Socket Options and Socket Ioctls

- Once a socket has been created its various attributes can be manipulated via **socket options** and ***ioctl*** commands to change or modify the behavior of the socket.

- Some of these options simply return information, while others affect the behavior of the socket in your application.

- We will discuss four Winsock functions: ***getsockopt***, ***setsockopt***, ***ioctlsocket***, and ***WSAIoctl***.

- Each function has numerous commands and options. Most of these ioctl commands and options are defined in either winsock.h or winsock2.h, depending on whether they are specific to winsock 1 or winsock 2.

Socket Options

- The ***getsockopt*** function is most frequently used to obtain information regarding the given socket and is prototyped as follows:

```
int getsockopt {
    SOCKET s,
    int level,
    int optname,
    char FAR* optval,
    int FAR* optlen
}
```

- The first parameter, **s**, is the socket on we want to perform the specified option. This must be a valid socket for the given protocol you are using.

- A number of options are specific to a particular **protocol** and **socket type**, while others pertain to all types of sockets. This ties in with the second parameter, **level**.

- An option of level **SOL_SOCKET** means it is a generic option that isn't necessarily specific to a given protocol.

- The **optname** parameter is the actual option you are interested in. These option names are constant values defined in the Winsock header files.

- The most common and protocol-independent options (such as those with the **SOL_SOCKET** level) are defined in winsock2.h.

- The **optval** and **optlen** parameters are the variables returned with the value of the desired option.

- The ***setsockopt*** function is used to set socket options on either a socket level or a protocol-specific level.

- The function is defined as

```
int setsockopt {
    SOCKET s,
    int level,
    int optname,
    const char FAR  * optval,
    int FAR * optlen;
}
```

- The parameters are the same as in ***getsockopt*** except that you pass in a value as the **optval** and **optlen** parameters, which are the values to be set for the specified option.

- The most common mistake associated with calling either ***getsockopt*** or ***setsockopt*** is attempting to obtain socket information for a socket whose underlying protocol does not possess that particular characteristic.

- For example, a socket of type **SOCK_STREAM** is not capable of broadcasting data; therefore, attempting to set or get the **SO_BROADCAST** option results in the error **WSAENOPROTOOPT.**

<u>SOL_SOCKET Option Level</u>

- The following is a brief description of the socket options that return information based on the characteristics of the socket itself and are not specific to the protocol of that socket.

- **SO_ACCEPTCONN**

    - If the socket has been put into listening mode by the ***listen*** function, this option returns TRUE. This indicates that the socket is in listening mode.

    - Sockets of type **SOCK_DGRAM** do not support this option.

- **SO_BROADCAST**

    - If the given socket has been configured for sending or receiving broadcast data, querying this socket option returns TRUE.

    - Use ***setsockopt*** with **SO_BROADCAST** to enable broadcast abilities on the socket. This option is valid for sockets that aren't of type **SOCK_STREAM**.

    - Broadcasting is the ability to send data so that every machine on the local subnet receives the data.

- The following code fragment illustrates how to send a broadcast message with UDP:

```
SOCKET          s;
BOOL            bBroadcast;
char            *sMsg = "This is a test";
SOCKADDR_IN     bcast;

s = WSASocket (
        AF_INET,
        SOCK_DGRAM,
        0,
        NULL,
        0,
        WSA_FLAG_OVERLAPPED
);

bBroadcast = TRUE;
setsockopt (
        s,
        SOL_SOCKET,
        SO_BROADCAST,
        (char *)&bBroadcast, sizeof (BOOL)
);

bcast.sin_family = AF_INET;
bcast.sin_addr.s_addr = inet_addr (INADDR_BROADCAST);
bcast.sin_port = htons (7000);

sendto (
        s,
        smsg,
        strlen(sMsg),
        0,
        (SOCKADDR *)&bcast,
        sizeof(bcast)
);
```

- UDP uses the defined type **INADDR_BROADCAST**, which is a special broadcast address (255.255.255.255).

