WINSOCK

Origins of Windows Sockets

Background

Early Microsoft operating systems, both MS-DOS and Windows, offered limited networking capability, chiefly based on NetBIOS (a technology that Microsoft adopted from IBM). In particular, Microsoft completely ignored the TCP/IP protocol stack. A number of university groups and commercial vendors, including the PC/IP group at MIT, FTP Software, Sun Microsystems, Ungermann-Bass, and Excelan, introduced TCP/IP products for MS-DOS, often as part of a hardware/software bundle. When Microsoft Windows was released, these vendors were joined by others such as Distinct and NetManage in offering TCP/IP for Windows. Even Microsoft offered a limited-function product.

The drawback faced by all of these vendors was that each of them used their own API. Without a single standard programming model, it was difficult to persuade independent software developers to create networking applications, and end users were wary of getting locked in to a single vendor.

Earlier Efforts

There had been a number of successful standardization efforts in the PC networking area over the years. The first of these was a program sponsored by the US Air Force to develop RFC1001/1002, a NetBIOS implementation running over TCP/IP. A second was the Crynwr packet driver effort initiated by FTP Software and led by Russ Nelson.

Finally...

WinSock was proposed by Martin Hall of JSB Software (later Stardust Technologies) at the Interop in October 1991, during a "Birds of a Feather" session. The first edition of the specification was authored by Martin Hall, Mark Towfiq of Microdyne (later Sun Microsystems), Geoff Arnold of Sun Microsystems, and Henry Sanders of Microsoft, with assistance from many others.

There was some discussion about how best to address the copyright, intellectual property, and potential anti-trust issues, and consideration was given to working through the IETF or establishing a non-profit foundation. In the end, it was decided that the specification would simply be copyrighted by the four authors as (unaffiliated) individuals.

The WinSock Specification

The Windows Sockets specification defines a network programming interface for Microsoft Windows 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.

- The Winsock specification defines two interfaces:
  - → the API used by application developers, and
  - → the SPI, which provides a means for network software developers to add new protocol modules to the system.
- Each interface represents a contract.
  - → The API guarantees that a conforming application will function correctly with a conformant protocol implementation from any network software vendor.
  - → The SPI contract guarantees that a conforming protocol module may be added to Windows and will thereby be usable by an API-conformant application.
- Although these contracts were important when

Winsock was first released, they are now of only academic interest. Microsoft has shipped a high-quality TCP/IP stack with all recent versions of Windows, and there are no significant independent alternatives. Nor has there been significant interest in implementing protocols other than TCP/IP.

Winsock is based on BSD sockets, but provides additional functionality to allow the API to comply with the standard Windows programming model. The Winsock API covered almost all the features of the BSD sockets API, but there were some unavoidable obstacles which mostly arose out of fundamental differences between Windows and Unix.

However it was a design goal of Winsock that it should be relatively easy for developers to port socket-based applications from Unix to Windows. It was not considered sufficient to create an API which was only useful for newly-written Windows programs. For this reason, Winsock included a number of elements which were designed to facilitate porting. For example, Unix applications were able to use the same `errno` variable to record both networking errors and errors detected within standard C library functions. Since this was not possible in Windows, Winsock introduced a dedicated function, `WSAGetLastError()`, to retrieve error information. Such mechanisms were helpful, but application porting remained extremely complex. Many "traditional" TCP/IP applications had been implemented by using system features specific to Unix, such as pseudo terminals and the fork system call, and reproducing such functionality in Windows was problematic. Within a relatively short time, porting gave way to the development of dedicated Windows applications.

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.

The Winsock 2.0 architecture

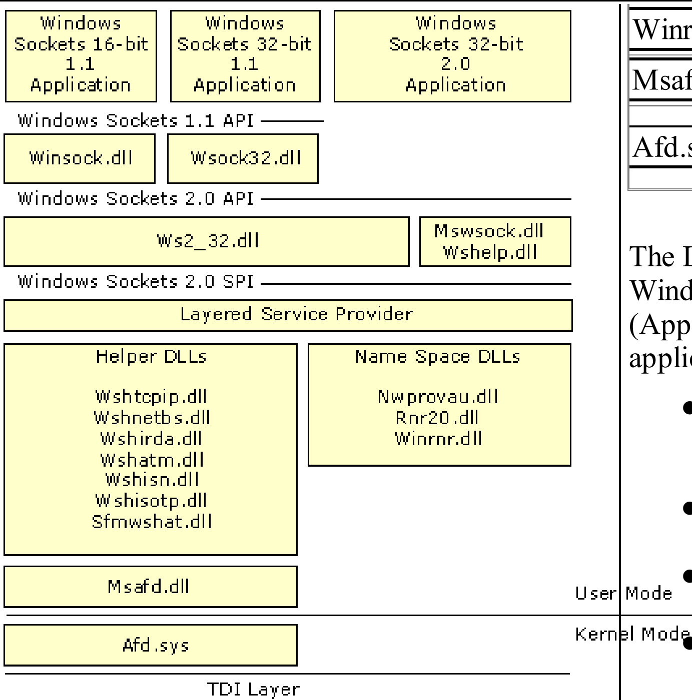
Winsock (Windows Socket) API is an application program interface (API) specification that defines how a windows network application should access underlying TCP/IP network services.

Winsock performs the following:

- Provides a familiar networking API for programmers using Windows or UNIX.
- Offers binary compatibility between the heterogeneous, Windows-based TCP/IP stack and utility vendors.
- Supports both connection-oriented and connectionless protocols.

