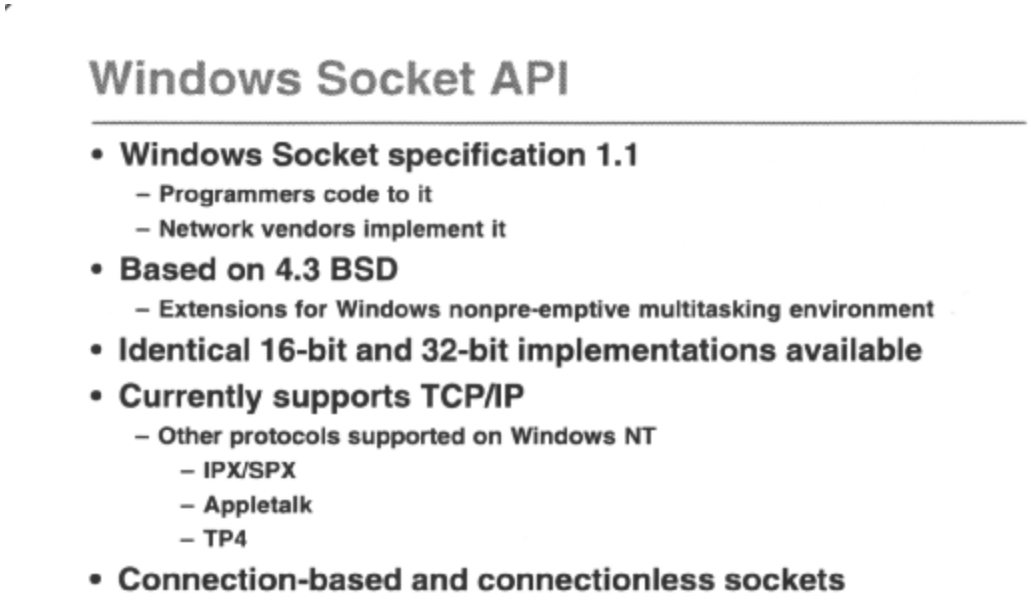


The *Windows Sockets* **specification** defines a network-programming interface for the Microsoft Windows product family, which is based on the Berkeley Sockets programming model (the de facto standard for TCP/IP networking) and is consistent with release 4.3 of the Berkeley Software Distribution (4.3BSD). The API also contains a set of Windows-specific extensions designed to cater for the message-driven, nonpreemptive multitasking nature of 16-bit Windows 3.1.

The specification provides a single API, which abstracts the networking software below and to which developers can program. Windows sockets are implemented as a DLL provided by the vendor of a given network protocol and must conform, at an API level, to the Windows Sockets specification.

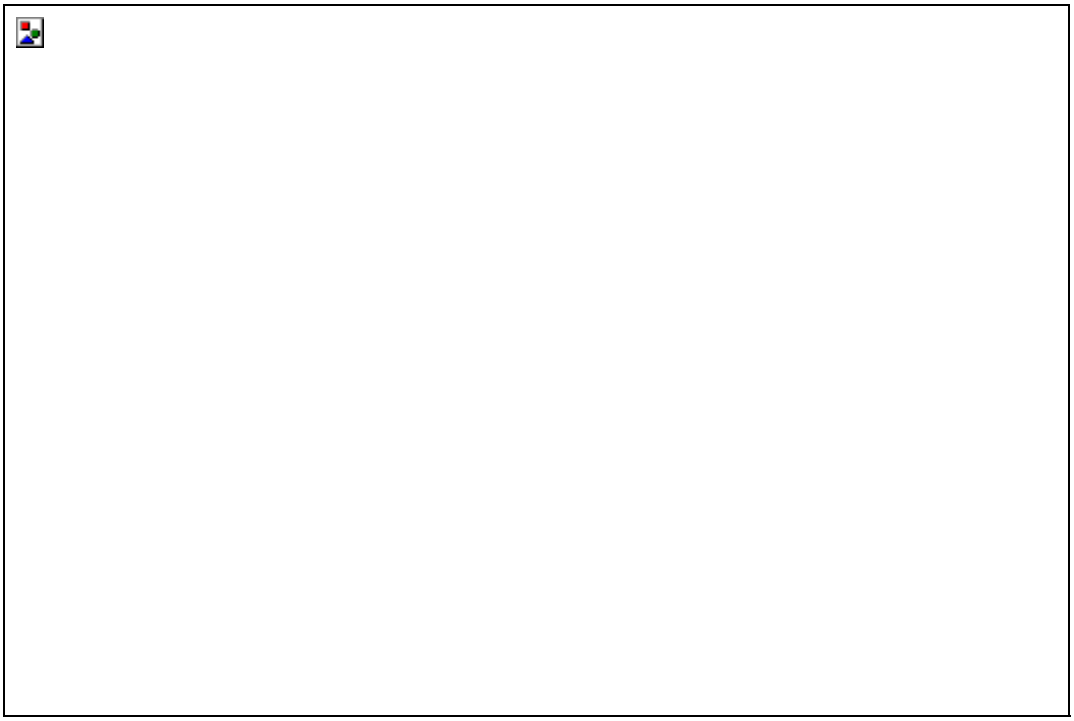
In the context of a particular version of Microsoft Windows, the API defines a *binary interface*; code written to the API can work with any conformant protocol implementation from any network software vendor.

The current version of the specification (1.1) defines and documents the use of the API with the TCP/IP protocol stack, and supports both stream **(TCP)** and datagram **(UDP)** sockets.

Although the initial focus of Windows Sockets was Windows 3.1, Windows NT and Windows 95 have full socket support. ***Winsock.dll*** is provided for 16-bit applications and *wsock32.dll* is the 32-bit implementation. The 32-bit socket API is, of course, widened to 32-bit, but is otherwise identical.

More significantly, the extensions added to Windows Sockets over and above the Berkeley Sockets programming model, to cope with the nonpreemptive multitasking of Windows 3.1 under blocking situations, are not strictly necessary with the pre-emptive multitasking of Windows NT or Windows *95.* These extensions will not be covered in this chapter.

Windows NT supports sockets over protocols other than TCP/IP. These include IPXISPX, OSI TP4, DecNET, AppleTalk and NetBIOS. These each have their own header file, e.g. WSIPX.H, WSHISOTP.H and WSNETBS.H



TCP/IP provides services for both connection-oriented and connectionless protocols. *Transmission Control Protocol* (TCP) is a reliable connection-oriented protocol that delivers large amounts of data in sequence. *User Datagram Protocol* (UDP) is an unreliable connectionless protocol which delivers smaller amounts of data.

A *socket* has a type that is used to describe how data is transferred across a link through the socket. A socket is normally only connected to another socket of the same type, although there is nothing that prevents communication between sockets of different types, should the underlying communication protocols support this.

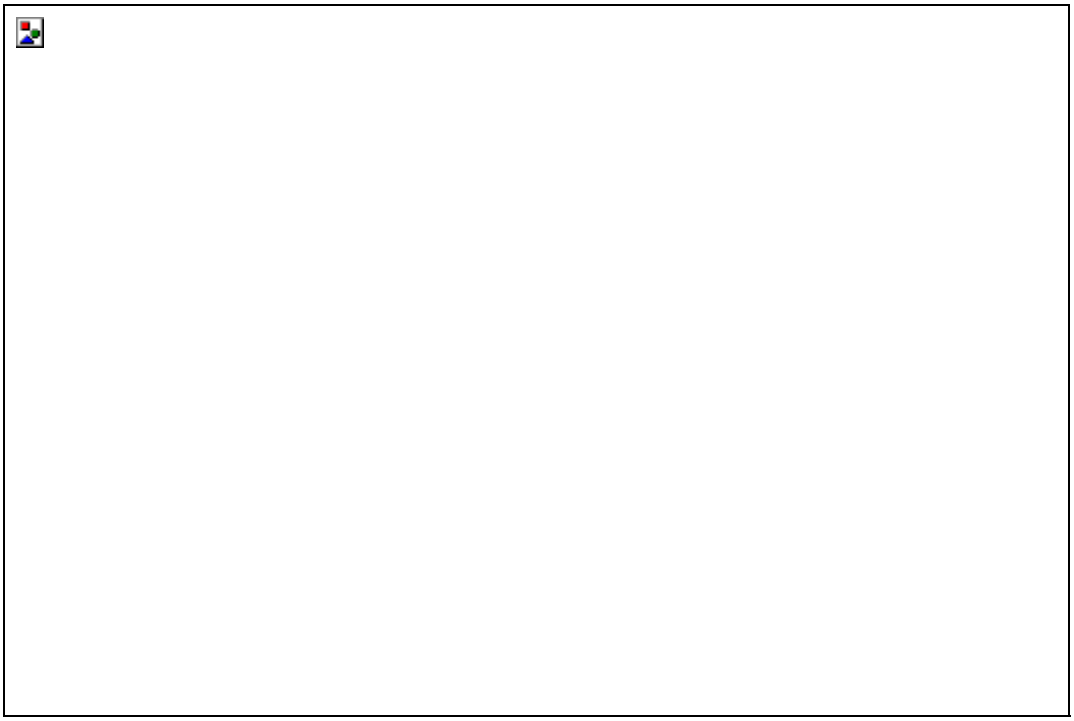
Only two of the 'standard' Unix socket types have been defined for the Internet domain on Windows

SOCK_DGRAM *'Datagram'* sockets. Data is sent in packets, and there is no guarantee of delivery. Packets may be lost or duplicated, or arrive out of order. 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. SOCK_DGRAM provides communications between processes using UDP.

SOCK_STREAM *'Stream'* sockets. These implement a *'virtual circuit'*, full-duplex connection where data is transferred in sequence, non-duplicated and reliably and without record boundaries. Stream sockets require a connection to be established before data may be transferred. If the connection is broken, the communicating processes will be informed. SOCK_STREAM provides communications between processes using TCP.

By analogy, a datagram is like a letter in the mail system; it is connectionless. You post a bunch of letters to a specific address, and cannot be sure whether they will turn up at the other end, or in what order. Record boundaries are maintained; you don't get half a letter arriving.

The stream socket establishes a connection right at the outset, over which data can be passed, reliably and in sequence, in both directions until one party breaks the connection. This is like making a telephone call.