- **SO_CONNECT_TIME**

  - This is a **Microsoft-specific** option that returns the number of seconds the socket has been connected.

- **SO_DEBUG**

  - This option enables or disables debug information from a Windows Sockets application. If TRUE, debug output is enabled.

- **SO_DONTLINGER**
    - This option enables or disables immediate return from the ***closesocket()*** function call. If TRUE, SO_LINGER is disabled.
    - Used in protocols that support graceful socket connection closure, so that if one or both sides close the socket, any data still pending or in transmission will be sent or received by both parties.
    - Sockets of type **SOCK_DGRAM** do not support this option.

- **SO_DONTROUTE**
    - This option tells the underlying network stack to ignore the routing table and to send the data out on the interface the socket is bound to. If TRUE, messages are sent directly to the network interface without consulting the routing table.
    - For example, if you create a UDP socket and bind it to interface A and then send a packet destined for a machine on the network attached to interface B, the packet will in fact be routed so that it is sent on interface B.

- **SO_ERROR**
    - This option returns the current socket error status and then clears it. It returns and resets the per-socket-based error code, which is different from the per-thread-based error code that is handled using the ***WSAGetLastError*** and ***WSASetLastError*** function calls.

- **SO_EXCLUSIVEADDRUSE**
    - If TRUE, the local port that the socket is bound to cannot be reused by another process.
    - This option is the complement of **SO_REUSEADDR,** which will be described later. This option exists to prevent other processes from using the **SO_REUSEADDR** on a local address that your application is using.

- **SO_KEEPALIVE**

  - This option enables or disables the keep-alive mechanism on a stream socket. If TRUE, the socket is configured to send "**keep-alive**" messages on the session.

  - Typically, we need to send or receive data to detect a problem on a stream connection. If neither side in a connection is actively sending data, there is normally no network traffic between the two hosts.

  - With this option enabled on an idle connection, the TCP/IP stack periodically sends an acknowledgement request call a **keep-alive**. When a TCP/IP stack receives a keep-alive, it simply acknowledges it.

  - If a sender does not receive an acknowledgement for a keep-alive, it will attempt to retransmit the keep-alive packet. If none of the retransmissions are acknowledged, the sender will reset the connection.

  - **RFC 1122** specifies two hours as the default period a TCP/IP stack should wait between sending keep-alive packets. This makes the keep-alive mechanism less than ideal for most applications.

  - In effect, this means that it may take up to two hours before an application with keep-alives enabled can detect a failed connection. Most TCP/IP stack vendors ignored the RFC 1122 two-hour default recommendation.

  - Rather than using **SO_KEEPALIVE,** designers implement their own keep-alive mechanism at the application level. This is only possible however if you control the code for both the server and the client.

- **SO_LINGER**

  - This option enables or disables immediate return from the ***closesocket()*** function call. It is disabled by default, thus enabling the SO_DONTLINGER option.

  - A call to ***getsockopt*** with this socket option returns the current linger times in a **linger** structure, which is defined as:

    ```
    struct linger {
        u_short l_onoff;
        u_short l_linger;
    }
    ```

  - A nonzero value for **l_onoff** means that lingering is enabled, while **l_linger** is the timeout in seconds, at which point any pending data to be sent or received is discarded and the connection with the peer is reset.

  - Conversely, an application can call ***setsockopt*** to turn lingering on and specify the length of time before discarding any queued data. This is accomplished by setting the desired values in a variable of type **struct linger**.

  - If SO_LINGER is set with a zero timeout interval (that is, the linger structure member l_onoff is not 0 and l_linger is 0), ***closesocket*** is not blocked, even if queued data has not yet been sent or acknowledged.

- This is called a hard, or abortive, close because the socket's virtual circuit is reset immediately and any unsent data is lost. Any receive call on the remote side of the circuit fails with **WSAECONNRESET.**

- If SO_LINGER is set with a nonzero timeout interval on a blocking socket, the ***closesocket*** call blocks on a blocking socket until the remaining data has been sent or until the timeout expires.