The current version of Winsock API is Version 2.2.2, which has several important features:

- Multiple Protocol support
- Transport Protocol Independence: Choose protocol by the services they provide
- Multiple Namespaces: Select the protocol you want to resolve hostnames, or locate services
- Scatter and Gather: Receive and send, to and from multiple buffers
- Overlapped I/O and Event Objects: Utilize Win32 paradigms for enhanced throughput
- Quality of Service: Negotiate and keep track of bandwidth per socket
- Multipoint and Multicast: Protocol independent APIs and protocol specific APIs
- Conditional Acceptance: Ability to reject or defer a connect request before it occurs
- Connect and Disconnect data: For transport protocols that support it (NOTE: TCP/IP does not)
- Socket Sharing: Two or more processes can share a socket handle
- Vendor IDs and a mechanism for vendor extensions: Vendor specific APIs can be added
- Layered Service Providers: The ability to add services to existing transport providers



Winsock Files

A list of files that Winsock uses to function. The table lists the files in order of the layer that they support and gives a brief description of their function.

Winsock Files

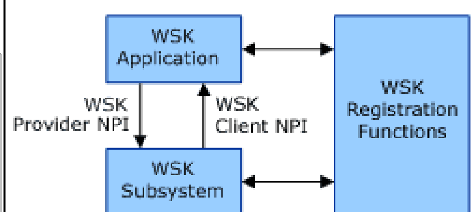
Winsock DLLs	Description
Winsock.dll	16-bit Winsock 1.1
Wsock32.dll	32-bit Winsock 1.1
Ws2_32.dll	Main Winsock 2.0
Mswsock.dll	Microsoft extensions to Winsock. Mswsock.dll is an API that supplies services that are not part of Winsock.
Ws2help.dll	Platform-specific utilities. Ws2help.dll supplies operating system–specific code that is not part of Winsock.
Wshtcpip.dll	Helper for TCP
Wshnetbs.dll	Helper for NetBT
Wshirda.dll	Helper for IrDA
Wshatm.dll	Helper for ATM
Wshisn.dll	Helper for Netware
Wshisotp.dll	Helper for OSI transports
Sfmwshat.dll	Helper for Macintosh
Nwprovau.dll	Name resolution provider for IPX
Rnr20.dll	Main name resolution
Winrnr.dll	LDAP name resolution
Msafd.dll	Winsock interface to kernel
Afd.sys	Winsock kernel interface to TDI transport protocols

The DLL (Dynamic Link Library) files installed on a Windows XP system to provide the Winsock API (Application Programming Interface) to all Winsock applications:

- WS2_32.DLL - Providing Winsock 2 32-bit API and running on top of a collection of Winsock 2 SPIs (Service Provider Interfaces).
- WSOCK32.DLL - Providing Winsock 1.1 32-bit API and running on top of the Winsock 2 API.
- WINSOCK.DLL - Providing Winsock 1.1 16-bit API and running on top of the Winsock 2 API.
- mswsock.dll is the DLL (Dynamic Link Library) file that implements the Winsock 2 SPI (Service Provider Interface) as the Basic Server Provider in the Winsock 2 SPI architecture as described in the previous section.

Winsock Kernel Architecture

The architecture of Winsock Kernel (WSK) is shown in the following diagram.



WSK applications discover and attach to the WSK subsystem by using a set of WSK registration functions. Applications can use these functions to dynamically detect when the WSK subsystem is available and to exchange dispatch tables that constitute the provider and client side implementations of the WSK NPI.

What is a DLL?

In a nut shell, a dynamic link library (DLL) is a collection of small programs, which can be called upon when needed by the executable program (EXE) that is running. The DLL lets the executable communicate with a specific device such as a printer or may contain source code to do particular functions.

An example would be if the program (exe) needs to get the free space of your hard drive. It can call the DLL file that contains the function with parameters and a call function. The DLL will then tell the executable the free space. This allows the executable to be smaller in size and not have to write the function that has already exists.

This allows any program the information about the free space, without having to write all the source code and it saves space on your hard drive as well. When a DLL is used in this fashion are also known as shared files.

Advantage of DLL

The advantage of DLL files is that, because they do not get loaded into random access memory (RAM) together with the main program, space is saved in RAM. When and if a DLL file is called, then it is loaded. For example, you are editing a Microsoft Word document, the printer DLL file does not need to be loaded into RAM. If you decide to print the document, then the printer DLL file is loaded and a call is made to print.

Uses fewer resources
When multiple programs use the same library of functions, a DLL can reduce the duplication of code that is loaded on the disk and in physical memory.

Promotes modular architecture
A DLL helps promote developing modular programs. This helps you develop large programs that require multiple language versions or a program that requires modular architecture.

Eases deployment and installation
When a function within a DLL needs an update or a fix, the deployment and installation of the DLL does not require the program to be relinked with the DLL. Additionally, if multiple programs use the same DLL, the multiple programs will all benefit from the update or the fix.

All in all a DLL is an executable file that cannot run on its own, it can only run from inside an executable file.

To do load a DLLl file, an executable needs to declare the DLL function. A DLL may have many different functions in it. Then when needed the call is made with the required parameters.

