQ         (WSABASEERR+39)
#define WSAEMSGSIZE             (WSABASEERR+40)
#define WSAEPROTOTYPE           (WSABASEERR+41)
#define WSAENOPROTOOPT          (WSABASEERR+42)
#define WSAEPROTONOSUPPORT      (WSABASEERR+43)
#define WSAESOCKTNOSUPPORT      (WSABASEERR+44)
#define WSAEOPNOTSUPP           (WSABASEERR+45)
#define WSAEPFNOSUPPORT         (WSABASEERR+46)
#define WSAEAFNOSUPPORT         (WSABASEERR+47)
#define WSAEADDRINUSE           (WSABASEERR+48)
#define WSAEADDRNOTAVAIL        (WSABASEERR+49)
#define WSAENETDOWN             (WSABASEERR+50)
#define WSAENETUNREACH          (WSABASEERR+51)
#define WSAENETRESET            (WSABASEERR+52)
#define WSAECONNABORTED         (WSABASEERR+53)
#define WSAECONNRESET           (WSABASEERR+54)
#define WSAENOBUFS              (WSABASEERR+55)
#define WSAEISCONN              (WSABASEERR+56)
#define WSAENOTCONN             (WSABASEERR+57)
#define WSAESHUTDOWN            (WSABASEERR+58)
#define WSAETOOMANYREFS         (WSABASEERR+59)
#define WSAETIMEDOUT            (WSABASEERR+60)
#define WSAECONNREFUSED         (WSABASEERR+61)
#define WSAELOOP                (WSABASEERR+62)
#define WSAENAMETOOLONG         (WSABASEERR+63)
#define WSAEHOSTDOWN            (WSABASEERR+64)
#define WSAEHOSTUNREACH         (WSABASEERR+65)
#define WSAENOTEMPTY            (WSABASEERR+66)
#define WSAEPROCLIM             (WSABASEERR+67)
#define WSAEUSERS               (WSABASEERR+68)
#define WSAEDQUOT               (WSABASEERR+69)
#define WSAESTALE               (WSABASEERR+70)
#define WSAEREMOTE              (WSABASEERR+71)

/*
 * Extended Windows Sockets error constant definitions
 */
#define WSASYSNOTREADY          (WSABASEERR+91)
#define WSAVERNOTSUPPORTED      (WSABASEERR+92)
#define WSANOTINITIALISED       (WSABASEERR+93)

/*
 * Error return codes from gethostbyname() and gethostbyaddr()
 * (when using the resolver). Note that these errors are
 * retrieved via WSAGetLastError() and must therefore follow
 * the rules for avoiding clashes with error numbers from
 * specific implementations or language run-time systems.
 * For this reason the codes are based at WSABASEERR+1001.
 * Note also that [WSA]NO_ADDRESS is defined only for
 * compatibility purposes.
 */

#define h_errno         WSAGetLastError()

/* Authoritative Answer: Host not found */
#define WSAHOST_NOT_FOUND       (WSABASEERR+1001)
#define HOST_NOT_FOUND          WSAHOST_NOT_FOUND

/* Non-Authoritative: Host not found, or SERVERFAIL */
#define WSATRY_AGAIN            (WSABASEERR+1002)
#define TRY_AGAIN               WSATRY_AGAIN

/* Non recoverable errors, FORMERR, REFUSED, NOTIMP */
#define WSANO_RECOVERY          (WSABASEERR+1003)
#define NO_RECOVERY             WSANO_RECOVERY

/* Valid name, no data record of requested type */
#define WSANO_DATA              (WSABASEERR+1004)
#define NO_DATA                 WSANO_DATA

/* no address, look for MX record */
#define WSANO_ADDRESS           WSANO_DATA
#define NO_ADDRESS              WSANO_ADDRESS
```