- This is called a graceful disconnect. if the timeout expires before all data has been sent, the Windows Sockets implementation terminates the connection before ***closesocket*** returns.

- **SO_MAX_MSG_SIZE**

  - This is a get-only socket option that indicates the maximum outbound (send) size of a message for message-oriented socket types as implemented by a particular service provider.

  - It has no meaning for byte-stream-oriented sockets. There is no provision for finding the maximum inbound message size.

- **SO_OOBINLINE**

  - By default, out-of-band (**OOB**) data is not "inlined". That means a call to a receive function (with the appropriate **MSG_OOB** flag set) returns the OOB data in a single call.

  - If this option is set, the OOB data appears within the data stream returned from a receive call, and a call to ***ioctlsocket*** with the **SIOCATMARK** option is required to determine which byte is the OOB data.

  - Sockets of type **SOCK_DGRAM** do not support this option.

- **SO_RCVBUF**

  - This option either returns the size or sets the **size of the buffer** allocated to this socket for receiving data.

  - When a socket is created, a default send buffer and a receive buffer are assigned to the socket for sending and receiving data.

  - One possible reason for changing the buffer size is to specifically tailor buffer sizes according to an application's behavior. For example, when writing code to receive UDP datagrams, applications generally make the receive buffer size an even multiple of the datagram size.

  - For overlapped I/O, setting the buffer sizes to 0 can increase performance in certain situations: when these buffers are nonzero, an extra memory copy is involved in moving data from the system buffer to the user-supplied buffer.

- If there is no intermediate buffer, data is immediately copied to the user-supplied buffer. The one caveat is that this is efficient only with multiple outstanding receive calls.

- Posting only a single receive can hurt performance, as the local system cannot accept any incoming data unless you have a buffer posted and ready to receive the data.

- **SO_REUSEADDR**

  - This option enables or disables the ability to reuse a local socket name on more than one socket. If TRUE, the socket can be bound to an address already in use by another socket or to an address in the TIME_WAIT state.

  - Two separate sockets cannot bind to the same local interface (and port, in the case of TCP/IP) to await incoming connections. If two sockets are actively listening on the same port, the behavior is undefined as to which socket will receive notification of an incoming connection.

  - The **SO_REUSEADDR** option is most useful in TCP when a server shuts down or exits abnormally so that the local address and port are in the TIME_WAIT state, which prevents any other sockets from binding to that port.

  - By setting this option, the server can listen on the same local interface and port when it is restarted.

- **SO_SNDBUF**

  - This option either returns the size or sets the size of the buffer allocated to this socket for sending data.

  - When a socket is created, a default send buffer and a receive buffer are assigned to the socket for sending and receiving data.

  - As with **SO_RCVBUF**, you can use the **SO_SNDBUF** option to set the size of the send buffer to 0. The advantage of the buffer size being 0 for blocking send calls is that when the call completes you know that your data is on the wire.

  - Also, as in the case of a receive operation with a zero-length buffer, there is no extra memory copy of your data to system buffers. The drawback is that you lose the pipelining gained by the default stack buffering when the send buffers are nonzero in size.

- In other words, if you have a loop performing sends, the local network stack can copy your data to a system buffer to be sent when possible (depending on the 1/0 model being used).

- **SO_TYPE**
  - The **SO_ TYPE** option is a get-only option that simply returns the socket type of the given socket.
  - The possible socket types are SOCK_DGRAM, SOCK_STREAM, SOCK_SEQPACKET, SOCK_RDM, and SOCK_RAW.