As mentioned before, *sockets* are designed around the client/server model of *IPC*. Communication in this way is asymmetric, in other words each end of the link must know its role. A server application normally listens at a well-known name or endpoint address, for service requests, remaining dormant until a connection is requested by a client's connection to the server's endpoint address. At such a time, the server process 'wakes up' and services the client, performing whatever appropriate actions the client requests of it. The following stages showing a conversation between a client and a server process are shown in the diagram.

First, the server must create a socket and **bind** it to a local name. In the Internet domain, a name is a combination of a unique network address and a host-relative port number. Name and endpoint address are terms used interchangeably. The socket is created using the `socket()` system call, at which time the socket type and communications domain are specified. A socket is referenced using a descriptor, much as an open file is. `bind()` will associate the socket with the appropriate socket name.

The next stage is to have the socket *listen* for connections from other processes. This is done with the `listen()` system call. Here we can also specify the size of a queue of waiting connection requests. Typically, a server application normally listens on a socket bound to a well-known name. In the Internet domain this means listening on a well-known port, over all installed network interfaces.

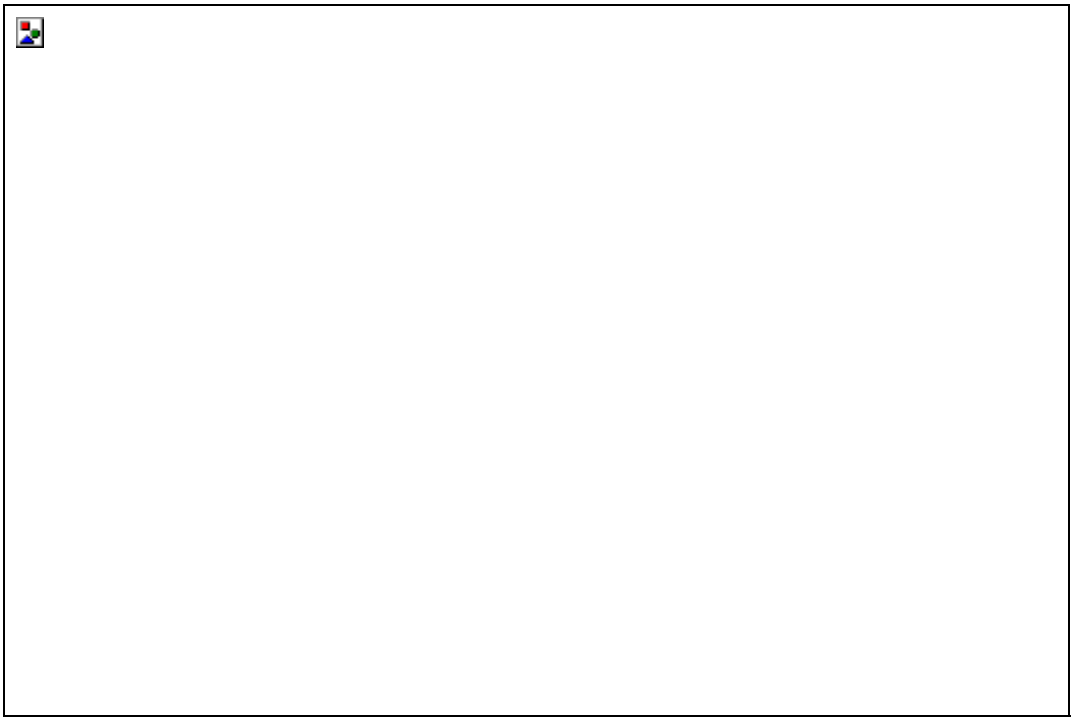
To make its end of a **connection,** the server calls the `accept()` system call. If there are no connection requests to be serviced, this call will normally cause the process to block, waiting for a request. If there are requests (i.e. on the queue that was specified with `listen()`), then a connection is made.

`accept()` returns a new socket descriptor with the same characteristics as the listening socket, which is then used to transfer the data using `recv()` and `send()` . This is done so that, if required, the server may execute a subprocess or thread to manage the data transfer, while it continues to listen for new requests from other clients.

The client process also creates a socket using `socket()` . This socket is to be used to connect to and communicate with the server.

To request a connection with the server process, the client process calls `connect()`, and specifies the name of the socket to which the connection is required. A client normally initiates a connection between a specifically named server socket and a local socket with any appropriate name. In the Internet domain, this means a client initiates a connection between a specific server IP address/port pair, using any appropriate local interface and port. There is no need for the client to `bind()` the socket to a local name, as a sensible bind can be done by `connect()`. In the Internet domain, using a local interface that is appropriate to the server interface, and any unused local port does this. If `connect()` returns successfully, then a rendezvous has been made with the server's `accept()` call, and the connection is established. Data may be transferred using `recv()` and `send()`.

To terminate a connection, either side calls `closesocket()` on the socket, or `shutdown()`. The details of how the connections are closed are dependent on the underlying protocols, but with stream sockets all outstanding data transfers should be completed before the socket is removed.



While *connection-based services* are the norm, some services are based on the use of datagram sockets. *Datagram* sockets lead to a slightly different pattern of interaction between the client and server processes.

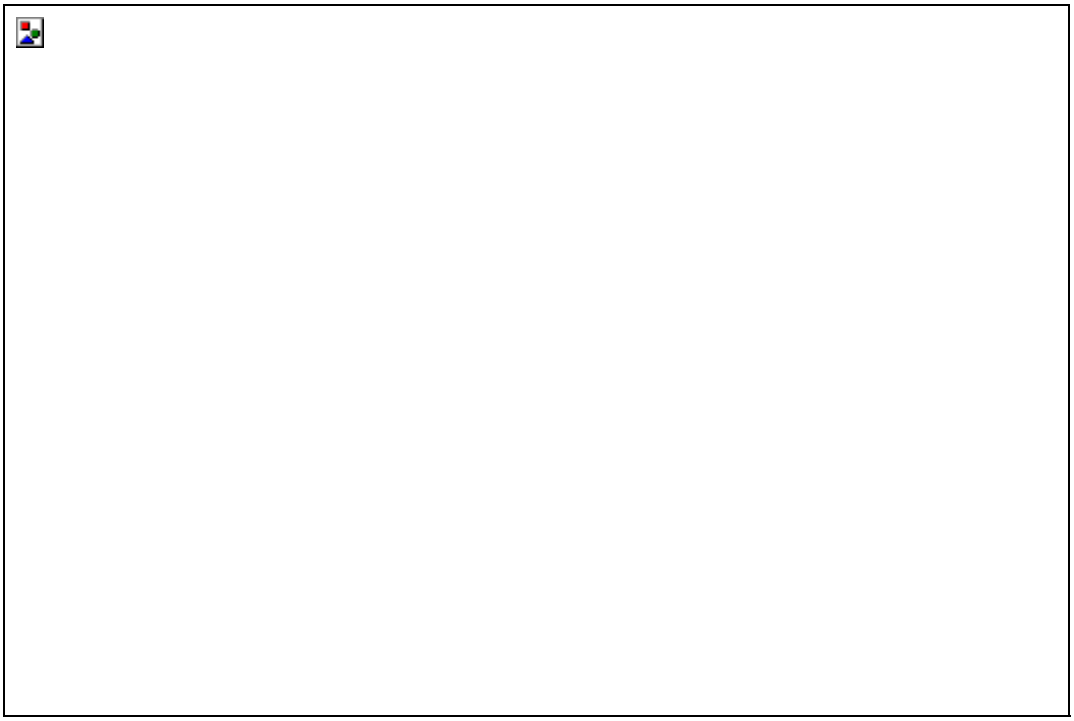
The server must still create a socket and bind it to a local name, as described for stream sockets. However, no firm connection is made in this case; the server waits for a request by calling the `recvfrom()` system call.

This normally blocks the process until a request comes in from a client. The request will contain the socket name of the client, so that after the request has been processed, a reply can be sent to the client using the `sendto()` call, specifying this name.

The client process creates a socket through which to communicate with the server. It must also create a socket for itself and bind a local name to that socket. Again, this bind can be done sensibly by the system, as described earlier. The client socket must be bound so that the server process has a name to which it can send the reply.

The request is made of the server using the `sendto()` call, and the reply received using `recvfrom()`.

Since there is no connection between the processes, there is no need to shut down' the link.



Generally, the Windows-specific extensions to the Berkeley sockets paradigm are to do with the peculiarities of the Windows 3.1 programming model and its nonpreemptive multitasking strategy. For the most part, these can be ignored when using Win32, especially if the process using sockets is a non-windowed application. All such extended functions have an WSA prefix, e.g. `WSAAsyncSelect()`.

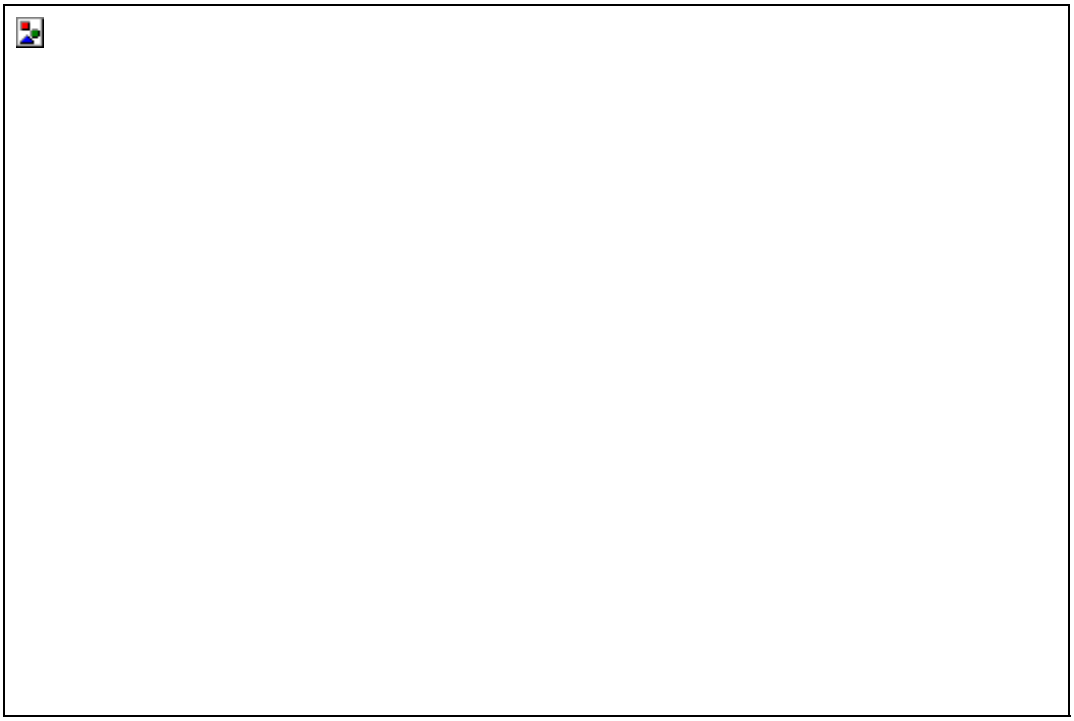
However, there are two WSA functions that you cannot ignore. These are unavoidable because of the 16-bit Windows DLL model.

