Socket Functions

- We will now discuss the remaining WinSock functions necessary to create a complete and useful networked application.

Communication End-Points

- The **socket()** function creates an end-point of communication called a socket:

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

- **af** specifies the address family this socket uses. Use **PF_INET** or **AF_INET (in the case of a bind call)**, for the "protocol" or "address" family format.

- **type** is the type specification for the socket which is either **SOCK_STREAM**, for a **connection-oriented** byte stream, or **SOCK_DGRAM**, for **connectionless** datagram service.

- **protocol** specifies the particular protocol to use and is usually set to 0, which lets **socket()** use a default value (AF_INET).

- The protocol can be defaulted because the address family (**af**) and socket type (**type**) combination already uniquely describe a socket's protocol.

- If the family is **AF_INET** and the socket type is **SOCK_DGRAM**, the protocol must be **UDP**.

- Likewise, if the family is **AF_INET** and the socket type is **SOCK_STREAM**, the protocol must be **TCP**.

- The **socket()** call returns a socket descriptor upon success.

- On failure, **INVALID_SOCKET** is returned and **WSAGetLastError()** should be called to find out the reason for the error.

Naming the socket

- Creating a socket simply allocates a socket descriptor for your application form the list of available descriptors.
- The **bind()** function is used to give the socket a name, thus making the socket useful:

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

- **s** is the socket descriptor returned by socket.
- **addr** is a pointer to the address, or name, to assign to the socket.
- **namelen** is the length of the structure **addr** points to. This is usually set to **sizeof (sockaddr**).
- The **sockaddr** structure is defined as follows:

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

- The above structure reflects Berkeley Sockets' original goal of supporting multiple protocols.
- The **sockaddr** does not contain any TCP/IP-specific information. **sa_family** identifies the type of address that the data structure contains.
- Windows Sockets defines the constant **AF_INET** to represent **TCP/IP** address information.
- Windows Sockets also defines an equivalent structure to be used when working with TCP/IP addresses, **sockaddr_in** defined as:

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

- The **sin_family** field must always be **AF_INET**.
- The **sin_addr** field stores the **IP address** in **network byte order**,

- The **sin_port** field stores the **port number** in **network byte order**.
- The **sin_addr** is another structure of type **in_addr**. The **in_addr** structure allows a program to access individual portions of the IP address and is defined as:
- Within the **sockaddr** structure, the address may be the address assigned to a network interface on the host or **INADDR_ANY**, in which case WinSock uses any appropriate network interface address that a computer has.
- The **port number** can be specified or set to 0, in which case a unique port number will be assigned.
- Most programs use **INADDR_ANY** for the interface address.
- Server programs usually specify their own well-known port number, while client programs typically specify port 0 to allow the WinSock implementation to chose any unused descriptor value.

Listening for Connections

- Once the socket is named, it can be put it to real use by **listening for connections** to that socket from **client applications**.
- The **listen()** function, defined below, does this:

  int listen (SOCKET s, int backlog);

- **s** is the socket descriptor on which to listen for connections.
- **backlog** is a count of pending requests that may be queued up for incoming connections. It is between one and five inclusively. It acts as a safety net by preventing the WinSock layer from allocating too many resources.
- If many connection requests arrive in a short period of time and the application is unable to service them fast enough, the incoming queue will reach its maximum value.
- A remote station (**client**) initiating subsequent requests will receive a **WSAECONNREFUSED** error.

Accepting a Connection

- The next thing for the server to do following **listen()** is to accept a connection from a client.
- the **accept()** function, prototyped below, does this:

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

- **s** is the socket descriptor on which to accept a connection request.
- **addr** is a pointer to a sockaddr structure that will accept the address of the connecting client.

Client to Server Connections

- The **connect()** function establishes a connection with a remote server:

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

- **s** designates the socket to be used for the connection.
- **name** is the address of the server to connect to.
- If **s** is unbound at the time the connect function is invoked upon it, WinSock will bind the socket to an interface address and port number as if the bind() function had been called on it, specifying **INADDR_ANY** and port number **0**.