- **SO_SNDTIMEO**
  - The **SO_ SNDTIMEO** option sets the timeout value on a blocking socket when calling a Winsock send function.
  - The timeout value is an integer in milliseconds that indicates how long the send function should block when attempting to send data.
  - To use this option an application must use the ***WSASocket*** function to create the socket and specify **WSA_FLAG_OVERLAPPED** as part of ***WSASocket***'s **dwflags** parameter.
  - Subsequent calls to any Winsock send function *(**send, sendto, WSASend, WSASendTo,*** and so on) block only for the amount of time specified.
  - If the send operation cannot complete within that time, the call fails with error 10060 *(***WSAETIMEOUT***).*

- **SO_RCVTIMEO**
  - This option sets the receive timeout value on a blocking socket. The timeout value is an integer in milliseconds that indicates how long a Winsock receive function should block when attempting to receive data.
  - To use this option an application must use the ***WSASocket*** function to create the socket and specify **WSA_FLAG_OVERLAPPED** as part of ***WSASocket***'s **dwflags** parameter.
  - Subsequent calls to any Winsock receive function *(**recv, recvfrom, WSARecv, WSARecvFrom,*** and so on) block only for the amount of time specified.
  - If no data arrives within that time, the call fails with the error 10060 (**WSAETIMEOUT**).

IPPROTO_IP Option Level

- The socket options on the **IPPROTO_IP** level pertain to attributes specific to the IP protocol, such as modifying certain fields in the IP header and adding a socket to an IP multicast group.

- Many of these options are declared in both winsock.h and winsock2.h with different values.

- For Winsock 2, you should include the Winsock 2 header file and link with WS2_32.lib. This is especially relevant to multicasting.

- **IP_OPTIONS**

  - This flag allows you to get and set various IP options within the IP header. Some of the possible options are described below.

  - Security and handling restrictions RFC 1108.

  - Record route. Each router adds its IP address to the header.

  - Timestamp. Each router adds its IP address and time.

  - Loose source routing. The packet is required to visit each IP address listed in the option header.

  - Strict source routing. The packet is required to visit only those IP addresses listed in the option header.

  - Be aware that hosts and routers do not support all of these options.

  - Once the option is set, it applies to any packets sent on the given socket. At any pointer thereafter, an application can call ***getsockopt*** with IP_OPTIONS to retrieve which options were set.

  - However, this will not return any data filled into the option-specific buffers. In order to retrieve the data set in the IP options, either the socket must be created as a raw socket (**SOCK_RAW**) or the **IP_HDRINCL** option should be set- in which case, the IP header is returned along with data after a call to a Winsock receive function.

- **IP_HDRINCL**
    - Setting this option to TRUE causes the send function to include the IP header ahead of the data it's sending and causes the receive function to include the IP header as part of the data.
    - Thus, when calling a Winsock send function, an application must include the entire IP header ahead of the data and fill each field of the IP header correctly.
    - The easiest way to include an IP header with the data that you are sending is to define a structure that contains the IP header and the data, and pass the structure into the Winsock ***send*** call.

- **IP_TOS**
    - This option allows an application to set one or more of the Type of Service (TOS) bits in the IP header.
    - The field is 8 bits long and is broken into three parts: a 3-bit precedence field (which is ignored), a 4-bit TOS field, and the remaining bit (which must be 0).
    - The 4 TOS bits are minimize delay, maximize throughput, maximize reliability, and minimize monetary costs. Only 1 bit can be set at a time. All 4 bits being 0 implies normal service.
    - RFC 1340 specifies the recommended bits to set for various standard applications such as TCP, SMTP, NNTP, and so on.

- **IP_TTL**
    - The Time-To-Live (TTL) field in an IP header is used to limit the number of routers through which the datagram can pass.
    - The idea behind this is that each router that the datagram passes through decrements the datagram's TTL value by 1. When the value equals 0, the datagram is discarded.

- **IP_MULTICAST_IF**
    - The IP multicast interface (IF) option gets and sets the local interface from which any multicast data sent by the local machine will be sent.
    - This option is only of interest on machines that have more than one connected network interface card.