```
/*
 * Windows Sockets errors redefined as regular Berkeley error constants
 */
#define EWOULDBLOCK             WSAEWOULDBLOCK
#define EINPROGRESS             WSAEINPROGRESS
#define EALREADY                WSAEALREADY
#define ENOTSOCK                WSAENOTSOCK
#define EDESTADDRREQ            WSAEDESTADDRREQ
#define EMSGSIZE                WSAEMSGSIZE
#define EPROTOTYPE              WSAEPROTOTYPE
#define ENOPROTOOPT             WSAENOPROTOOPT
#define EPROTONOSUPPORT         WSAEPROTONOSUPPORT
#define ESOCKTNOSUPPORT         WSAESOCKTNOSUPPORT
#define EOPNOTSUPP              WSAEOPNOTSUPP
#define EPFNOSUPPORT            WSAEPFNOSUPPORT
#define EAFNOSUPPORT            WSAEAFNOSUPPORT
#define EADDRINUSE              WSAEADDRINUSE
#define EADDRNOTAVAIL           WSAEADDRNOTAVAIL
#define ENETDOWN                WSAENETDOWN
#define ENETUNREACH             WSAENETUNREACH
#define ENETRESET               WSAENETRESET
#define ECONNABORTED            WSAECONNABORTED
#define ECONNRESET              WSAECONNRESET
#define ENOBUFS                 WSAENOBUFS
#define EISCONN                 WSAEISCONN
#define ENOTCONN                WSAENOTCONN
#define ESHUTDOWN               WSAESHUTDOWN
#define ETOOMANYREFS            WSAETOOMANYREFS
#define ETIMEDOUT               WSAETIMEDOUT
#define ECONNREFUSED            WSAECONNREFUSED
#define ELOOP                   WSAELOOP
#define ENAMETOOLONG            WSAENAMETOOLONG
#define EHOSTDOWN               WSAEHOSTDOWN
#define EHOSTUNREACH            WSAEHOSTUNREACH
#define ENOTEMPTY               WSAENOTEMPTY
#define EPROCLIM                WSAEPROCLIM
#define EUSERS                  WSAEUSERS
#define EDQUOT                  WSAEDQUOT
#define ESTALE                  WSAESTALE
#define EREMOTE                 WSAEREMOTE

/* Socket function prototypes */

#ifdef __cplusplus
extern "C" {
#endif

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

int PASCAL FAR bind (SOCKET s, const struct sockaddr FAR *addr, int namelen);

int PASCAL FAR closesocket (SOCKET s);

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

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

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

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

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

u_long PASCAL FAR htonl (u_long hostlong);

u_short PASCAL FAR htons (u_short hostshort);