`WSAStartup()`, called once at application startup, registers your application with the sockets **DLL** to initialize any structures etc that may be needed and increases a reference count. It also provides version negotiation; potentially there could be many Windows Sockets implementations out there supporting different versions of the specification. Even a single implementation could support multiple versions of the specification. The application and the DLL must use the same version. Version control is always a problem in the 'late binding' architecture of DLLs. Other implementation-specific details like vendor details, maximum datagram size supported and maximum number of sockets that can be opened, are also returned.

`WSACleanup()` called once at application cleanup before exit, deregisters your application from the sockets DLL, allows it to tidy up any garbage and decrements the reference count.

The following code gives a guide to using these functions:

```
void main()
{
       if ( WSAStartup ( MAKEWORD (1,0), &wsaData ) !=     0 )
              DisplayLastError ( 'Socket start-up failed");
       else
       {
              /*Confirm that the Windows Socket DLL supports 1.0. Note that if the DLL
              supports versions greater than 1.0 in addition to 1.0, it will still return 1.0
              in aversion since that is the version we requested.*/

       if (LOBYTE(wsaData.wVersion) != 1 || HIBYTE(wsaData.wVersion) != 0 )
              /* Couldn't find a useable winsock.dll - it's a socket library Jim, hot not as we
              know it */
              printf( "Couldn't agree on socket lib version\n");
       else {
              /* it's my kinda socket library "/
              ……
       }
       if ( WSACleanup() = SOCKET_ ERROR )
              DisplayLastError ( "Socket clean-up failed");
       }
}
```



The `socket()` system call allocates a socket descriptor of the specified address family, `data` type and protocol. The call simply creates the endpoint. It is not at this stage bound to any name in the domain name space. or connectcd to any other socket.

If no error occurs, socket-) returns the socket descriptor, otherwise it returns INVALID_SOCKET, with `WSAGetLastError ()` indicating the precise error code.

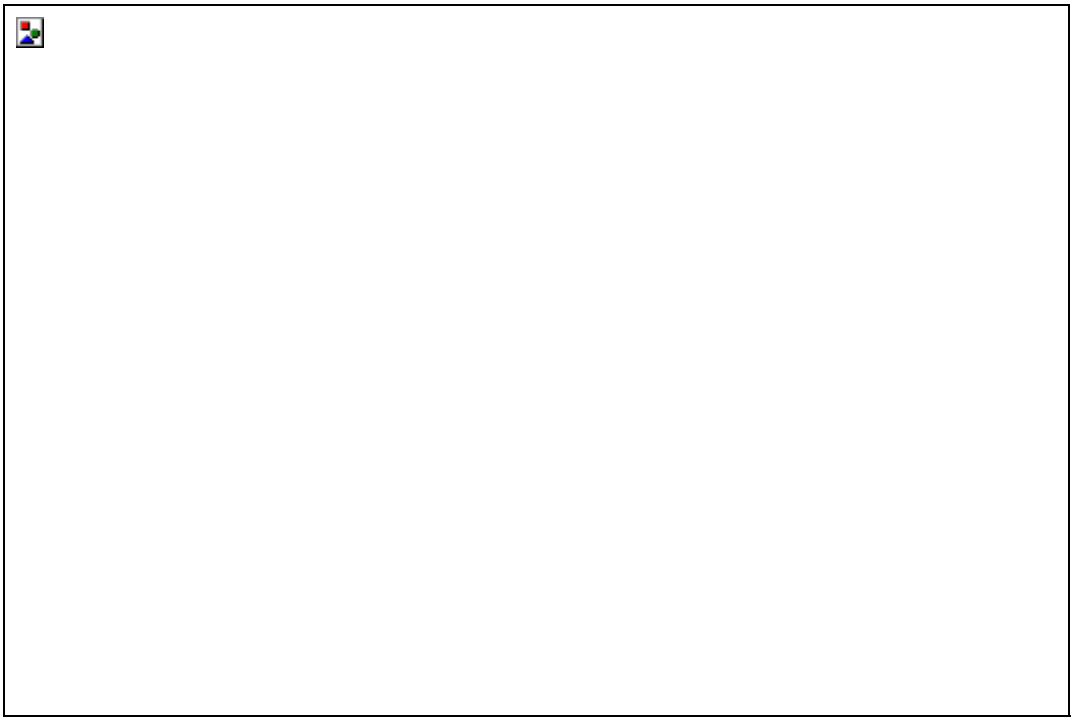
It is necessary to specify the type of the socket and the domain, or address family, in which the socket is to operate. Both these can be specified as constants deflncd in the include file *winsock.h.* The only domain that can be specified is AF_INET for the Internet domain. The only valid values for the socket type within the Internet domain axr SOCK_STREAM or SOCK_DGRAM as described earlier.

There will be a single protocol used when a particular type of socket is specified for a particular domain (eg. TCP used for a stream socket in the Internet domain), so in most cases, the protocol parameter may safely be 0.

It should be noted that on Windows NT the returned socket descriptor is a standard file handle and can be used in calls to `ReadFile()` and `WriteFile()`, However by default socket handles are asynchronous so an OVERLAPPED structure must be initialised and passed to any call to `ReadFile()` or `WriteFile()`. Socket handles can be synchronous; use `setsockopt()` to change behaviour of select and accept on a per-thread basis;

```
int optionValue = SO_ SYNCHRONOUS_NONALERT;

setsockopt ( INVALID_ SOCKET, SOL_SOCKET SO_OPENTYPE, (char *)
&optionValue, sizeof(optionValue) ) ;
```

this will make any socket descriptors returned by socket ( ) or accept ( ) non-overlapped.



When a socket is created with `socket()` ,it exists for an address family and is referenced by a socket descriptor, but it has no local 'name' or 'endpoint address' assigned.

`bind()` is used to associate a name with a socket. The format of the name will depend on the domain oft he socket, but the routine is specified in a fairly generic way.

The local name to hind a socket to is specified using a generic structure

```
typedef struct sockaddr {
u_short sa_family;      // e.g. AF_INET
char  sa_data[14] ;
  }SOCKADDR,*PSOCKADDR;
```

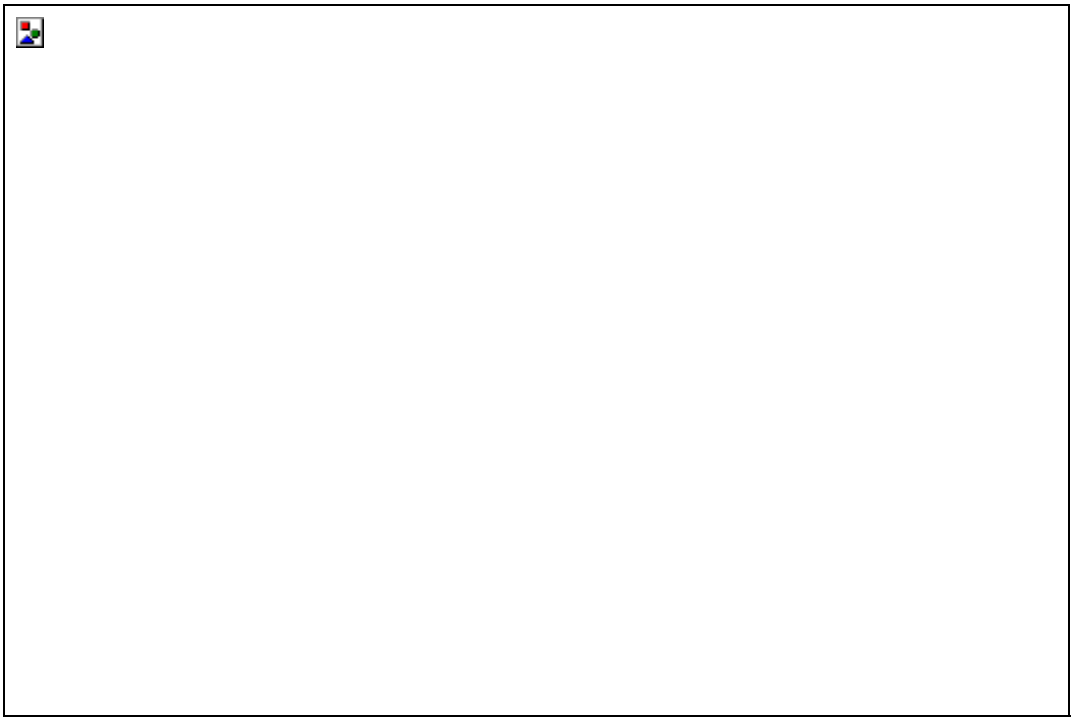
The detail of the contents of this structure is dependent on the address format of the domain, and contains enough information for the underlying network transport to deliver the data to a process. For example, the Internet address family uses the 32-bit Internet address that identities the network interface, plus the 16-bit port number that identifies the process, plus the protocol number (set implicitly to UDP or TCP for datagram sockets and stream sockets, respectively). Sockets in the Internet domain are named with a structure of type SOCKADDR_IN as described in die slide.
The port and IP addresses are always specified in network order.
The `bind()` routine requires the name details to be passed by a pointer to the appropriate structure, plus the actual length in bytes of the structure.

lf no error occurs, `bind()` returns 0, otherwise it returns SOCKET _ERROR with `WSAGetLastError()` indicating the precise error code.

