Socket Options

The *getsockopt* function is most frequently used to obtain information regarding the given socket. The prototype for this function is as follows:

```
int getsockopt (
    SOCKET s,
    int level,
    int optname,
    char FAR* optval,
    int FAR* optlen
);
```

The first parameter, *s*, is the socket on which you 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. We say "necessarily" because not all protocols implement each socket option of level *SOL_SOCKET*. For example, *SO_BROADCAST* puts the socket into broadcast mode, but not all supported protocols support the notion of broadcast sockets. 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 Winsock.h and Winsock2.h. Each specific protocol has its own header file that defines options specific to it. Finally, the *optval* and *optlen* parameters are the variables returned with the value of the desired option. In most cases—but not all—the option value is an integer.

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 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 set for the specified option. As with *getsockopt*, *optval* is often, but not always, an integer. Consult each option for the specifics on what is passed as the option value.

The most common mistake associated with calling either *getsockopt* or *setsockopt* is attempting to obtain socket information for a socket whose underlying protocol doesn't 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*.

SOL_SOCKET Option Level

This section describes 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

optval *Type*	*Get/Set*	*Winsock Version*	*Description*
BOOL	Get only	1+	If *TRUE*, socket is in listening mode.

If the socket has been put into listening mode by the *listen* function, this option returns *TRUE*. Sockets of type *SOCK_DGRAM* do not support this option.

SO_BROADCAST

optval *Type*	*Get/Set*	*Winsock Version*	*Description*
BOOL	Both	1+	If *TRUE*, socket is configured for sending broadcast messages.