The following list describes some of the files that are implemented as DLLs in Windows operating systems:

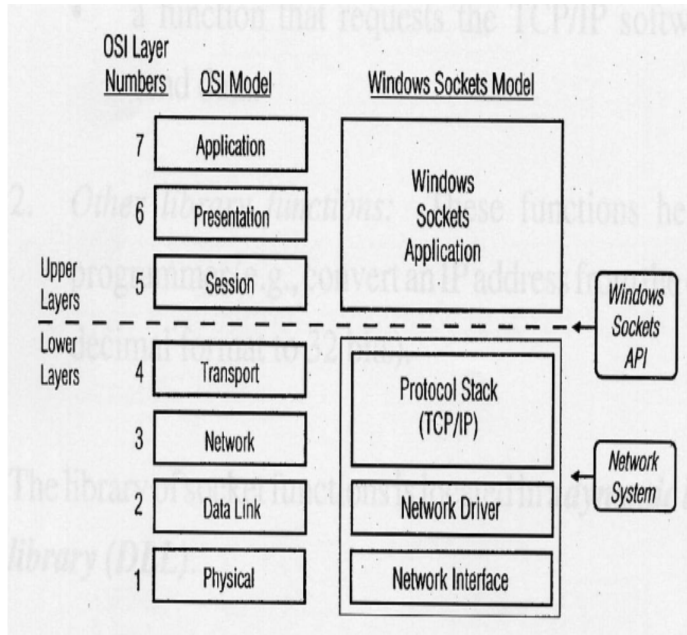
- **ActiveX Controls (.ocx) files**
  An example of an ActiveX control is a calendar control that lets you select a date from a calendar.
- **Control Panel (.cpl) files**
  An example of a .cpl file is an item that is located in Control Panel. Each item is a specialized DLL.
- **Device driver (.drv) files**
  An example of a device driver is a printer driver that controls the printing to a printer.

DLL dependencies

When a program or a DLL uses a DLL function in another DLL, a dependency is created. Therefore, the program is no longer self-contained, and the program may experience problems if the dependency is broken. For example, the program may not run if one of the following actions occurs:

- A dependent DLL is upgraded to a new version.
- A dependent DLL is fixed.
- A dependent DLL is overwritten with an earlier version.
- A dependent DLL is removed from the computer.

Role of WinSock:



– WinSock provides a library of functions. These functions can be classified into two types:
• Primary socket functions: These functions perform specific operations to interact with the TCP/IP protocol software. Examples:
– a function that requests the TCP/IP software to establish a TCP connection
to a remote server;
– a function that requests the TCP/IP software to send data.
Other library functions: These functions help the programmer (e.g., convert an IP address from the dotted decimal format to 32 bits).
– The library of socket functions is located in a dynamic linked library (DLL).
• A DLL is a library that is loaded into main memory only when the library is first used.
- WinSock is an interface but it is not a protocol.
• If the client and the server use the same protocol suite (TCP/IP), then they can communicate even if they use different application program interfaces:
– There are cases where an interface to a protocol suite is adopted to another protocol suite.• e.g., WinSock API for the IPX/SPX protocol suite
– IPX (Internetwork Packet Exchange) is a networking protocol from Novell that interconnects networks that use Novell's Netware clients and servers. IPX is a datagram protocol. IPX works at the Network layer of communication protocols and is connectionless

Windows V1 vs V2

Winsock2 is completely backwards compatible with the original winsock
Winsock2 introduces some new functions for new networking protocols (like bluetooth). It also introduces something called LSP which layers all winsock functions on top of the base networking functions.
you can create your own layers which will in turn get called by the winsock implementation when any application calls the function you layered.
winsock.h should be used with wsock32.lib and winsock2.h should be used with ws2_32.lib

It added support for protocol-independent name resolution, asynchronous operations with event-based notifications and completion routines, layered protocol implementations, multicasting, and quality of service. It also formalized support for multiple protocols, including IPX/SPX and DECnet. The new specification allowed sockets to be optionally shared between processes, incoming connection requests to be conditionally accepted, and certain operations to be performed on socket groups rather than individual sockets. Although the new specification differed substantially from Winsock 1, it provided source- and binary-level compatibility with the Winsock 1.1 API. One of the lesser known additions was the Service Provider Interface (SPI) API and Layered Service Providers.

Sockets - 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.

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.

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.

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*. 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 to indicate the point at which the out-of-band data was sent. An application can use the *SIOCATMARK* ioctlsocket() command 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.

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 neighbours.

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.

Byte Ordering

The Intel byte ordering is like that of the DEC VAX, and therefore differs from the Internet and 68000-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.

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 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 will call the blocking hook function 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.

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

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.

Hooking Blocking Methods

As noted in Blocking/Non blocking & Data Volatility, 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.