If a client application does not care what IP address is assigned to it. or a server wishes to listen on any available network interface IP address, then an special Internet address if INETADDR_ANY may be specified. This simplifies application programming in the presence of hosts that have several installed network interlaces.

If a client application doesn't care what port it is bound to, a port equal to 0 may be specified. and an appropriate unused local port is chosen. If the port is specified as 0. the system will assign a unique port to the application with a value between (IPPORT _RESERVED)1024 and 5000 Ports less than IPPORT_ RESERVED arereserved for privileged processes. like TELNET.

In these cases, the application may use `getsockname()` after a `bind()` to find the name that has been assigned to it. Of course. on the Internet domain `getsockname()` will not necessarily fill in the Internet address until the socket is connected, since several Intenet addresses may be valid if the client and server hosts contain several installed network interfaces.

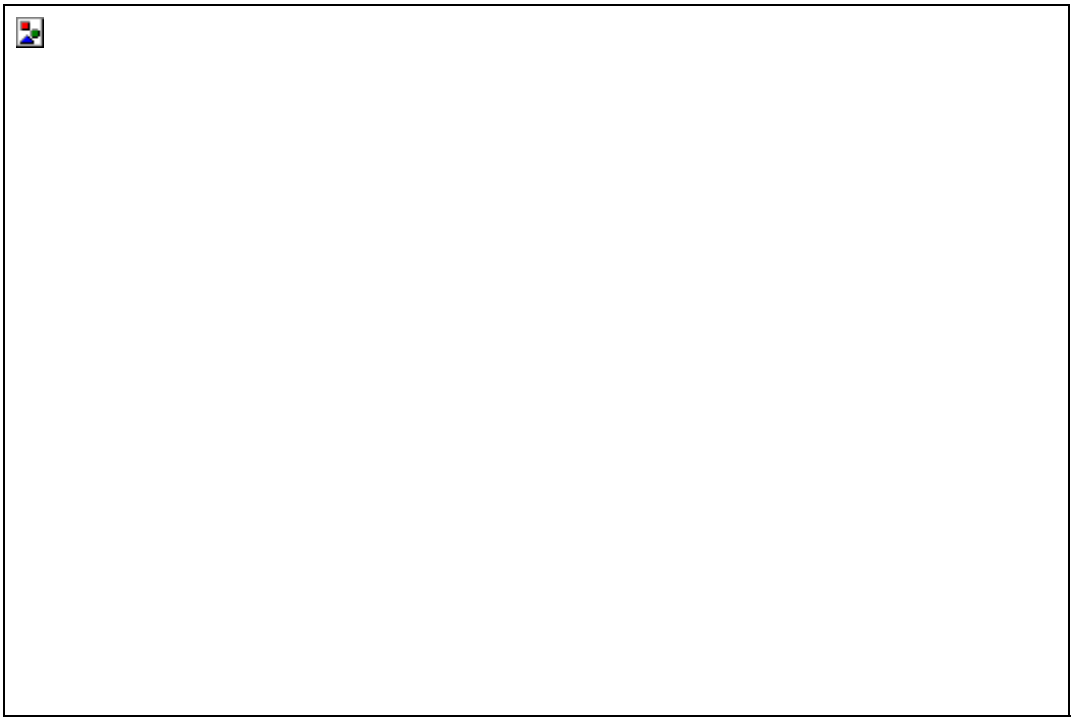


`listen()` is uscd to prepare a connection-oriented socket (of type SOCK_ STREAM) for accepting incoming connection requests. Basically the call sets up a *queue* onto which connection requests are placed. The maximum size of the queue is set here.
Any incoming connection requests that arrive when the queue is full will he refused, with an errorcode WSACONNREFUSED as a result of the `connect()` attempted by the client. Servers that could have more than one connection typically use this function.

If no error occurs, `listen()` returns 0, otherwise it returns SOCKET_ERROR with `WSAGetLastError()` indicating the precise error code.

To extend the analogy with thc phone system for connection-based sockets, this is like a call-waiting system. Callers, up to a limit, will be queued and put on hold awaiting a connection. Callers after this will be refused a connection with the engaged tone.

The queue 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.



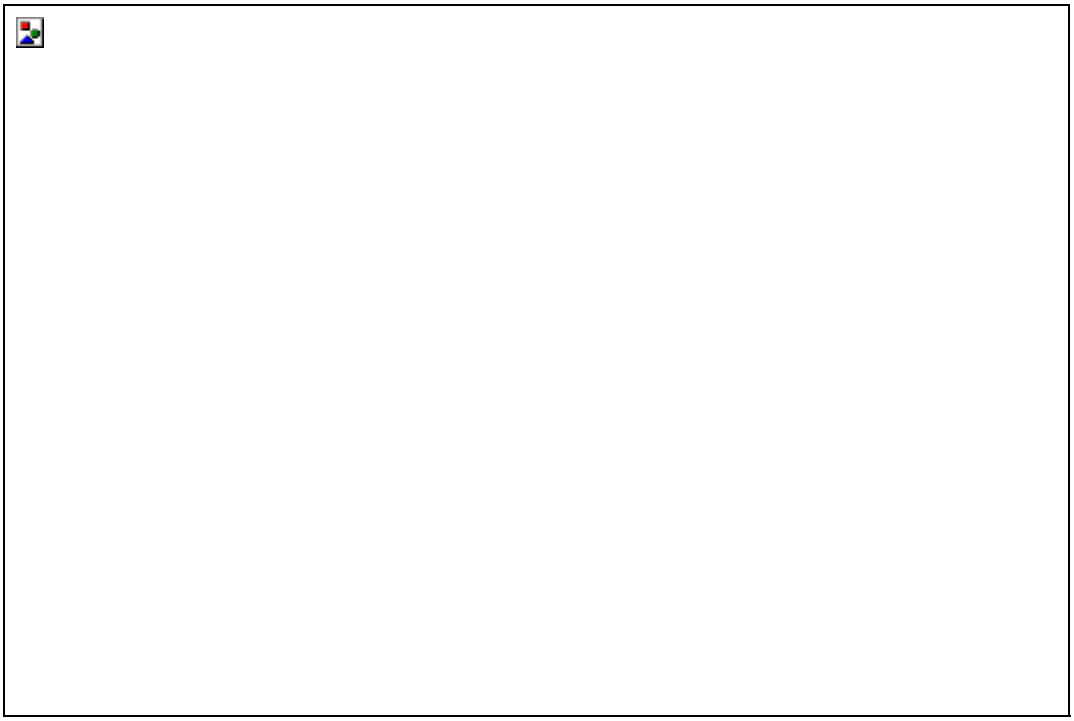
Establishing a connection between a client and a server process is done by a form of rendezvous. Alter the server has set up the characteristics of its socket, it waits for a connection request to come in.
The `accept()` system call achieves this.

`accept()` extracts the first connection on the backlog queue of pending connections on a listening socket. If there are any pending connection requests in the queue then the call returns immediately. If the connection request is accepted, `accept()` returns 0, othenwise it returns SOCKET_ERROR, with `WSAGetLastError()` indicating the precise error code.
If no pending connections requests are present on the queue, and the socket is not marked as nonblocking, `accept()` blocks the calling thread until a connection request is received. If the socket is marked nonblocking, and no pending connection requests are present on thc queue, `accept()` returns SOCKET_ERROR with an error code of WSAWOULDBLOCK. In this case `select()` can be used to wait for the socket to become readable, or periodically check if the socket is readable, to determine a pending connection request. se lect Is covered later.

When a request is received, and the connection is made, `accept()`returns a new socket descriptor that has the same properties as the listening socket, which is intended to be used for the communication between the two processes. Now the `recv()` and `send()` calls can safely be used on the new descriptor. The accepted socket may not be used to accept more connections- The original listening socket remains open and listening and ready to accept more connections.

This implies a certain mode of operation of the server process. listening on one socket for connection requests while at the same time performing the required data exchanges on different sockets. This allows multiple connections to he handled relatively easily by using threads or subproeesses.

When a connection is made, `accept()`optionally fills in the name of the client socket in the structure pointed to by the `client_name` parameter, and its length in the namelen parameter. This is so the server knows which client it is dealing with, e.g. for security reasons. Thc exact format of the `client_name` parameter is determined by the address family in which the communication is occurring These parameters can both be set to NULL if the address of the peer socket is not required.



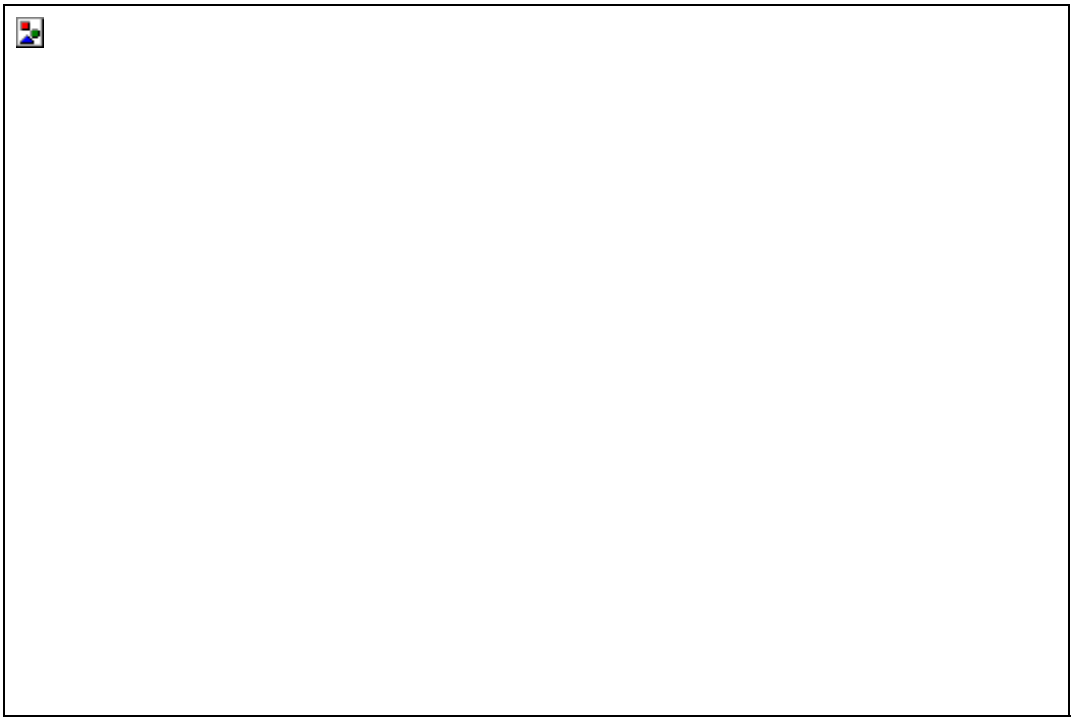
The main use of `connect()` is as the client end of the rendezvous procedure, to make a connection between processes on a connection-orriented socket. The server has made (or will make) a call to `accept()` and when the client calls `connect()` the link is made.