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. Of course, there must be some process on each machine that listens for incoming broadcast data. The drawback of broadcasting is that if many processes are all sending broadcast data, the network can become saturated and network performance suffers. In order to receive a broadcast message, you must enable the broadcast option and then use one of the datagram receive functions, such as *recvfrom* or *WSARecvfrom*. You can also connect the socket to the broadcast address by calling *connect* or *WSAConnect* and then use *recv* or *WSARecv*. For UDP broadcasts, you must specify a port number to send the datagram to; likewise, the receiver must request to receive the broadcast data on that port also. The following code example 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(5150);
sendto(s, sMsg, strlen(sMsg), 0, (SOCKADDR *)&bcast, sizeof(bcast));
```

For UDP, a special broadcast address exists to which broadcast data should be sent. This address is 255.255.255.255. A *#define* directive for *INADDR_BROADCAST* is provided to make things a bit simpler and easier to read.

AppleTalk is another protocol capable of sending broadcast messages. AppleTalk also has a special address used by broadcast data. You learned in Chapter 6 that an AppleTalk address has three parts: network, node, and socket (destination). For broadcasting, set the destination to *ATADDR_BROADCAST* (0xFF), which causes the datagram to be sent to all endpoints on the given network.

Normally, you need to set only the *SO_BROADCAST* option when sending broadcast datagrams. To receive a broadcast datagram, you need to be listening only for incoming datagrams on that specified port. However, on Windows 95 when using IPX, the receiving socket must set the *SO_BROADCAST* option in order to receive broadcast data, as described in Knowledge Base article Q137914, which can be found at http://support.microsoft.com/support/search. This is a bug in Windows 95.

SO_CONNECT_TIME

optval *Type*	*Get/Set*	*Winsock Version*	*Description*
int	Get only	1+	Returns the number of seconds the socket has been connected

SO_CONNECT_TIME is a Microsoft-specific option that returns the number of seconds a connection has been established. The most frequent use of this option is with the *AcceptEx* function. *AcceptEx* requires that a valid socket handle be passed for the incoming client connection. This option can be called on the client's *SOCKET* handle to determine whether the connection has been made and how long the connection as been established. If the socket is not currently connected, the value returned is 0xFFFFFFFF.

SO_DEBUG

optval *Type*	*Get/Set*	*Winsock Version*	*Description*
BOOL	Both	1+	If *TRUE*, debug output is enabled.

Winsock service providers are encouraged (but not required) to supply output debug information if the *SO_DEBUG* option is set by an application. How the debug information is presented depends on the underlying service provider's implementation. To turn debug information on, call *setsockopt* with *SO_DEBUG* and a Boolean variable set to *TRUE*. Calling *getsockopt* with *SO_DEBUG* returns *TRUE* or *FALSE* if debugging is enabled or disabled, respectively. Unfortunately, no Win32 platform currently implements the *SO_DEBUG* option, as described in Knowledge Base article Q138965. No error is returned when the option is set, but the underlying network provider ignores the option.

SO_DONTLINGER

optval *Type*	*Get/Set*	*Winsock Version*	*Description*
BOOL	Both	1+	If *TRUE*, *SO_LINGER* is disabled.

For protocols that support graceful socket connection closure, a mechanism is implemented 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. It is possible, with *setsockopt* and the *SO_LINGER* option, to change this behavior so that after a specified period of time, the socket and all its resources will be torn down. Any pending or arriving data associated with that socket is discarded and the peer's connection reset (*WSAECONNRESET*). The *SO_DONTLINGER* option can be checked to ensure that a linger period has not been set. Calling *getsockopt* with *SO_DONTLINGER* will return a Boolean *TRUE* or *FALSE* if a linger value is set or not set, respectively. A call to *setsockopt* with *SO_DONTLINGER* disables lingering. Sockets of type *SOCK_DGRAM* do not support this option.

SO_DONTROUTE

optval *Type*	*Get/Set*	*Winsock Version*	*Description*
BOOL	Both	1+	If *TRUE*, messages are sent directly to the network interface without consulting the routing table.

The *SO_DONTROUTE* 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. 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. Using *setsockopt* with the Boolean value *TRUE* prevents this because the packet goes out on the bound interface. The *getsockopt* function can be called to determine whether routing is enabled (which it is by default).

Calling this option on a Win32 platform will succeed; however, the Microsoft provider silently ignores the request and always uses the routing table to determine the appropriate interface for outgoing data.

SO_ERROR

optval *Type*	*Get/Set*	*Winsock Version*	*Description*
int	Get only	1+	Returns the error status

The *SO_ERROR* option 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. A successful call using the socket does not reset the per-socket-based error code returned by the *SO_ERROR* option. Calling this option will not fail; however, the error value is not always updated immediately, so there is a possibility of this option returning 0 (indicating no error). It is best to use *WSAGetLastError* unless it is absolutely necessary to rely on the individual error code.

SO_EXCLUSIVEADDRUSE

optval *Type*	*Get/Set*	*Winsock Version*	*Description*
BOOL	Both	2+	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 we will describe shortly. This option exists to prevent other processes from using the *SO_REUSEADDR* on a local address that your application is using. If two separate processes are bound to the same local address (assuming that *SO_REUSEADDR* is set earlier), which of the two sockets receives notifications for incoming connections is not defined. This is extremely unfortunate if your application is mission-critical. The *SO_EXCLUSIVEADDRUSE* option locks down the local address to which the socket is bound, so if any other process tries to use *SO_REUSEADDR* with the same local address, that process fails. Administrator rights are required to set this option. It is available only on Windows 2000.

SO_KEEPALIVE

optval *Type*	*Get/Set*	*Winsock Version*	*Description*
BOOL	Both	1+	If *TRUE*, socket is configured to send keepalive messages on the session.

For a TCP-based socket, an application can request that the underlying service provider enable the use of keepalive packets on TCP connections by turning on the *SO_KEEPALIVE* socket option. On Win32 platforms, keepalives are implemented in accordance with section 4.2.3.6 of RFC 1122. If a connection is dropped as the result of keepalives, the error code *WSAENETRESET* is returned to any calls in progress on the socket, and any subsequent calls will fail with *WSAENOTCONN*. For the exact implementation details, consult the RFC. The important thing to note here is that keepalives are sent at intervals no less than 2 hours apart. The 2-hour keepalive time is configurable via the Registry; however, changing the default value changes the keepalive behavior for all TCP connections on the system, which is generally discouraged. Another solution is to implement your own keepalive strategy. Chapter 7 discusses this kind of strategy. Sockets of type *SOCK_DGRAM* do not support this option.

The Registry keys for keepalives are *KeepAliveInterval* and *KeepAliveTime*. Both keys store values of type *REG_DWORD* in milliseconds. The former key is the interval separating keepalive retransmissions until a response is received; the latter entry controls how often TCP sends a keepalive packet in an attempt to verify that an ideal connection is still valid. In Windows 95 and Windows 98, these keys are located under the following Registry path:

```
\HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services\VxD\MSTCP
```

In Windows NT and Windows 2000, store the keys under

```
\HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services\TCPIP\Parameters
```

With Windows 2000, a new socket ioctl command—*SIO_KEEPALIVE_VALS*—allows you to change the keepalive value and interval on a per-socket basis, as opposed to a system-wide basis. This ioctl command is described later in this chapter.

SO_LINGER

optval *Type*	*Get/Set*	*Winsock Version*	*Description*
struct linger	Both	1+	Sets or gets the current linger values

SO_LINGER controls the action taken when unsent data is queued on a socket and a *closesocket* is performed. 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, you 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*. When setting a linger value with *setsockopt,* you must set the *l_onoff* field of the structure to a nonzero value. To turn lingering off once it has been enabled, you can call *setsockopt* with the *SO_LINGER* option and the *l_onoff* field of the *linger* structure set to 0, or you can call *setsockopt* with the *SO_DONTLINGER* option, passing the value *TRUE* for the *optval* parameter. Sockets of type *SOCK_DGRAM* do not support this option.

Setting the linger option directly affects how a connection behaves when the *closesocket* function is called. Table 9-1 lists these behaviors.

Table 9-1. *Linger options*

Option	*Interval*	*Type of Close*	*Wait for Close?*
SO_DONTLINGER	Not applicable	Graceful	No
SO_LINGER	0	Hard	No
SO_LINGER	Nonzero	Graceful	Yes

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

optval *Type*	*Get/Set*	*Winsock Version*	*Description*
unsigned int	Get only	2+	The maximum size of a message for a message-oriented socket

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

optval *Type*	*Get/Set*	*Winsock Version*	*Description*
BOOL	Both	1+	If *TRUE*, any out-of-band data is returned in the normal data stream.

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. Unfortunately, this socket option is broken on all current Win32 implementations. See Chapter 7 for more details on OOB data.

SO_PROTOCOL_INFO

optval *Type*	*Get/Set*	*Winsock Version*	*Description*
WSAPROTOCOL_INFO	Get only	2+	The protocol characteristics for the protocol that the socket is bound to

This is another get-only option that fills in the supplied *WSAPROTOCOL_INFO* structure with the characteristics of the protocol associated with the socket. See Chapter 6 for a description of the *WSAPROTOCOL_INFO* structure and its member fields.

SO_RCVBUF

optval *Type*	*Get/Set*	*Winsock Version*	*Description*
int	Both	1+	Gets or sets the per-socket buffer size for receive operations

This is a simple option that either returns the size or sets the size of the buffer allocated to this socket for receiving data. When a socket is created, a send buffer and a receive buffer are assigned to the socket for sending and receiving data. When requesting to set the receive buffer size to a value, the call to *setsockopt* can succeed even when the implementation does not provide the entire amount requested. To ensure that the requested buffer size is allocated, call *getsockopt* to get the actual size allocated. All Win32 platforms can get or set the receive buffer size except Windows CE, which does not allow you to change the value—you can get only the receive buffer size.

One possible reason for changing the buffer size is to specifically tailor buffer sizes according to your application's behavior. For example, when writing code to receive UDP datagrams, you should 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. For more information, see the section called "Other Issues" under the completion port I/O model in Chapter 8.

SO_REUSEADDR

optval *Type*	*Get/Set*	*Winsock Version*	*Description*

BOOL	Both	1+	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.

By default, a socket cannot be bound to a local address that is already in use; however, occasionally it is necessary to reuse an address in this way. Remember from Chapter 7 that each connection is uniquely identified by the combination of its local and remote addresses. As long as the address to which you are connecting is unique in the slightest respect (such as a different port number in TCP/IP), the binding will be allowed.

The only exception is for a listening socket. 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

optval Type	Get/Set	Winsock Version	Description
BOOL	Both	1+	*TRUE* (nonzero) means socket is configured for sending broadcast messages.

This is a simple option that either returns the size or sets the size of the buffer allocated to this socket for sending data. When a socket is created, a send buffer and a receive buffer are assigned to the socket for sending and receiving data. When requesting to set the size of the send buffer, the call to *setsockopt* can succeed even when the implementation does not provide the entire amount requested. To ensure that the requested buffer size is allocated, call *getsockopt* to get the actual size allocated. All Win32 platforms can get or set the send buffer size except Windows CE, which does not allow you to change the value—you can get only the receive buffer size.