- **IP_MULTICAST_TTL**
  - This option gets/sets the time to live on multicast packets for this socket.
  - Similar to the IP_TTL, this option performs the same function except that it applies only to multicast data sent using the given socket.
  - Again, the purpose of the TTL is to prevent routing loops, but in the case of multicasting, setting the TTL narrows the scope of how far the data will travel.
  - Therefore, multicast group members must be within “range" to receive datagrams. The default TTL value for multicast datagrams is 1.

- **IP_MULTICAST_LOOP**
  - If TRUE, data sent to a multicast address will be echoed to the socket's incoming buffer.
  - By default, when you send IP multicast data, the data will be looped back to the sending socket if it is also a member of that multicast group.
  - If you this option is set to FALSE, any data sent will not he posted to the incoming data queue for the socket.

- **IP_ADD_MEMBERSHIP**
  - This option is the method for adding a socket to an IP multicast group.
  - This is done by creating a socket of address family **AF_INET** and the socket type **SOCK_DGRAM** with the ***socket*** function.
  - To add the socket to a multicast group, use the following structure:

```
struct ip_mreq
{
    struct in_addr imr_multiaddr;
    struct in_addr imr_interface;
 };
```

  - The **imr_multiaddr** member is the binary address of the multicast group to join, while **imr_interface** is the local interface that multicast data should be sent out and received on.

- **IP_DROP_MEMBERSHIP**
  - Removes the socket from the given IP group membership
  - This option is the opposite of **IP_ADD_MEMBERSHIP**.
  - By calling this option with an **ip_mreq** structure that contains the same values used when joining the given multicast group, the socket will be removed from the given group.

- **IP_DONTFRAGMENT**
    - This flag tells the network not to fragment the IP datagram during transmission.
    - However, if the size of the IP datagram exceeds the maximum transmission unit (MTU) and the IP don't fragment flag is set within the IP header, the datagram will be dropped and an ICMP error message ("fragmentation needed but don't fragment bit set") will be returned to the sender.

IPPROTO_TCP Option Level

- There is only one option belonging to the **IPPROTO_TCP** level.

- **TCP_NODELAY**
- The option is valid only for sockets that are stream sockets (**SOCK_STREAM**) and belong to family **AF_INET**.
- If TRUE, the Nagle algorithm (RFC 896)is disabled on the socket. In order to increase performance and throughput by minimizing overhead, the system implements the Nagle algorithm.
- When an application requests to send a chunk of data, the system might hold on to that data for a while and wait for other data to accumulate before actually sending it on the wire.
- If no other data accumulates in a given period of time, the data will be sent regardless. This results in more data in a single TCP packet, as opposed to smaller chunks of data in multiple TCP packets.
- The overhead is that the TCP header for each packet is 20 bytes long. Sending a couple bytes here and there with a 20-byte header is wasteful.
- The other part of this algorithm is the delayed acknowledgments. Once a system receives TCP data it must send an ACK to the peer.
- However, the host will wait to see whether it has data it is sending to the peer so that it can piggyback the ACK on the data to be sent resulting in one less packet on the network.
- The purpose of this option is to disable the Nagle algorithm, as its behavior can be detrimental in a few cases.
- This algorithm can adversely affect any network application that sends relatively small amounts of data and expects a timely response.

- A classic example is Telnet. Typically the user hits only a few keystrokes per second. The Nagle algorithm would make such a session seem sluggish and unresponsive.

IOCTLSOCKET AND WSAIoctl

- The socket ***ioctl*** functions are used to control the behavior of I/O upon the socket, as well as obtain information about I/O pending on that socket.

- The first function, ***ioctlsocket,*** originated in the Winsock 1 specification and is declared as:

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

- The parameter **s** is the socket descriptor to act upon, while **cmd** is a predefined flag for the I/O control command to execute.

- The last parameter, **argp**, is a pointer to a variable specific to the given command.

- Winsock 2 introduced a new **ioctl** function that adds quite a few new options. First, it breaks the single **argp** parameter into a set of input parameters for values passed into the function and a set of output parameters used to return data from the call.

