Winsock

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

```
struct in_addr
{
u_long s_addr;
};
```

-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.