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 Error Codes

Winsock error codes

10004 WSAEINTR Interrupted function call
10009 WSAEBADF WSAEBADF
10013 WSAEACCES WSAEACCES
10014 WSAEFAULT Bad address
10022 WSAEINVAL Invalid argument
10024 WSAEMFILE Too many open files
10035 WSAEWOULDBLOCK Operation would block `
10036 WSAEINPROGRESS Operation now in progress
10037 WSAEALREADY Operation already in progress
10038 WSAENOTSOCK Socket operation on non-socket
10039 WSAEDESTADDRREQ Destination address required
10040 WSAEMSGSIZE Message too long
10041 WSAEPROTOTYPE Protocol wrong type for socket
10042 WSAENOPROTOOPT Bad protocol option
10043 WSAEPROTONOSUPPORT Protocol not supported
10044 WSAESOCKTNOSUPPORT Socket type not supported
10045 WSAEOPNOTSUPP Operation not supported
10046 WSAEPFNOSUPPORT Protocol family not supported
10047 WSAEAFNOSUPPORT Address family not supported by protocol family
10048 WSAEADDRINUSE Address already in use
10049 WSAEADDRNOTAVAIL Cannot assign requested address
10050 WSAENETDOWN Network is down
10051 WSAENETUNREACH Network is unreachable
10052 WSAENETRESET Network dropped connection on reset
10053 WSAECONNABORTED Software caused connection abort
10054 WSAECONNRESET Connection reset by peer
10055 WSAENOBUFS No buffer space available
10056 WSAEISCONN Socket is already connected
10057 WSAENOTCONN Socket is not connected
10058 WSAESHUTDOWN Cannot send after socket shutdown
10059 WSAETOOMANYREFS WSAETOOMANYREFS
10060 WSAETIMEDOUT Connection timed out
10061 WSAECONNREFUSED Connection refused
10062 WSAELOOP WSAELOOP
10063 WSAENAMETOOLONG WSAENAMETOOLONG
10064 WSAEHOSTDOWN Host is down
10065 WSAEHOSTUNREACH No route to host
10066 WSAENOTEMPTY WSAENOTEMPTY
10067 WSAEPROCLIM Too many processes
10068 WSAEUSERS WSAEUSERS
10069 WSAEDQUOT WSAEDQUOT
10070 WSAESTALE WSAESTALE
10071 WSAEREMOTE WSAEREMOTE
10091 WSASYSNOTREADY Network subsystem is unavailable
10092 WSAVERNOTSUPPORTED WINSOCK.DLL version out of range
10093 WSANOTINITIALISED Successful WSAStartup() not yet performed
10101 WSAEDISCON WSAEDISCON
10102 WSAENOMORE WSAENOMORE
10103 WSAECANCELLED WSAECANCELLED
10104 WSAEINVALIDPROCTABLE WSAEINVALIDPROCTABLE
10105 WSAEINVALIDPROVIDER WSAEINVALIDPROVIDER
10106 WSAEPROVIDERFAILEDINIT WSAEPROVIDERFAILEDINIT
10107 WSASYSCALLFAILURE WSASYSCALLFAILURE
10108 WSASERVICE_NOT_FOUND WSASERVICE_NOT_FOUND
10109 WSATYPE_NOT_FOUND WSATYPE_NOT_FOUND
10110 WSA_E_NO_MORE WSA_E_NO_MORE
10111 WSA_E_CANCELLED WSA_E_CANCELLED
10112 WSAEREFUSED WSAEREFUSED
11001 WSAHOST_NOT_FOUND Host not found
11002 WSATRY_AGAIN Non-authoritative host not found
11003 WSANO_RECOVERY This is a non-recoverable error
11004 WSANO_DATA Valid name, no data record of requested type

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.

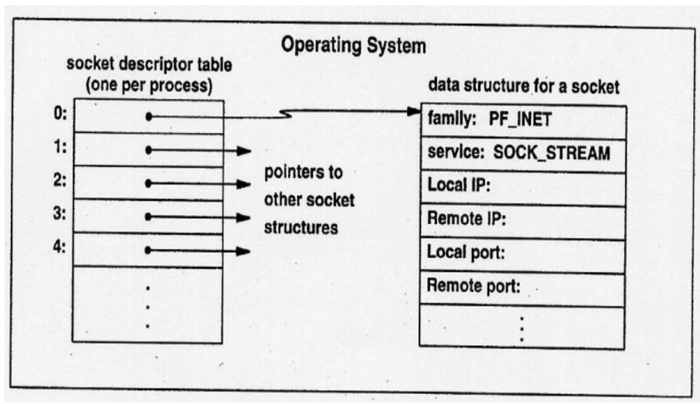
There are (at least) two ways to accomplish this. The simplest method is for the intermediate DLL to have calls similiar to WSAStartup() and WSACleanup() that applications call as appropriate. The DLL would then call WSAStartup() or WSACleanup() from within these routines. Another mechanism is for the 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.

Socket Descriptor

– Each socket is identified by an integer called socket descriptor, which is an unsigned integer.
– A process may open multiple sockets for multiple concurrent communication sessions (e.g., a web server is serving multiple browsers simultaneously).
• Windows keeps a table of socket descriptors for each process.
– Each socket descriptor is associated with a pointer, which points to a data structure that holds the information about the communication session of that socket.
– The data structure contains many fields, and they will be filled as the application calls additional WinSock functions.



Data types for TCP/IP endpoint address

```
struct  sockaddr_in {          /* struct to hold an address        */
    u_short sin_family;        /* type of address(always AF_INET)  */
    u_short sin_port;          /* protocol port number             */
    struct  in_addr sin_addr;  /* IP address (declared to be       */
                               /*  u_long on some systems)         */
    char    sin_zero[8];       /* unused (set to zero)             */
};
```

```
struct  in_addr {
   u_long    s_addr ;  /*  IP
  address  */
};
```

Windows Programming APIs

Address Structure

Every computers in the network are assigned an IP address that is represented as a 32-bit quantity, formally known as an IP version 4 (IPv4) address. When a client wants to communicate with a server through TCP or UDP, it must specify the server's IP address along with a service port number. Apart from that, when servers want to listen for incoming client requests, they must specify an IP address and a port number. In Winsock, applications specify IP addresses and service port information through the SOCKADDR_IN structure, which is defined as

```
struct sockaddr_in              struct in_addr
{                               {
  short   sin_family;           u_long s_addr;
  u_short  sin_port;            };
  struct  in_addr
sin_addr;
  char    sin_zero[8];
};
```