Sending Data on a Stream Socket

- The **send()** function, defined below, is used to send data over a stream socket:

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

- The **s** parameter specifies the socket on which the data should be queued for transmission.
- The **buf** pointer points to a buffer containing the data to be transmitted.
- **len** is the number of bytes in buf.
- **flags** can be used to alter the behavior of the send function beyond the various socket options that may have been set for the socket.

- The **flags** parameter is constructed by **logically OR-ing** together the constant values defined in the table below.

Flags	Description
MSG_DONTROUTE	Indicates that the data should not be routed. WinSock implementations may ignore this value as well as the corresponding socket option.
MSG_OOB	Used with stream sockets only. The data should be sent as out-of-band data (OOB data is delivered to the user independently of normal stream data).

- Note that this function will block if no buffer space is available for the data, unless the socket was put into non-blocking mode.
- If the socket is a stream socket and non-blocking, the send function accepts as much of the buffer as it can, and return the number of bytes accepted.
- The application should send the remaining bytes at a later time, as determined using the **WSAAsyncSelect** function.
- The **sendto()** function is similar to the **send()** function, and is prototyped as:

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

- The first four parameters are the same as those in **send().**
- The **to** and **tolen** parameters are used to indicate a remote host address to which the data should be sent.
- The **sendto()** function is typically used on an unconnected socket datagram socket.
- When used on a stream socket the **to** and **tolen** parameters are ignored, making the function identical to the **send()** function.

Receiving Data on a Stream Socket

- The **recv()** function, defined below, receives data from a socket.

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

- The **s** parameter specifies the socket from which to receive the data.
- The **buf** pointer points to a buffer into which the received data should be stored.
- **len** indicates the size of **buf**.
- **flags** can be used to alter the behavior of the **recv()** function beyond the various socket options that may have been set for the socket.
- The **flags** parameter is constructed by **logically OR-ing** together the constant values defined in the table below.

Flag	Description
MSG_PEEK	Used to peek at incoming data. The data at the head of the queue is copied to the return buffer but the data is not removed from the queue.
MSG_OOB	Used to process out-of-band data.

- If the socket is of type **SOCK_STREAM**, as much data as is queued and that fits into the buffer is returned.
- Subsequent calls to **recv()** will return data that follows in the stream.
- If there is no data to be read and the socket is in non-blocking mode, **recv()** returns **SOCKET_ERROR** and the code returned by **WSAGetLastError** is **WSAEWOULDBLOCK**.
- An application can determine when the data arrives using the **WSAAsyncSelect** function.
- The **recvfrom()** function is similar to the **recv()** function, and is prototyped as:

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

- The first four parameters are the same as those in **recv().**
- The **from** and **fromlen** parameters return the address of the sender of the datagram.
- The **fromlen** function is typically used on a datagram socket.

- When used on a stream socket the **from** and **fromlen** parameters are ignored, making the function identical to the **recv()** function.

Shutting down a Socket

- The **shutdown()** function is used to disable sends or receives on a socket. It is defined as:

  int shutdown (SOCKET s, int how);

- The **s** parameter specifies the socket on which to shut down transmission or reception.

- The **how** parameter specifies which operation to shutdown:

  - how = 0; all receives on the socket are disabled
  - how = 1, all sends on the socket are disabled
  - how = 2, all sends and receives are disabled

- The **shutdown()** function does not close a socket, and no resources are freed until the **closesocket()** function is invoked on the socket.

Closing a Socket

- The **closesocket()** function frees the resources associated with a socket descriptor when invoked on a socket:

  int closesocket (SOCKET s);

- There are several variables that determine the closing characteristics of a socket. These are determined by the socket's linger options as set with **setsockopt().**

- The following table summarizes these options.

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