The socket that is to be used from the client must specified, together with the name of the server socket to which a connection is being sought. The socket must be bound to a local name, but if the socket is unbound at this point, unique values are assigned to the local association by the system, and the socket is marked as bound, In the Internet domain, the IP address is chosen from any appropriate network interfaces installed on the system and the port is chosen from any unused, valid port. The client shouldn't care what its local association is, only the name that will connect it to a server socket.

If he socket is not marked as nonblocking. connect () will block the calling thread until the connection is made or the connection is refused (for example the backlog is full, or perhaps some protocol specific timeout occurs). If the connection is made, `connect()` retums 0. otherwise it returns SOCKET_ERROR with WSAGetLastError( ) indicating the precise error code.

If the socket is marked as nonblocking, and the connection request would block, `connect()` will return SOCKET_ERROR with an error code of WSAEWOULDBLOCK. In this case `select()` can be used to wait for the socket to become writeable or periodically check *if* the socket is writeable, to determine the completion of the connection request. `select()` is covered later.

`connect()` can also be used with connectionless protocols, forcing a particular socket to always send its datagrams to a particular named socket.



`send()` and `sendto()` are used to transmit outgoing data through a socket. They are similar to the standard C run-time `write()` call, but more flexible. The flags parameter allows protocol-specific details to be specified. It will normally be 0.

`send()` is normally used to write data on connection-oriented sockets, but may also be used on connectionless sockets where `connect()` has been used to mark the destination thr packets (see earlier).

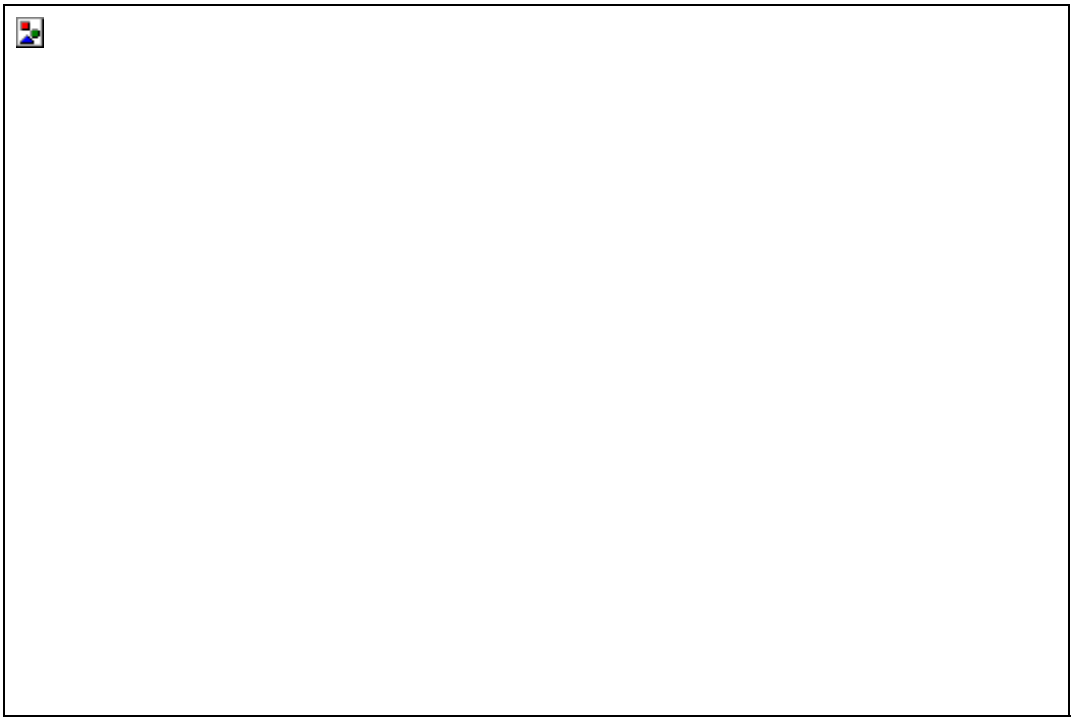
`sendto()`is designed for writing data on connectionless sockets, and the destination name must be included with each packet of data sent in this way. The name of the sending socket will automatically be included in the message when it is sent. If `sendto()` is used on a connection-oriented socket, the `toname` and `namelen` parameters are ignored; in this case the `sendto()` is equivalent to `send()`

To send a broadcast (on a connectionless socket only), the address in the `toname` parameter should be constructed using the special IP address INADOX_BROADCAST, together with the intended port number.

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, an error is returned and no data is transmitted.

Both routines return the number of bytes successfully sent or SOCKET_EPROR, with `WSAGetLastError()` indicating the precise error code. Note that the number of bytes sent may be less than the number of bytes requested to send (see below). Note also that the successful completion of a `send()` or `sendto()` 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()` and `sendto()` will block, unless the socket has been placed in a nonblocking **I/O** mode. On nonblocking SOCK_STREAM sockets, the number of bytes written may be between I 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 remaining data.



`recv()` and `recvfrom()` are used to receive incoming data through a socket. They are similar to the standard C runtime `read()` call, but more flexible. The flags parameter allows protocol-specific details to be handled, and will normally be 0. However, one value of possibie interest will be MSG_PEEK which allows a process to examine data in a socket without actually removing it from the socket

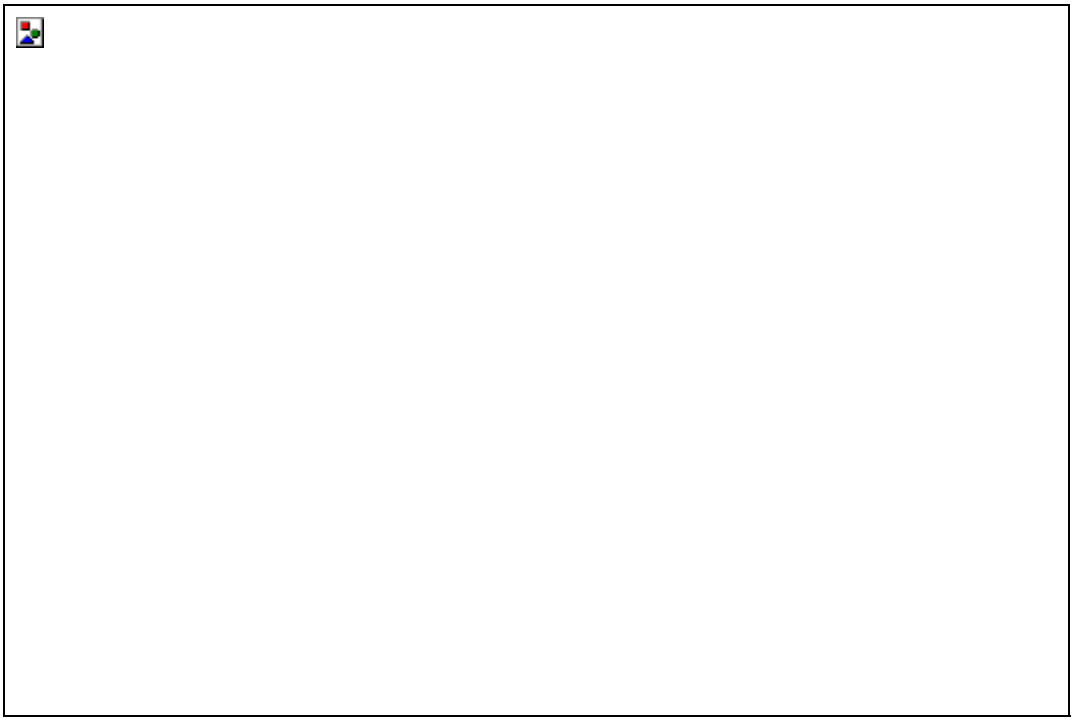
`recv()` is normally used to read data on connection-oriented sockets, but may also be used on conneetionless sockets where connect () has been used to mark the source of the packets (see earlier).

With `recvfrom()` the name of the sender is filled in. If the calling process is not interested in the name of the sender, the `fromname` and `namelen` parameters may be set to NULL, and the address will not be filled in.

Both routines return the number of bytes read or SOCKET_ERROR. with WSAGetLast Error() indicating the precise error code. A zero return means that the connection has been closed gracefully.

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 excess data is lost, and `recv ()` returns SOCKET_ERROR with the error WSAEMSCSIZE

If no incoming data is available at the socket, the `recv()` call waits for data to arrive, unless the socket is nonblocking. For a nonblocking socket, a value of SOCKET_ERROR is returned with the error code set to WSAEWOULDBLOCK. The `select()`call may be used to determine when it is possible to receive data.



As mentioned previously, the binary information used by TCP/IP that goes to make up a socket name can be abstracted using 'user friendly' textual ***names***. The mapping between the two is normally held in local database files. The socket API defines a number of 'database' routines, to translate between the two forms. These may be implemented in a manner that does not depend on local database files: network interaction may be required as with using DNS. If this is the case, these functions may block.

The functions fall into three categories. The service name resolution functions are:

`getservbyname()` Get service name(s) and port corresponding to a service name.
`getservbyport()` Get service name(s) and port corresponding to a port.

and retum a
```
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
      };
```

The host name resolution functions are:

`gethostbyaddr()` Get name(s) and address corresponding to a network address.
`gethostbyname()` Get name(s) and address corresponding to a host name.