- Additionally, the function call can use overlapped I/O. This function is ***WSAIoctl,*** which is defined as:

```
int WSAIoctl (
        SOCKET s,
        DWORD dwIoControlCode,
        LPVOID lpvInBuffer,
        DWORD cbInBuffer,
        LPVOID lpvOutBuffer,
        DWORD cbOutBuffer,
        LPDWORD lpcbBytesReturned,
        LPWSAOVERLAPPED lpOverlapped,
        LPWSAOVERLAPPED_COMPLETION_ROUTINE lpCompletionRoutine
);
```

- The first two parameters are the same as those in ***ioctlsocket.***

- The second two parameters, **lpvInBuffer** and **cbInBuffer**, describe the input parameters.

- The **lpvInBuffer** parameter is a pointer to the value passed in, while **cbInBuffer** is the size of that data in bytes.

- Likewise, **lpvOutBuffer and cbOutBuffer** are used for any data returned from the call.

- The **lpvOutBuffer** parameter points to a data buffer in which any information returned is placed.

- The **cbOutBuffer** parameter is the size in bytes of the buffer passed in as l**pvOutBuffer**.

- The seventh parameter, **lpcbBytesReturned**, is the number of bytes actually returned.

- The last two parameters, **lpOverlapped** and **lpCompletionRoutine**, are used when calling this function with overlapped I/O.

Standard Ioctl Commands

- The following three ioctl commands are the most common and are carryovers from the Unix world.

- They are available on all Win32 platforms. Also, these three commands can be called using either ***ioctlsocket*** or ***WSAIoctl.***

- **FIONBIO**

    - This command enables or disables **nonblocking** mode on a socket. By default, all sockets are blocking sockets upon creation.

    - When you call ***ioctlsocket*** with the **FIONBIO** ioctl command, set **argp** to pass a pointer to an unsigned long integer whose value is nonzero if nonblocking mode is to be enabled.

    - The value 0 places the socket in blocking mode. If you use ***WSAIoctl*** instead, simply pass the unsigned long integer in as the **lpvInBuffer** parameter.

    - Calling the ***WSAAsyncSelect*** *or* ***WSAEventSelect*** function automatically sets a socket to nonblocking mode. If either of these functions has been called, any attempt to set the socket back to blocking mode fails with **WSAEINVAL**.

    - To set the socket back to blocking mode, an application must first disable ***WSAAsyncSelect*** by calling it with the **lEvent** parameter equal to 0, or disable ***WSAEventSelect*** by calling it with the **lNetworkEvents** parameter equal to 0.

- **FIONREAD**
    - This command determines (returns) the amount of data that can be read atomically from the socket.
    - For ***ioctlsocket***, the argp value returns with an unsigned integer that will contain the number of bytes to be read.
    - When using ***WSAIoctl***, the unsigned integer is returned in **lpvOutBuffer**.
    - If the socket is a stream-oriented (**SOCK_STREAM**), **FIONREAD** returns the total amount of data that can be read in a single receive call.
    - Note that using this or any other message-peeking mechanism is not always guaranteed to return the correct amount.
    - When this ioctl command is used on a datagram socket (**SOCK_DGRAM**), the return value is the size of the first message queued on the socket.

- **SIOCATMARK**
    - Determines whether Out-Of-Band data has been read.
    - When a socket has been configured to receive Out-Of-Band (OOB) data and has been set to receive this OOB data inline (by setting the **SO_OOBINLINE** socket option), this ioctl command returns a Boolean value indicating TRUE if the OOB data is to be read next.
    - Otherwise, FALSE is returned and the next receive operation returns all or some of the data that precedes the OOB data.
    - For ***ioctlsocket***, **argp** returns with a pointer to a Boolean variable, while for ***WSAIoctl***, the pointer to the Boolean variable returns in **lpvOutBuffer**.