-The "sin_family" field must be set to "AF_INET", which tells Winsock that the IP address family is -being used.
-The "sin_port" defines which TCP or UDP communication port will be used to identify a server service.
- The "sin_addr" is used for storing an IP address as a 4 byte quantity, which is an unsigned long integer data type. Depending on how this field is used, it can represent a local or remote IP address.
- The "sin_zero" enables the "SOCKADDR_IN" structure the same size as the SOCKADDR" structure.
A useful support function "inet_addr" converts a dotted IP address to a 32-bit unsigned long integer quantity. The "inet_addr" function is defined as

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

Name Resolution

Winsock provides two support functions that help to resolve a host name to an IP address. The Window Sockets "gethostbyname" and "WSAsyncGetHostByName" API functions retrieve host information corresponding to a host name from a host database. Both functions return a "HOSTENT" structure that is defined as:

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

The "h_name" is the official name of the host. If the network uses the DNS, it is the Fully Qualified Domain Name (FQDN) that causes the name server to return a reply. But if the network uses a local "hosts" file, it is the first entry after the IP address. The "h_aliases" is a null_terminated array alternative name for the host. The "h_addrtype" represents the address family being returned. The "h_length" defines the length in bytes of each address in the "h_addr_list". The "h_addr_list" field is a null-terminated array of IP addresses fro the host. Normally applications use the first address in the array. But if more than one address is returned, applications should randomly choose an available address.The "gethostbyname" API function is defined as

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

Here, "name" represents a friendly name of the host being looked up. If this function succeeds, a pointer to a "HOSTENT" structure is returned. The "WSAAsyncGetHostByName" API function is an asynchronous version of the "gethostbyname" function that uses Windows messages to inform an application when this function completes.
"WSAAsyncGetHostByName" is defined as
HANDLE WSAAsyncGetHostByName(HWND hWnd, Unsigned int wMsg, const char FAR * name, char FAR * buf, int buflen);

Here, "hWnd" is the handle of the window that will receive a message when the asynchronous request completes. The "wMsg" is the Windows message to be received when the asynchronous requests completes. The "name" parameter represents a user-friendly name of the host being looked up. The "buf" parameter is a pointer to the data area to receive the "HOSTENT" data.

Port numbers:

Apart from the IP address of a remote computer, an application must know the service's port number to communicate with a service running on a local or remote computer. When using TCP and UDP, applications must decide which ports they plan to communicate over. It is possible to retrieve port numbers for well-known services by calling the "getservbyname" and "WSAAsynGetServByName" functions. These functions retrieve static information from a file named "services". In Windows95 and Windows98, the services file is located under %WINDOWS% and in WindowsNT and Windows2000; it is located under %WINDOWS%\System32\Drivers\etc. The "getservbyname" function is defined as

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

Here, the "name" represents the name of the service you are looking for. The "proto" parameter optionally points to a string that indicates the protocol that the service in "name" is registered under. The "WSAAsynGetSrvByName" function is asynchronous version of "getservbyname".

Initializing Winsock:

Before calling a Winsock function, it is necessary to load the correct version of the Winsock library. The Winsock initialization routine is WSAStartup, defined as

int WSAStartup(WORD wVersionRequested, LPWSADATA lpWSAData)

The first parameter is the version of the Winsock library that is required to load. For current Win32 platforms, the latest Winsock 2 library is version 2.2. If this version to be used, either the value (0x0202) is to be specified, or the macro MAKEWORD(2, 2) is to be used. The high-order byte specifies the minor version number, while the low-order byte specifies the major version number.

The second parameter is a structure WSADATA, which is returned upon completion. WSADATA contains information about the version of Winsock that WSAStartup loaded. It is defined as

```
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;
};
```

It should be noted that when the Winsock functions are no longer required to be called, the companion routine WSACleanup unloads the library and frees any resources. This function is defined as

```
int WSACleanup (void);
```

For each call to WSAStartup, a matching call to WSACleanup is required as each startup call increments the reference count to the loaded Winsock DLL, requiring an equal number of calls to WSACleanup to decrement the count.

Windows Sockets:

A socket is a handle to a transport provider. In Win32, a socket is not the same thing as a file descriptor and therefore is a separate type, SOCKET. Two functions create a socket:

SOCKET WSASocket (int af,   int type,   int protocol,   LPWSAPROTOCOL_INFO lpProtocolInfo, GROUP g,   DWORD dwFlags);

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

Here, “af” is the address family of the protocol. For instance, if it is required to create either a UDP or TCP socket, the constant “AF_INET” is used to indicate the Internet Protocol (IP). The “type” is the socket type of the protocol. A socket type can be one of five values: SOCK_STREAM, SOCK_DGRAM, SOCK_SEQPACKET, SOCK_RAW and SOCK_RDM. The “protocol” is used to qualify a specific transport if there are multiple entries for the given address family and socket type. The table below shows the values used for the address family, socket, and protocol fields for a given network transport.