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

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

```
int PASCAL FAR listen (SOCKET s, int backlog);

u_long PASCAL FAR ntohl (u_long netlong);

u_short PASCAL FAR ntohs (u_short netshort);

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

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

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

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

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

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

int PASCAL FAR shutdown (SOCKET s, int how);

SOCKET PASCAL FAR socket (int af, int type, int protocol);

/* Database function prototypes */

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

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

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

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

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

struct protoent FAR * PASCAL FAR getprotobynumber(int proto);

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

/* Microsoft Windows Extension function prototypes */

int PASCAL FAR WSAStartup(WORD wVersionRequired, LPWSADATA lpWSAData);

int PASCAL FAR WSACleanup(void);

void PASCAL FAR WSASetLastError(int iError);

int PASCAL FAR WSAGetLastError(void);

BOOL PASCAL FAR WSAIsBlocking(void);

int PASCAL FAR WSAUnhookBlockingHook(void);

FARPROC PASCAL FAR WSASetBlockingHook(FARPROC lpBlockFunc);

int PASCAL FAR WSACancelBlockingCall(void);

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

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

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

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

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

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

int PASCAL FAR WSACancelAsyncRequest(HANDLE hAsyncTaskHandle);

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

#ifdef __cplusplus
}
#endif

/* Microsoft Windows Extended data types */
typedef struct sockaddr SOCKADDR;
typedef struct sockaddr *PSOCKADDR;
typedef struct sockaddr FAR *LPSOCKADDR;

typedef struct sockaddr_in SOCKADDR_IN;
typedef struct sockaddr_in *PSOCKADDR_IN;
typedef struct sockaddr_in FAR *LPSOCKADDR_IN;

typedef struct linger LINGER;
typedef struct linger *PLINGER;
typedef struct linger FAR *LPLINGER;

typedef struct in_addr IN_ADDR;
typedef struct in_addr *PIN_ADDR;
typedef struct in_addr FAR *LPIN_ADDR;

typedef struct fd_set FD_SET;
typedef struct fd_set *PFD_SET;
typedef struct fd_set FAR *LPFD_SET;

typedef struct hostent HOSTENT;
typedef struct hostent *PHOSTENT;
typedef struct hostent FAR *LPHOSTENT;

typedef struct servent SERVENT;
typedef struct servent *PSERVENT;
typedef struct servent FAR *LPSERVENT;

typedef struct protoent PROTOENT;
typedef struct protoent *PPROTOENT;
typedef struct protoent FAR *LPPROTOENT;

typedef struct timeval TIMEVAL;
typedef struct timeval *PTIMEVAL;
typedef struct timeval FAR *LPTIMEVAL;

/*
 * Windows message parameter composition and decomposition
 * macros.
 *
 * WSAMAKEASYNCREPLY is intended for use by the Windows Sockets implementation
 * when constructing the response to a WSAAsyncGetXByY() routine.
 */
#define WSAMAKEASYNCREPLY(buflen,error)     MAKELONG(buflen,error)
/*
 * WSAMAKESELECTREPLY is intended for use by the Windows Sockets implementation
 * when constructing the response to WSAAsyncSelect().
 */
#define WSAMAKESELECTREPLY(event,error)     MAKELONG(event,error)
/*
 * WSAGETASYNCBUFLEN is intended for use by the Windows Sockets application
 * to extract the buffer length from the lParam in the response
```

```
 * to a WSAGetXByY().
 */
#define WSAGETASYNCBUFLEN(lParam)           LOWORD(lParam)
/*
 * WSAGETASYNCERROR is intended for use by the Windows Sockets application
 * to extract the error code from the lParam in the response
 * to a WSAGetXByY().
 */
#define WSAGETASYNCERROR(lParam)            HIWORD(lParam)
/*
 * WSAGETSELECTEVENT is intended for use by the Windows Sockets application
 * to extract the event code from the lParam in the response
 * to a WSAAsyncSelect().
 */
#define WSAGETSELECTEVENT(lParam)           LOWORD(lParam)
/*
 * WSAGETSELECTERROR is intended for use by the Windows Sockets application
 * to extract the error code from the lParam in the response
 * to a WSAAsyncSelect().
 */
#define WSAGETSELECTERROR(lParam)           HIWORD(lParam)

#endif  /* _WINSOCKAPI_ */
```

Appendix B. Notes for Windows Sockets Suppliers

B.1 Introduction

A Windows Sockets implementation must implement ALL the functionality described in the Windows Sockets documentation. Validation of compliance is discussed in section B.8.

Windows Sockets Version 1.1 implementations must support both TCP and UDP type sockets. An implementation may support raw sockets (of type SOCK_RAW), but their use is deprecated.

Certain APIs documented above have special notes for Windows Sockets implementors. A Windows Sockets implementation should pay special attention to conforming to the API as documented. The Special Notes are provided for assistance and clarification.

B.2 Windows Sockets Components

B.2.1 Development Components

The Windows Sockets development components for use by Windows Sockets application developers will be provided by each Windows Sockets supplier. These Windows Sockets development components are:

Component	Description
Windows Sockets Documentation	This document
WINSOCK.LIB file	Windows Sockets API Import Library
WINSOCK.H file	Windows Sockets Header File
NETDB.H file	Berkeley Compatible Header File
ARPA/INET.H file	Berkeley Compatible Header File
SYS/TIME.H file	Berkeley Compatible Header File
SYS/SOCKET.H file	Berkeley Compatible Header File
NETINET/IN.H file	Berkeley Compatible Header File

B.2.2 Run Time Components

The run time component provided by each Windows Sockets supplier is:

Component	Description
WINSOCK.DLL	The Windows Sockets API implementation DLL

B.3 Multithreadedness and blocking routines.

Data areas returned by, for example, the getXbyY() routines MUST be on a per thread basis.

Note that an application MUST be prevented from making multiple nested Windows Sockets function calls. Only one outstanding function call will be allowed for a particular task. Any Windows Sockets call performed when an existing blocking call is already outstanding will fail with an error code of WSAEINPROGRESS. There are two exceptions to this restriction: WSACancelBlockingCall() and WSAIsBlocking() may be called at any time. Windows Sockets suppliers should note that although preliminary drafts of this specification indicated that the restriction only applied to blocking function calls, and that it would be permissible to make non-blocking calls while a blocking call was in progress, this is no longer true.