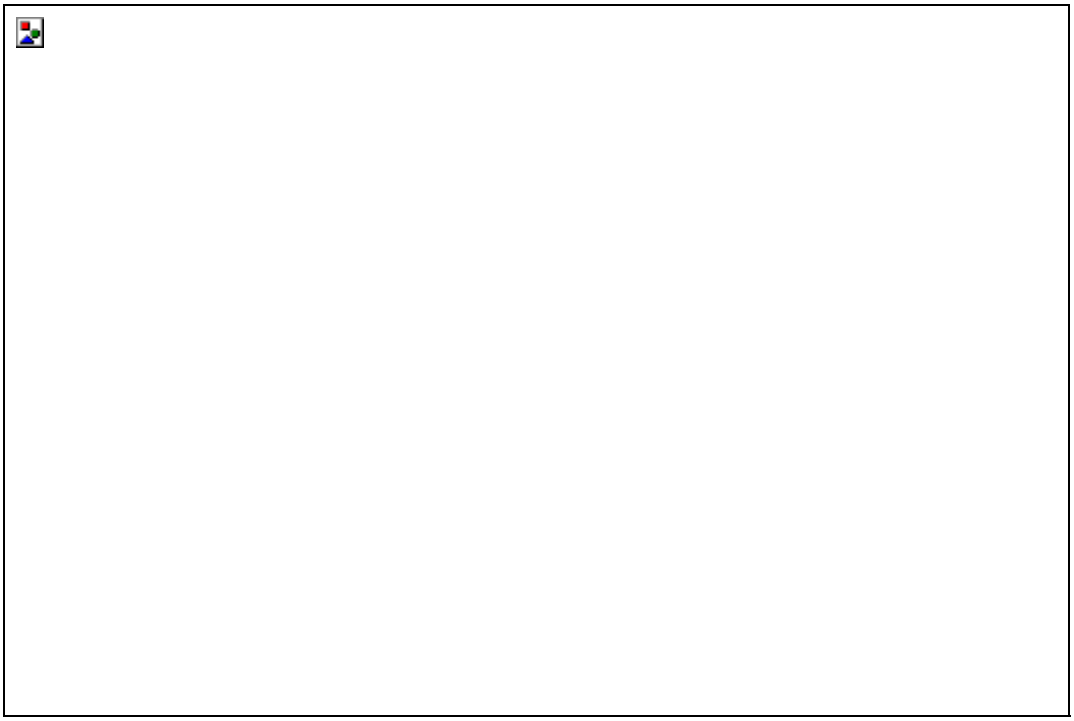
and retum a
```
struct hostent {
      char  FAR * h_name;           // official name of host
      char  FAR * FAR * h_aliases;  // alias list
      short h_addrtype;             // host address type
      short h_length;               // length of address
      char  FAR * FAR * h_addr_list; // list of addresses
      };
```

The protocol name resolution functions are:

`getprotobyname()`	Get protocol name(s) and number corresponding to a protocol name.
`getprotobynumber()`	Get protocol name(s) and number corresponding to a protocol number.

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

These routines return a pointer to a volatile structure that is good only until the next socket API call from that thread. This structure must not be modified in any way, nor should any of its components be freed. Only one copy of this structure is allocated for a thread, so tbe application should copy any information that it needs before issuing any other socket API calls.



In a *distributed environment*, assumptions cannot be made about the type of remote computer with which communication will take place. Different processors have different byte ordering, thus care must he taken to ensure correct orientation.

‘When four parts of a dotted IP address a.b.c.d are specified, each part is interpreted as a byte of data and assigned, from left to fight, 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.

IP addresses or port numbers passed to or from a socket API are in network order (Big-Endian). For instance, the IP address and port fields of a SOCKADER_IN describing an Internet address are in network order.

Conversion routines are supplied:

`ntohl()` / `ntohs()` convert long/short from network order to host order `ntonl() / htons()` convert long/short from host order to network order

Other useful routines are*:*

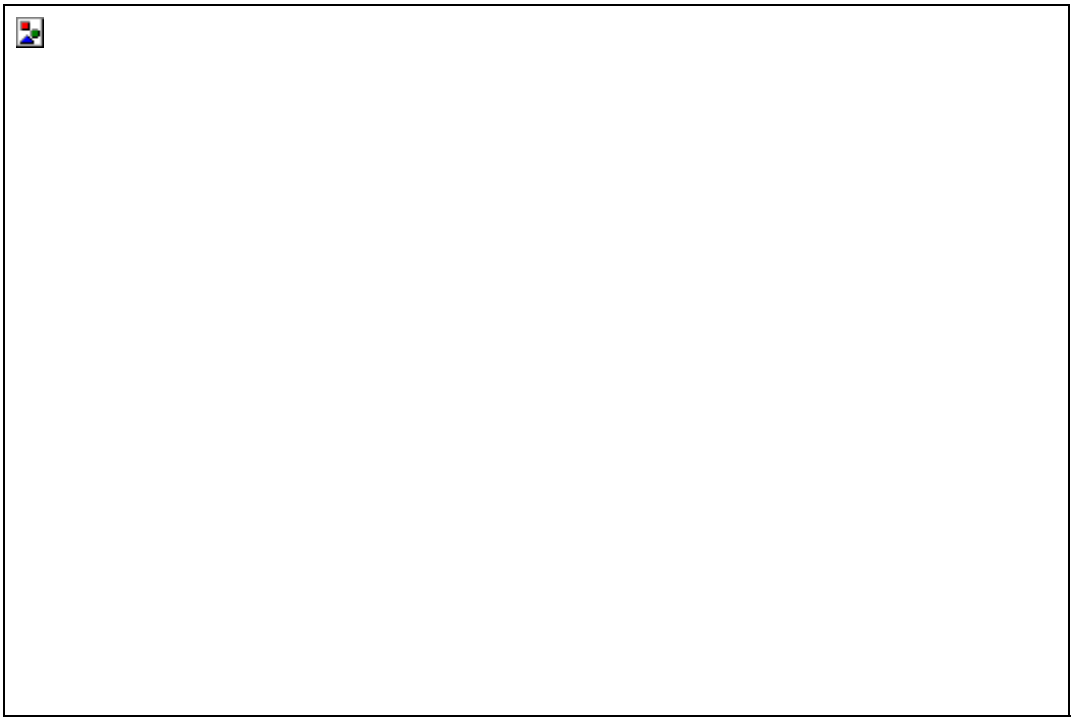
`inet_addr()` convert IP address from dotted-string format to network ordered 32-bit value
`inet_ntoa()` convert IP address from network-ordered 32-bit value to dotted-string format

Contacting a server on a well-known service, such as the FINGER service. using `getservbyname()` will yield a TCP port that is already in network order, so no translation is required before constructing the address.

Contacting a server on a port TO (perhaps entered as an integer by the user) forces the application to convert this from host to network order, using `htons()`, before constructing the address.

Conversely, to display the port number of an address, say returned by `getpeername()`, the port number must be converted from network to host order, using `ntohs()` before it can be displayed.

These standard conversion functions should be used for portable code. Remember that Intel and Internet byte orders are different, but Windows NT, for instance, runs on systems for which the host order is identical to the network byte order.



In line with a similar facility in the Win32 API, the socket API provides a mechanism for reporting errors on a per-thread basis, rather than relying on a global error variable, which doesn't work too well in multi-threaded envfronments

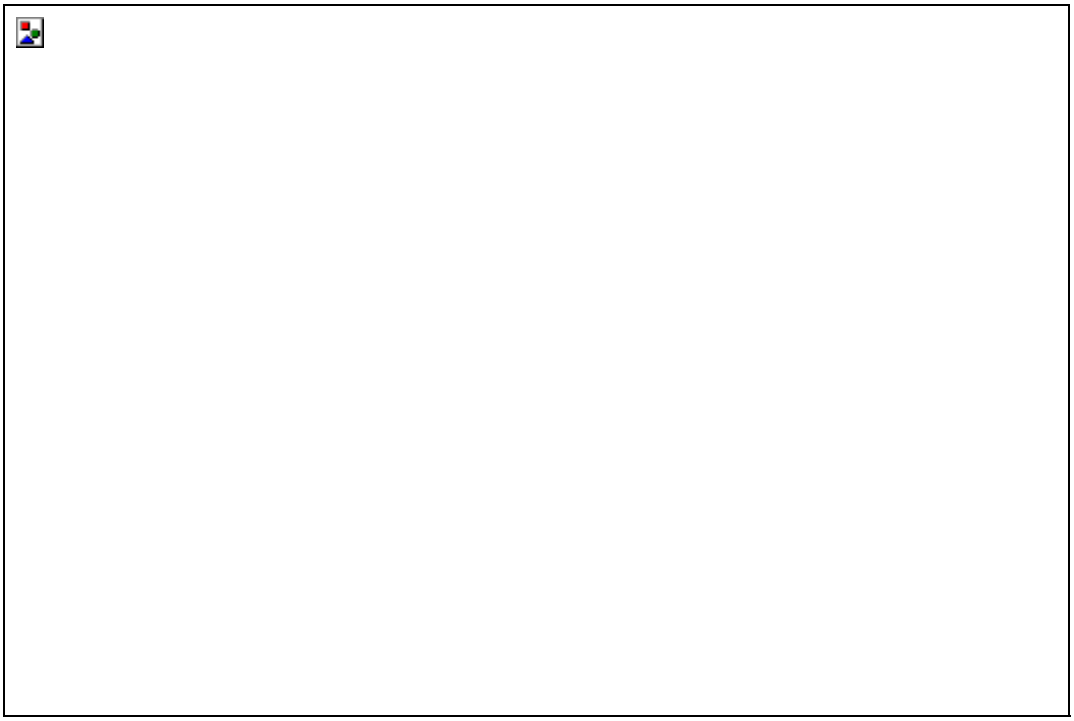
`WSAGetLastError()` returns the last socket API error for the calling thread. Most socket APIs return the fact that the call was successful, or mat there was some kind of error; it is the responsibility of the programmer to call `WSAGetLastError()` to find out precisely what kind of error has occurred so that appropriate action can be taken.

Socket errors can also be set by `WSASetLastError()`, to be picked up by a subsequent `WSAGetLastError()`

In a Win32 environment. `WSAGetLastError ()` will invoke `GetLastError()` , which is used to retrieve the error status for ail Win32 All functions on a per-thread basis. For portability, an application should use `WSAGetLastError ()`.