Error Checking and Handling:

The most common return value for an unsuccessful Winsock call is “SOCKET_ERROR”. The constant “SOCKET_ERROR” actually is -1. While calling a Winsock function, if an error condition occurs, the function “WSAGetLastError” can be used to obtain a code that indicates specifically what had happened.This function is defined as:

```
int WSAGetLastError (void);
```

A call to the function after an error occurs will return an integer code for the particular error that occurred. The error codes returned from “WSAGetLastError” have predefined constant values that are declared either in Winsock.h or Winsock2.h, de pending on the version of Winsock.

bind

Once the socket of a particular protocol is created, it is compulsory to bind the socket to a well-known address. The “bind” function associates the given socket with a well-known address. This function is declared as:

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

The first parameter “s” is the socket on which it is required to wait for client connections. The second parameter is of type “struct sockaddr”, which is simply is a generic buffer. It is necessary to fill out an address buffer specific to the protocol being used and cast that as a “struct sockaddr” when calling “bind”. The Winsock header file defines the type “SOCKADDR” as “struct sockaddr”. The third parameter “namelen” is the size of the protocol-specific address structure being passed. For example, the following code illustrates how this is done on a TCP connection:

```
SOCKET   s;
struct   sockaddr_in tcpaddr;
int    port = 5150;
s = socket (AF_INET, SOCK_STREAM,
IPPROTO_TCP);

tcpaddr.sin_family   = AF_INET;
tcpaddr.sin_port   = htons(port);
tcpaddr.sin_addr.s_addr = htonl(INADDR_ANY);

bind(s, (SOCKADDR *)&tcpaddr, sizeof(tcpaddr));
```

Here, the socket is being bound to the default IP interface on port number 5150. The call to “bind” formally establishes this association of the socket with the IP interface and port. On error, “bind” returns “SOCKET_ERROR”. The most common error encountered with “bind” is “WSAEADDRINUSE”. While using TCP/IP, the “WSAEADDRINUSE” error indicates that another process is already bound to the local IP interface and port number.

listen

The “bind” function hardly associates the socket with a given address. The API function that tells a socket to wait for incoming connections is “listen”, which is defined as

int listen(SOCKET s,   int backlog);

Here, the “backlog” parameter specifies the maximum queue length for pending connections. This is important when several simultaneous requests are made to the server. For instance, let the “backlog” is set to 2, and if 3 client requests are made simultaneously, the first two will be placed in a “pending” queue so that the application can serve their requests, whereas, the third connection request will fail with “WSAECONNREFUSED”. It should be noted that once the server accepts a connection, the connection request is removed from the queue so that others can make a request. One of the most common errors associated with “listen” is “WSAEINVAL”, which usually indicates that “bind’ was not called before “listen”.accept and WSAAccept The functions “accept” or “WSAAccept” are used to accept client connections. The “accept” function is defined as

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

Here, “s” is the bound socket that is in a listening state and “sockaddr” is the address of a valid SOCKADDR_IN structure, while “addrlen” is a reference to the length of the “SOCKADDR_IN” structure. A call to “accept” serves the first connection request in the queue of pending connections. When the “accept” function returns, the “addr’ structure contains the IP address information of the client making the connection request, while the “addrlen” indicates the size of the structure. Additionally, “accept” returns a new socket descriptor that corresponds to the accepted client connection. For all subsequent operations with this client, the new socket should be used. The original listening socket is still used to accept other client connections and is still in listening mode.Winsock2 introduced the function “WSAAccept” that has the ability to conditionally accept a connection based on the return value of a condition function. The “WSAAccept” is defined as

```
SOCKET WSAAccept
  SOCKET s,
  struct  sockaddr FAR * addr,
  LPINT addrlen,
  LPCONDITIONPROC lpfnCondition,
  DWORD dwCallbackData
);
```

Here, the “lpfnCondition” argument is a pointer to a function that is called upon a request. This function determines whether to accept the client’s connection request. It is defined as

```
int CALLBACK ConditionFunc(
  LPWSABUF   lpCallerId,
  LPWSABUF   lpCallerData,
  LPQOS   lpSQOS,
  LPQOS   lpGQOS,
  LPWSABUF  lpCalleeId,
  LPWSAF  lpCalleeData,
  GROUP FAR * g,
  DWORD   dwCallbackData
);
```

Here, the “lpCallerId” is a value that contains the address of the connecting entity. The “WSABUF” structure is commonly used by many Winsock 2 functions. It is declared as

```
typedef struct __WSABUF {
  u_long   len;
  char FAR * buf;
} WSABUF, FAR * LPWSABUF;
```

Here, the “len” field refers either to the size of the buffer pointed to by the “buf” field or to the amount of data contained in the data buffer “buf”. For “lpCallerId”, the “buf” pointer points to an address structure for the given protocol on which the connection is made. The “lpCallerData” contains any connection data sent by the client along with the connection request. The next two parameters “lpSQOS” and “lpGQOS” specify any quality of service (QOS) parameter that are being requested by the client, which contains information regarding bandwidth requirements for both sending and receiving data. The “lpSOS” refers to a single connection, while “lpGQOS” is used for socket groups. The “lpCalleeId” is another “WSABUF” structure containing the local address to which the client has connected. The “lpCalleeData” is the complement of “lpCallerData”. The “lpCalleeData” parameter points to a “WSABUF” structure that the server can use to send data back to the client as a part of the connection request process connect and WSAConnect: The “connect” function is defined as

int connect(SOCEKT s,Const struct sockaddr FAR* name, int namelen);

Here, the parameter “s” is the valid ICP socket on which to establish the connection, “name” is the socket address “SOCKADDR_IN” for TCP that describes the server to connect to, and “namelen” is the length of the “name” variable. The Winsock 2 defines “WSASocket” as