Regarding the implementation of blocking routines, the solution in Windows Sockets is to simulate the blocking mechanism by having each routine call PeekMessage() as it waits for the completion of its operation. In anticipation of this, the function WSASetBlockingHook() is provided to allow the programmer to define a special routine to be called instead of the default PeekMessage() loop. The blocking hook functions are discussed in more detail in 4.3.13, **WSASetBlockingHook()**.

B.4 Database Files

The database routines in the **getXbyY()** family (**gethostbyaddr()**, etc.) were originally designed (in the first Berkeley UNIX releases) as mechanisms for looking up information in text databases. A Windows Sockets supplier may choose to employ local files OR a name service to provide some or all of this information. If local files exist, the format of the files must be identical to that used in BSD UNIX, allowing for the differences in text file formats.

B.5 FD_ISSET

It is necessary to implement the FD_ISSET Berkeley macro using a supporting function: **__WSAFDIsSet()**. It is the responsibility of a Windows Sockets implementation to make this available as part of the Windows Sockets API. Unlike the other functions exported by a Windows Sockets DLL, however, this function is not intended to be invoked directly by Windows Sockets applications: it should be used only to support the FD_ISSET macro. The source code for this function is listed below:

```
int FAR
__WSAFDIsSet(SOCKET fd, fd_set FAR *set)
{
    int i = set->fd_count;

    while (i--)
      if (set->fd_array[i] == fd)
          return 1;

    return 0;
}
```

B.6 Error Codes

In order to avoid conflict between various compiler environments Windows Sockets implementations MUST return the error codes listed in the API specification, using the manifest constants beginning with "WSA". The Berkeley-compatible error code definitions are provided solely for compatibility purposes for applications which are being ported from other platforms.

B.7 DLL Ordinal Numbers

The **winsock.def** file for use by every Windows Sockets implementation is as follows. Ordinal values starting at 1000 are reserved for Windows Sockets implementors to use for exporting private interfaces to their DLLs. A Windows Sockets implementation must not use any ordinals 999 and below except for those APIs listed below. An application which wishes to work with any Windows Sockets DLL must use only those routines listed below; using a private export makes an application dependent on a particular Windows Sockets implementation.

```
;
;         File: winsock.def
;       System: MS-Windows 3.x
;      Summary: Module definition file for Windows Sockets DLL.
;

LIBRARY         WINSOCK         ; Application's module name

DESCRIPTION     'BSD Socket API for Windows'

EXETYPE         WINDOWS         ; required for all windows applications

STUB            'WINSTUB.EXE'   ; generates error message if application
                                ; is run without Windows

;CODE can be FIXED in memory because of potential upcalls
```

```
CODE            PRELOAD         FIXED

;DATA must be SINGLE and at a FIXED location since this is a DLL
DATA            PRELOAD         FIXED           SINGLE

HEAPSIZE        1024
STACKSIZE       16384

; All functions that will be called by any Windows routine
; must be exported.  Any additional exports beyond those defined
; here must have ordinal numbers 1000 or above.

EXPORTS
        accept                          @1
        bind                            @2
        closesocket                     @3
        connect                         @4
        getpeername                     @5
        getsockname                     @6
        getsockopt                      @7
        htonl                           @8
        htons                           @9
        inet_addr                       @10
        inet_ntoa                       @11
        ioctlsocket                     @12
        listen                          @13
        ntohl                           @14
        ntohs                           @15
        recv                            @16
        recvfrom                        @17
        select                          @18
        send                            @19
        sendto                          @20
        setsockopt                      @21
        shutdown                        @22
        socket                          @23

        gethostbyaddr                   @51
        gethostbyname                   @52
        getprotobyname                  @53
        getprotobynumber                @54
        getservbyname                   @55
        getservbyport                   @56
        gethostname                     @57

        WSAAsyncSelect                  @101
        WSAAsyncGetHostByAddr           @102
        WSAAsyncGetHostByName           @103
        WSAAsyncGetProtoByNumber        @104
        WSAAsyncGetProtoByName          @105
        WSAAsyncGetServByPort           @106
        WSAAsyncGetServByName           @107
        WSACancelAsyncRequest           @108
        WSASetBlockingHook              @109
        WSAUnhookBlockingHook           @110
        WSAGetLastError                 @111
        WSASetLastError                 @112
        WSACancelBlockingCall           @113
        WSAIsBlocking                   @114
        WSAStartup                      @115
        WSACleanup                      @116

        __WSAFDIsSet                    @151

        WEP                             @500    RESIDENTNAME

;eof
```