Here is a function tat will report socket errors:

```
int DisplayLastError( LPSTR lpszText) {

int nError = WSAGetLastError();

printf( "%s because error %d\n", lpszText && lpszText[O] ? lpszText
failed", nError );

return nError;
}
```



By default, socket calls will block, if necessary, until the requested operation is complete or fails.
This is obvious in the case of, say `recv()` where one thread waits on another to send data. It is also well understood that a server `accept()` or a client `connect()` may block before a connection is established.
It is not quite so obvious that the database functions will also block, especially if network interaction is required to resolve names, as is the case if the DNS service is used.

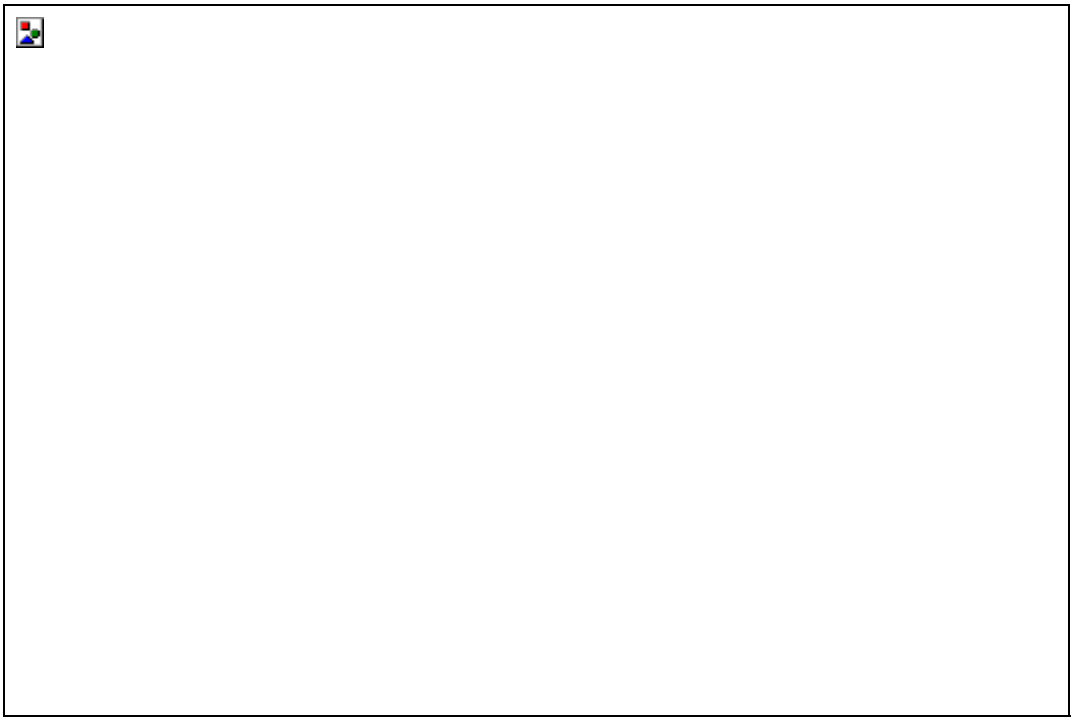
This blocking behaviour is often unacceptable for some applications that could be getting on with other things instead of being suspended for an arbitrarily long period pending a network event. To support such applications, sockets can be set to nonblocking mode. In this mode, any operation on a socket that would cause a call to block for an extended period of time, returns with an error code of WSAWOULDBLOCK. This indicates that the requested operation is pending, and the socket can be inspected at a later date to test whether the operation has completed.

`ioctlsocket()` can be used to set the blocking mode of a socket

Of course, in a pre-emptive multitasking environment it is possible to just start another thread that will call a socket API that is likely to block. However, there are no API extensions that cope with multi-threading, so care must be taken to synchronise access to a socket between threads. Failure to do so may lead to unpredictable results, much as it would with, say file I/O. For example, with two simultaneous' calls to `send()`, there is no guarantee of the order the data will be sent in.

The default blocking behaviour would certainly not be acceptable in a Windows 3.x application, because of the nature of the nonpre-emptive multitasking. There are parts of the Windows Sockets interface that are extensions to BSD sockets designed to cope with this. They extend the APT for use in the message-passing Windows environment. These calls can be implemented in Win32 applications, but are not strictly necessary.

A Windows Sockets implementation on a nonpre-emptive multitasking version of Windows includes a default mechanism for blocking socket functions. The function `WSASetBlockingHook()` gives the application the ability to execute its own 'blocking hook' function at 'blocking' time in place of the default blocking function. This could implement application-specific cooperative multitasking. There is no such default blocking-hook function for a Windows Sockets implementation on a pre-emptive multitasking version of Windows; blocking is performed by the socket functions themselves.

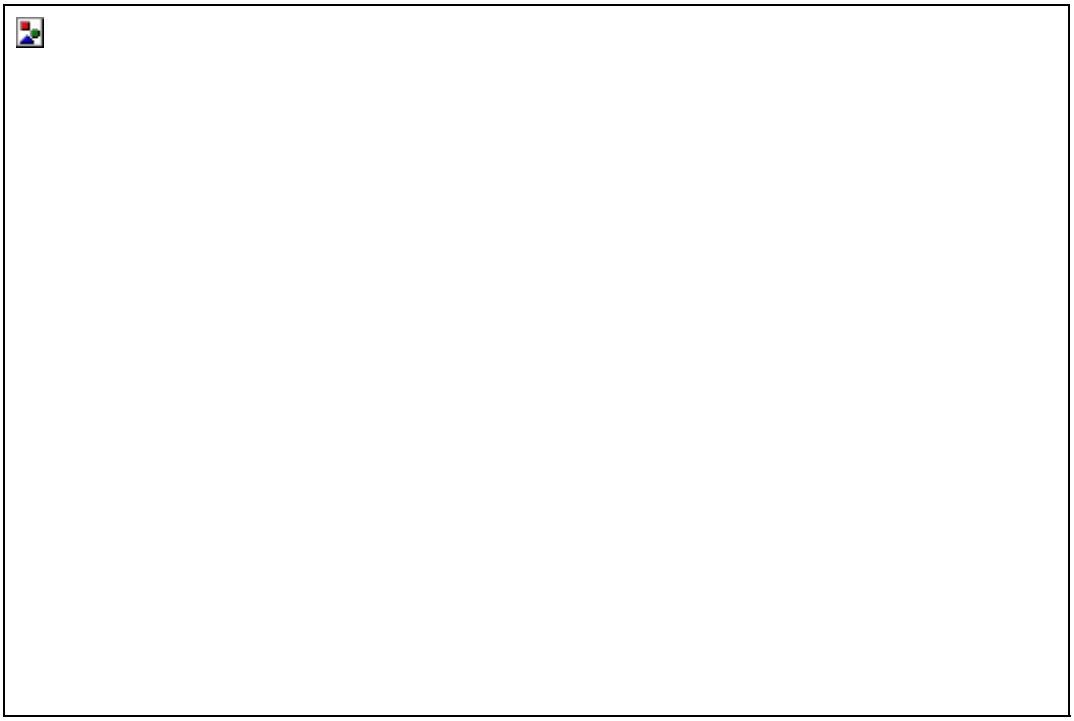


When a socket is set to nonblocking mode, `select()` becomes important. It is possible to wait for a given period of time (perhaps indefinitely) for a socket tat would otherwise be blocked to attain a given state. It is also possible to not wait, but to just check.

An arbitrary set of sockets can be checked for readability, writeability or error condition. If the socket is currently listening, it will be marked as readable if an incoming connection request has been received, so that an `accept()`is guaranteed to complete without blocking. On sockets that have called `accept()`, readability means that queued data is available for reading or the socket connection was closed. If a socket is connecting, writeability means tat the connection establishment is complete. Otherwise, writeability means that a `send()` or `sendto()` will complete without blocking, but this is liable to change quickly in a multithreaded environment. An error condition could be the breaking of the connection by the peer.

In a nonpre-emptive Windows environment, polling for socket-state change is not considered a stylish solution, so `WSAAsyncSelect()` will notify a specified window procedure tat a particular socket event has occurred by posting a message. For example a SOCKET_MESSAGE with FD_ READ in the HIWORD of the 1Param means that data has arrived on the socket The use of `WSAASyncSelect()` automatically marks the socket as nonblocking.

There are also asynchronous versions of all the database routines, for example:
`WsaAsyncGetHostByName()` instead of `gethostbyname()`, will post a message to a specified window procedure, indicating that the hostname has been resolved.

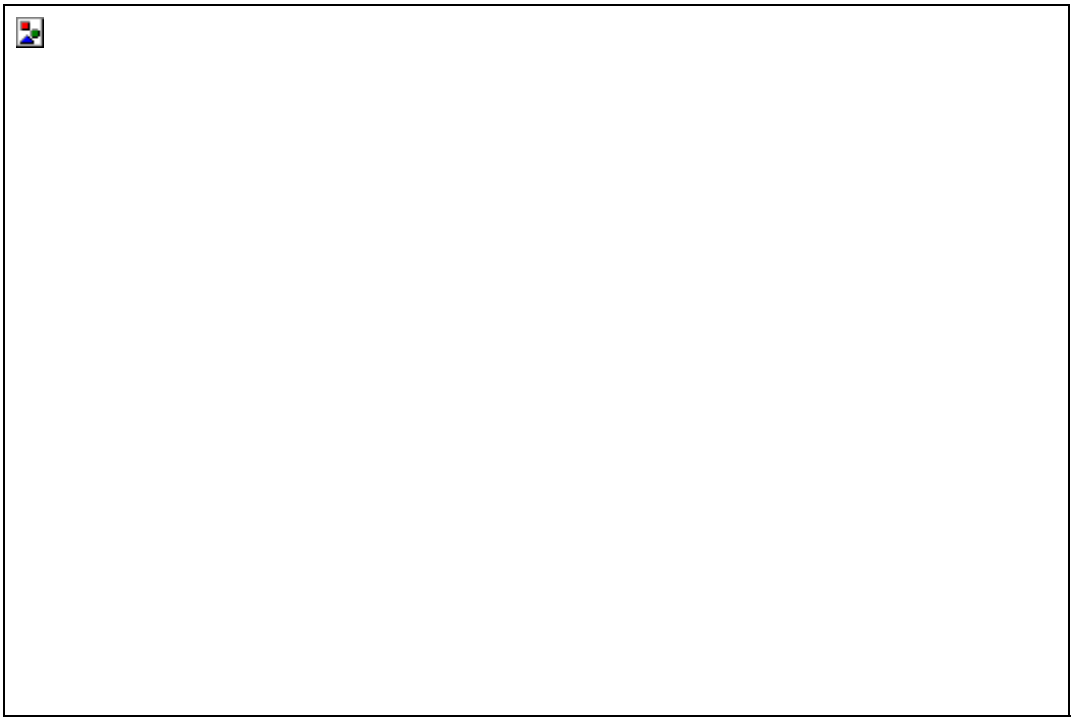


This example shows the basic structure of a server using *connection-oriented sockets*. The socket is created, and bound to an address. The queue is established with `listen()`, and then the server main loop is entered. The server continually waits for a connection request with `accept()`. When a request is received, it calls `_beginthread()` , creating a thread that will handle the request. This allows the server to continually monitor the '*connection socket*' for requests coming in, so that many requests can be handled concurrently. `accept()` returns a new socket descriptor when it completes, and it is this new socket that can be used in the new thread, to transfer the data between the server and client.