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

Here, the first three parameters are exactly the same as the “connect” API function. The next two, “lpCallerData” and “lpCalleeData” are string buffers used to send and receive data at the time of the connection request. The “lpCallerData” parameter is a pointer to a buffer that holds data the client sends to ther server with the connection request. The “lpCalleeData” parameter points to a buffer that will be filled with any data sent back from the server at the time of connection setup. And finally, the last two parameters are also same as that of “connect” function. Data Transmission: For sending data on a connected socket, there are two API functions: “send” and “WSASend”. Similarly, for receiving data on a connected socket: “recv” and “WSARecv” are usedsend and WSASend: The “send” function is defined as

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

Here, the "SOCKET" parameter is the connected socket to send the data on. The second parameter "buf" is a pointer to the character buffer that contains the data to be sent. The third parameter "len" specifies the number of characters in the buffer to send. Finally, the "flags" parameter can be either "0", "MSG_DONTROUTE", or "MS_OOB". The "MSG_DONTROUTE" flag tells the transport not to route the packets it sends. The "MS_OOB" flag signifies that the data should be sent out of band. On successful return, "send" returns the number of bytes sent; otherwise, if an error occurs, "SOCKET_ERROR" is returned. A common error is "WSAECONNABORTED", which occurs when the virtual circui t terminates because of timeout failure or a protocol error. When this error occurs, the socket should be closed, as it is no longer usable. The error "WSAECONNRESET" occurs when the application on the remote host resets the virtual circuit unexpectedly or when the remote host is rebooted. The next common error is "WSAETIMEDOUT", which occurs when the connection is dropped because of the network failure or the remote connected system going down without notice. The Winsock 2 version of the "send" API function "WSASend" is defined as

int WSASend(SOCKET s, LPWSABUF lpBuffers, DWORD dwBufferCount, LPDWORD lpNumberofBytesSent, DWORD dwFlags, LPWSAOVERLAPPED lpOverlapped, LPWSAOVERLAPPED_COMPLETION_ROUTINE lpCompletionROUTINE);

Here, "lpBuffers" is a pointer to one or more "WSABUF" structures. This can be either a single structure or an array of such structures. The third parameter "dwBufferCount" indicates the number of "WSABUF" structures being passed. The "WSABUF" itself is a character buffer and the length of that buffer. The "lpNumberOfBytesSent" is a pointer to a "DWORD" and the "dwFlags" parameter is similar to that in "send" function. The last two parameters "lpOverlapped" and "lpCompletionROUTINE" are used for overlapped I/O, one of the asynchronous I/O models supported by Winsock. The "WSASend" function sets "lpNUmberOfBytesSent" to the number of bytes written. The function returns 0 on success and "SOCKET_ERROR" on any error. recv and WSARecv: The "recv" function is the most basic way to accept incoming data on a connected socket. This function is defined as

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

Here, "s" is the socket on which data will be received, "buf" is the character buffer that will receive the data, while "len" is either the number of bytes required to receive or the size of the buffer "buf". The "flags" parameter can be one of the following values: 0, "MSG_PEEK" OR "MSG_OOB". The "0" specifies no special actions, "MSG_PEEK" causes the data is available to be copied into the supplied receive buffer and "MSG_OOB" is same as that in "send" function. There are some considerations when using "recv" on a datagram based socket. In case the data pending is larger than the supplied buffer, the buffer is filled with as much data as it will contain. In this case, the "recv" call generates the error "WSAEMSGSIZE". It should be noted that the message-size error occurs with message-oriented protocols. Stream protocols buffer incoming data and will return as much data as the application requests, even if the amount of pending data is greater. Thus for streaming protocols, the "WSAEMSGSIZE" error will not be encountered. The "WSARecv" function strengthens "recv" by adding some new capabilities such as overlapped I/O and partial datagram notifications. The "WSARecv" is defined as

int WSARecv(SOCKET s, LPWSABUF lpBuffers, DWORD dwBufferCount, LPDWORD lpNumberOfBytesRecvd, LPDWORD lpFlags, LPWSAOVERLAPPED lpOverlapped. LPWSAOVERLAPPED_COMPLETION_ROUTINE lpCompletionROUTINE);

The "lpNumberOfBytesRecvd" parameter points to the number of bytes received by this call if the receive operation completes immediately. The "lpFlags" can be one of the following values: "MSG_PEEK", "MSG_OOB" or "MSG_PARTIAL". The "MSG_PARTIAL" flag has several different meanings depending on where it is used or encountered. For message-oriented protocols, this flag is set upon return from "WSARecv". In this case, subsequent "WSARecv" calls set this flag until the entire message is returned i.e. when the "MSG_PARTIAL" flag is cleared. The "MSG_PARTIAL" flag is used only with message-oriented protocols.

Breaking the Connection:

Once the socket connection is completed, it is required to close the connection and release any resources associated with that socket handle. It is one with the "closesocket" call. But "shutdown" function should be called before the "closesocket" function.

shutdown:

In order to ensure that all data an application sends is received by the peer, a well-written application is used, which notifies the receiver that no more data is to be sent. Likewise, the peer should do the same. This is known as a graceful close is performed by the "shutdown" function, defined as

int shutdown(SOCKET s, int how);

The "how" parameter can be one of the following: "SD_RECEIVE", "SD_SEND" or "SD_BOTH". For

"SD_RECEIVE", subsequent calls to any receive function on the socket are disallowed. For TCP sockets, if data is queued for receive or if data subsequently arrives, the connection is reset. But for UDP sockets, incoming data is still accepted and queued. For "SD_SEND", subsequent calls to any send function are disallowed. For TCP sockets, this causes a "FIN" packet to be generated after all data is sent and acknowledged by the receiver. Finally, "SD_BOTH" specifies to disable both send and receive.

closesocket:

The "closesocket" function closes a socket and is defined as

int closesocket(SOCKET s);

Here, calling "closesocket" releases the socket descriptor and any further calls using the socket fail with "WSAENOTSTOCK". If there are no other references to this socket, all resources associated with the descriptor are released, including any queued data.