B.8 Validation Suite

The Windows Sockets API Tester (WSAT) to ensure Windows Sockets compatibility between Windows Sockets DLL implementations is currently in beta test. This beta version includes functionality testing of

the Windows Sockets interface and is supported by a comprehensive scripting language. The final version of WSAT will be available in Spring 1993. If you wish to receive the tester or more information on the beta, send email to wsat@microsoft.com.

Appendix C. For Further Reference

This specification is intended to cover the Windows Sockets interface to TCP/IP in detail. Many details of TCP/IP and Windows, however, are intentionally omitted in the interest of brevity, and this specification often assumes background knowledge of these topics. For more information, the following references may be helpful:

Braden, R.[1989], *RFC 1122, Requirements for Internet Hosts--Communication Layers*, Internet Engineering Task Force.

Comer, D. [1991], *Internetworking with TCP/IP Volume I: Principles, Protocols, and Architecture,* Prentice Hall, Englewood Cliffs, New Jersey.

Comer, D. and Stevens, D. [1991], *Internetworking with TCP/IP Volume II: Design, Implementation, and Internals,* Prentice Hall, Englewood Cliffs, New Jersey.

Comer, D. and Stevens, D. [1991], *Internetworking with TCP/IP Volume III: Client-Server Programming and Applications,* Prentice Hall, Englewood Cliffs, New Jersey.

Leffler, S. et al., *An Advanced 4.3BSD Interprocess Communication Tutorial.*

Petzold, C. [1992], *Programming Windows 3.1,* Microsoft Press, Redmond, Washington.

Stevens, W.R. [1990], *Unix Network Programming,* Prentice Hall, Englewood Cliffs, New Jersey.

Appendix D. Background Information

D.1 Legal Status of Windows Sockets

The copyright for the Windows Sockets specification is owned by the specification authors listed on the title page. Permission is granted to redistribute this specification in any form, provided that the contents of the specification are not modified. Windows Sockets implementors are encouraged to include this specification with their product documentation.

The Windows Sockets logo on the title page of this document is meant for use on both Windows Sockets implementations and for applications that use the Windows Sockets interface. Use of the logo is encouraged on packaging, documentation, collateral, and advertising. The logo is available on microdyne.com in pub/winsock as winsock.bmp. The suggested color for the logo's title bar is blue, the electrical socket grey, and the text and outline black.

D.2 The Story Behind the Windows Sockets Icon

(by Alistair Banks, Microsoft Corporation)

We thought we'd do a "Wind Sock" at one stage--but you try to get that into 32x32 bits! It would have had to look wavy and colorful, and... well, it just didn't work. Also, our graphics designers have "opinions" about the icons truly representing what they are--people would have thought this was "The colorful wavy tube specification 1.0!"

I tried to explain "API" "Programming Interface" to the artist--we ended up with toolbox icons with little flying windows

Then we came to realise that we should be going after the shortened form of the name, rather the name in full... Windows Sockets... And so we went for that - so she drew (now remember I'm English and you're probably American) "Windows Spanner", a.k.a. a socket wrench. In the U.S. you'd have been talking about the "Windows Socket spec" OK, but in England that would have been translated as "Windows Spanner Spec 1.0" - so we went to Electrical sockets - well the first ones came out looking like "Windows Pignose Spec 1.0"!!!!

So how do you use 32x32, get an international electrical socket! You take the square type (American & English OK, Europe & Australia are too rounded)--you choose the American one, because it's on the wall in front of you (and it's more compact (but less safe, IMHO) and then you turn it upside down, thereby compromising its nationality!

[IMHO = "In My Humble Opinion"--ed.]