Below is the thread function to perform socket I/O, and the function to construct the local-address structure for the server on an Internet domain socket:

```
void ReceiveThread(void *p) {

        SOCKET sock=(SOCKET)p;
        char szBuf [128];

        while (1) {
                int status=recv(sock,szBuf,sizeof(szBuf),0);
                if (status==SOCKET_ERROR) {
                        DisplayLastError ( "Couldn't read IO socket)');
                        break;
                }
                else
                if (status){
                        szBuf[status] =0;
                        printf("%s",szBuf);
                }
                else
                        break;
        }
        closesocket (sock);
}
void vtocalSrvName(PSTR pszServiceName,int nPortID, PSOCKADDR_IN psin ){
        psin->sin_family=AF_INET;
        psin->sinport=htons (nPortID) ; psin->sin_addr.s_addr=INADDR_ANY;
        if (pszServiceName&&pszServiceName[0) ) {
```

```
                PSERVENT pse=getservbyname(pszServiceName, “tcp" );
                if (pse)
                        psin->sin_port=pse->s_port;
        }
}
```

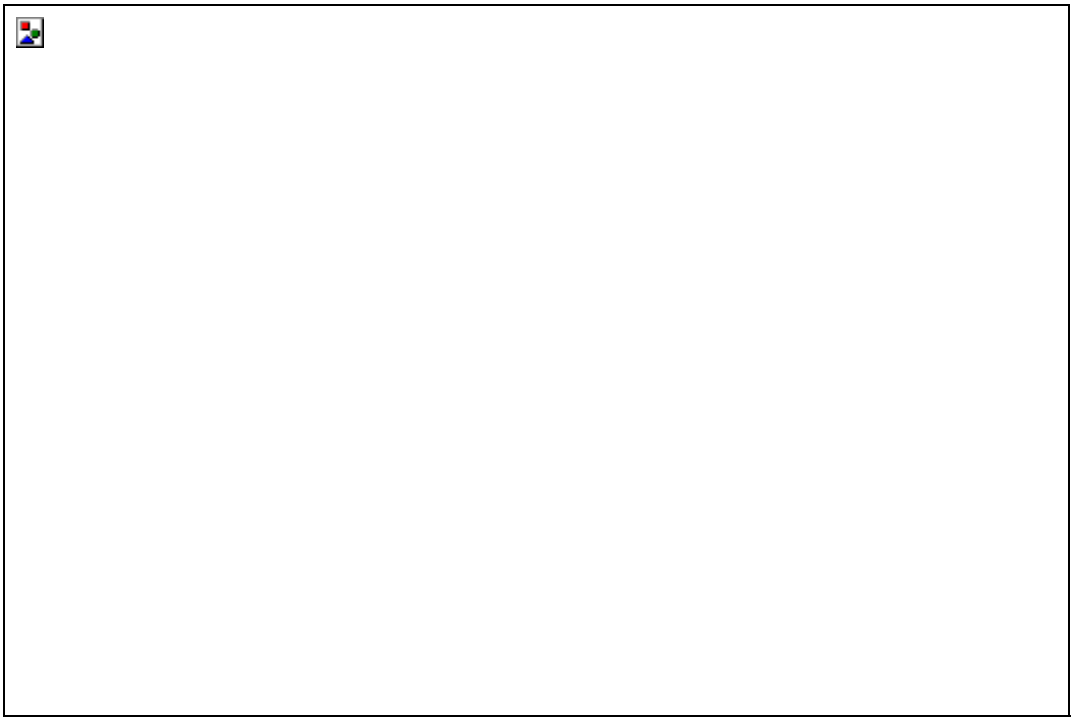


this slide shows the format of a client process corresponding to, and on me same machine as, the connection-oriented server shown earlier. A stream socket is created in the Internet domain, and then the connection request is made to the server process. Clearly, the name of the server socket must be known t the client, but as the processes are on the same machine it can be deduced. After the connect() has completed, the client can make its request of the server.

Below is the function to perform client I/O, and the function to construct the remote server address structure for an Internet domain socket:

```
void vRemoteSrvName(PSTR pszHost,PSTR pszService,int nPort, PSOCKADDR_IN psin)

PHOSTENT phe;
psin->sin_family=AF_INET;
psin->sin_port=htons(nPortlD);
psin->sin_addr.s_addr=INADDR_ANY;

if (pszHostName==NULL || pszHostName[0] == 0) {

        char szHostName[MAX _COMPUTERNAME_LENGTH + l];

        gethostname(szHostName,sizeof(szHostName) );
        phe=gethostbyname (szHostName);
}
else {
        long addr=inet_addr(pszHostName);
        phe=gethostbyaddr( (LPSTR) &addr, sizeof (addr) PF_INET);

        if (phe==NULL)
                phe=gethostbyname (pszHostName);
}
if (phe)
        memcpy( (LPSTR)&(psin->sin_ addr) ,phe->h_ addr,phe->h length);

        if (pszServiceName&&pszServlceName [0] ) ) {
                PSERVENT pse=getservbyname(pszServiceName, "tcp");
                if (pse)
```

```
                        psin->sin_port=pse->s_port;
        }
}
```



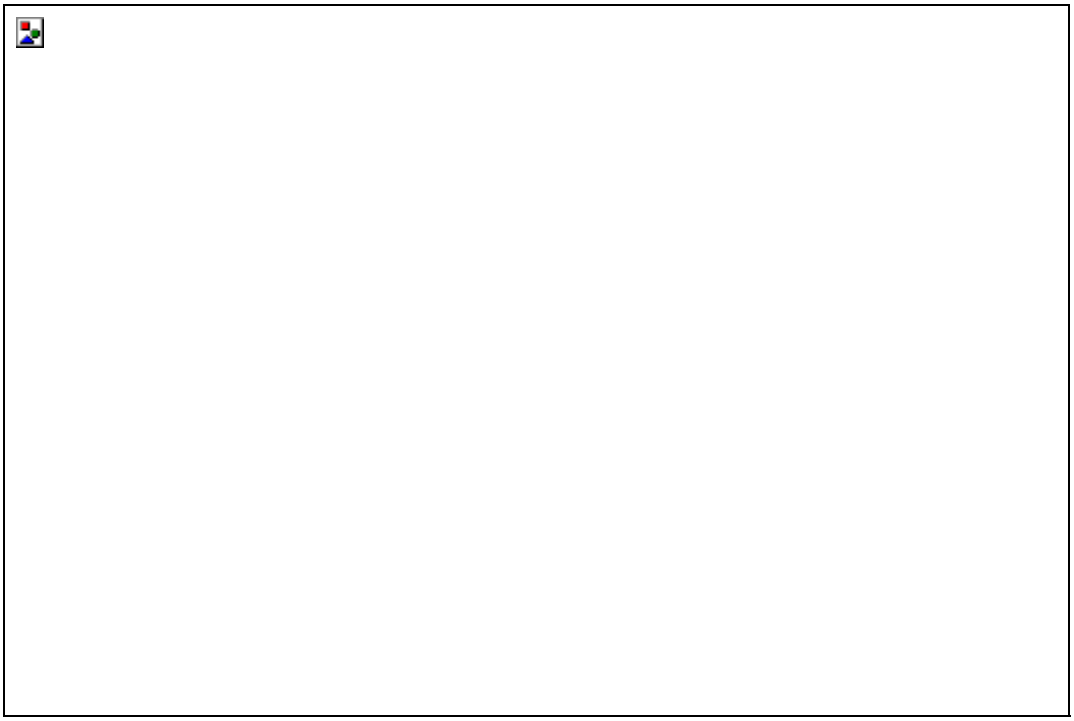
This example shows the basic structure of a server process when the communication is to be on a connectionless basis.

The socket is created and bound to an address as before. However, there is no need to prepare for connections or go through the rendezvous procedure; the server simply reads requests using `recvfrom()`, processes the requests and sends replies using `sendto()`.

Since the address of the sender is always returned from a call to `recvfrom()`, the server will always be able to send the reply to the correct place.

Notice that in this case we have not implemented a concurrent server such as the one in the connection-oriented example; the architecture of sockets makes this form of server more convenient to use with connectionless sockets.

The code for the `vLocalSrvName()` is the same as for the connection-oriented server.



The client in a connectionless link must bind itself to a local address, so that the server process knows where to send replies. So it must create the socket and then use `bind()`

Then the request is sent using `sendto()` and the reply received with `recvfrom()` In this case, it is unlikely that the client is interested in the "sender address" in this reply, so it would be common to pass NULL as the 'from' parameter.

It may be the case that the client could make a call to `connect()` before transmitting the request. This would restrict the link to the server process only, and then `send()` and `recv()` could be used.

The code for the `vRemoteSrvName()` is the same as for the connection-oriented client. The code for the `vLocalClientName()` is below.

```
void vLocalClientName(PSTR pszServiceName,int nportID,PSOCKADDR_IN psin ) {

psin—>sin_family=AF_INET;
psin->sin_port=htons(nPortID);
psin->sin_addr.s_addr=INADDR_ANY;

if (pszServiceName && pszServiceName [0] )
        PSERVENT pse=getservbyname(pszServiceName , , "tcp");

        if (pse)
                psin->sin_port=pse->s_port;

}
```