Receiver:

For a process to receive data on a connectionless socket, first create the socket with either "socket" or "WSASocket". Next bind the socket to the interface on which the data is to be received. This is done with the "bind" function. The difference between connectionless sockets is that it is not necessary to call "listen" or "accept". Instead, simply wait to receive the incoming data. Because there is no connection, the receiving socket can receive datagram originating from any machine on the network. The "recvfrom" function is defined as

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

Here, most of the parameters are almost same as that in "recv" function. The "from" parameter is a "SOCKADDR" structure for the given protocol of the listening socket, with "fromlen" pointing to the size of the address structure. When the API call returns with data, the "SOCKADDR" structure is filled with the address of the workstation that sent the data. The Winsock 2 version of the "recvfrom" function is "WSARecvFrom", which is defined as

```
int WSARecvFrom(
  SOCKET s,
  LPWSABUF lpBuffers,
  DWORD dwBufferCount,
  LPDWORD lpNumberOfBytesRecvd,
  LPDWORD lpFlags,
  struct sockaddr FAR * lpFrom,
  LPINT lpFromlen,
  LPWSAOVERLAPPED lpOverlapped,
  LPWSAOVERLAPPED
_COMPLETION_ROUTINE lpCompletionROUTINE
);
```

The difference is the use of "WSABUF" structures for receiving the data. It is possible to supply one or more "WSABUF" buffers to "WSARecvFrom" with "dwBufferCount". The total number of bytes read is returned in "lpNumberOfBytesRecvd". When "WSARecvFrom" is called, the "lpFlags" parameter can be either "MSG_OOB", or "MSG_PEEK", or "MSG_PARTIAL". While calling the function, if "MSG_PARTIAL" is specified, the provider knows to return data even if only a partial message was received. Upon return, the flag "MSG_PARTIAL" is set only if a partial message was received. Upon return, "WSARecvFrom" will set the "lpFrom" parameter (a pointer to a "SOCKADDR" structure) to the address of the sending machine. Again, "lpFromLen" points to the size of the "SOCKADDR" structure, as well as to a "DWORD". The last two parameters are "lpOverlapped" and "lpCompletionROUTINE", which are used for overlapped I/O

Another method of receiving and sending data on a connectionless socket i s to establish a connection. Once a connectionless socket is created, "connect" or "WSAConnect" can be called with the "SOCKADDR" parameter set to the address of the remote machine. The socket address passed into a connect function is associate with the socket so that "recv" and "WSARecv" can be used instead of "recvfrom" or "WSARecvFrom" as the sender is known Sender: For sending data on a connectionless socket, there are two options: the first is simple; create a socket and call either "sendto" or "WSASendTo". The "sendto" function is defined as

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

Here, the parameters are the same as "recvfrom" except that "buf" is the buffer of data to s end and "len" indicates how many bytes to send. The "to" parameter is a pointer to a "SOCKADDR" structure with the destination address of the workstation to receive the data. The second one, the Winsock 2 function "WSASendTo" is defined as

```
int WSASendTo(
  SOCKET s,
  LPWSABUF lpBuffers,
  DWORD dwBufferCount,
```

LPDWORD lpNumberOfBytesSent,
DWORD dwFlags,
Const struct sockaddr FAR * lpTo,
int iToLen,
LPWSAOVERLAPPED lpOverlapped,

LPWSAOVERLAPPED_COMPLETION_ROUTINE lpCompletionROUTINE
);

Here, before returning, "WSASendTo" sets the "lpNumberOfBytesSent" to the number of bytes actually sent to the receiver. The "lpTo" is a "SOCKADDR" structure for the given protocol, with the recipient's address. The "iToLen" parameter is the length of the "SOCKADDR" structure. A connectionless socket can be connected to an end-point address and data can be sent with "send" and "WSASend". Once it is initiated, it is not possible to go back to "sendto" or WSASendTo" with an address other than the address passed to one of the connect functions.

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.

Windows Socket Library Overview

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 descriptor 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 descriptor.
- getsockname() Retrieve the current name for the specified socket
- getsockopt() Retrieve options associated with the specified socket descriptor.
- 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 descriptors.
- 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 descriptor.
- shutdown() Shut down part of a full-duplex connection.
- socket() Create an endpoint for communication and return a socket descriptor.
  - * The routine can block if acting on a blocking socket.

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.
- gethostname() Retrieve the name of the local host.
- gethostbyname()* Retrieve the name(s) and address corresponding to a host name.
- 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.

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.

- Asynchronous select() Mechanism
- Asynchronous Support Routines
- Hooking Blocking Methods
- Error Handling
- Accessing a Windows Sockets DLL from an Intermediate DLL
- Internal Use of Messages by Windows Sockets Implementations
- Private API Interfaces
- WSAAsyncGetHostByAddr() A set of functions which provide asynchronous
- WSAAsyncGetHostByName() versions of the standard Berkeley
- WSAAsyncGetProtoByName() ***getXbyY()*** functions. For example, the
- WSAAsyncGetProtoByNumber()
  WSAAsyncGetHostByName() function provides an asynchronous message based
- WSAAsyncGetServByName() implementation of the standard Berkeley
- WSAAsyncGetServByPort() ***gethostbyname()*** function.
- 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