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 I/O model being used). On the other hand, if your application is concerned with other logistics, disabling the send buffers can save you a few machine instructions in the memory copy. For additional information, see the section entitled "Other Issues" under the completion port I/O model in Chapter 8.

SO_TYPE

optval Type	Get/Set	Winsock Version	Description
int	Get only	1+	Returns the socket type (e.g., *SOCK_DGRAM*, *SOCK_STREAM*, etc.) of the given socket

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

optval Type	Get/Set	Winsock Version	Description
int	Both	1+	Gets/sets the timeout value (in milliseconds) associated with sending data on the socket

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. If you need to use the *SO_SNDTIMEO* option and you use the *WSASocket* function to create the socket, you must 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 (*WSAETIMEDOUT*).

For performance reasons, this option was disabled in Windows CE 2.1. If you attempt to set this option, the option is silently ignored and no failure is returned. Previous versions of Windows CE do implement this option.

SO_RCVTIMEO

optval *Type*	*Get/Set*	*Winsock Version*	*Description*
int	Both	1+	Gets/sets the timeout value (in milliseconds) associated with receiving data on the socket

The *SO_RCVTIMEO* 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. If you need to use the *SO_RCVTIMEO* option and you use the *WSASocket* function to create the socket, you must 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 (*WSAETIMEDOUT*).

For performance reasons, this option was disabled in Windows CE 2.1. If you attempt to set this option, it is silently ignored and no failure returns. Previous versions of Windows CE do implement this option.

SO_UPDATE_ACCEPT_CONTEXT

optval *Type*	*Get/Set*	*Winsock Version*	*Description*
SOCKET	Both	1+	Gets/sets the timeout value (in milliseconds) associated with receiving data on the socket

This option is a Microsoft-specific extension most commonly used in conjunction with the *AcceptEx* function. The unique characteristic of this function is that it is part of the Winsock 1 specification and allows the use of overlapped I/O for an accept call. The function takes the listening socket as a parameter as well as a socket handle that becomes the accepted client. This socket option must be set in order for the characteristics of the listening socket to be carried over to the client socket. This is particularly important for Quality of Service (QOS)-enabled listening sockets. In order for the client socket to be QOS-enabled, this option must be set. To set this option on a socket, use the listening socket as the *SOCKET* parameter to *setsockopt* and pass the accepting socket handle (for example, the client handle) as *optval*. This option is specific to Windows NT and Windows 2000.

SOL_APPLETALK Option Level

The following options are socket options specific to the AppleTalk protocol and can be used only with sockets created using *socket* or *WSASocket* with the *AF_APPLETALK* flag. A majority of the options listed here deal with either setting or obtaining AppleTalk names. For more information on the AppleTalk address family, refer back to Chapter 6. Some AppleTalk socket options—such as *SO_DEREGISTER_NAME*—have more than one option name. In such cases, all the option's names can be used interchangeably.

SO_CONFIRM_NAME

optval *Type*	*Get/Set*	*Winsock Version*	*Description*

WSH_NBP_TUPLE	Get only	1	Confirms that the given AppleTalk name is bound to the given address

The *SO_CONFIRM_NAME* option is used to verify that a given AppleTalk name is bound to the supplied address. This results in a Name Binding Protocol (NBP) lookup request being sent to the address to verify the name. If the call fails with the error *WSAEADDRNOTAVAIL*, the name is no longer bound to the address given.

SO_DEREGISTER_NAME, SO_REMOVE_NAME

optval *Type*	*Get/Set*	*Winsock Version*	*Description*
WSH_REGISTER_NAME	Set only	1	Deregisters the given name from the network

This option is used to deregister a name from the network. If the name does not currently exist on the network, the call will return indicating success. Refer to the section entitled "Registering an AppleTalk Name" in Chapter 6 for a description of the *WSH_REGISTER_NAME* structure, which is simply another name for the *WSH_NBP_NAME* structure.

SO_LOOKUP_MYZONE, SO_GETMYZONE

optval *Type*	*Get/Set*	*Winsock Version*	*Description*
char *	Get only	1	Returns the default zone on the network

This option returns the default zone on the network. The *optval* parameter to *getsockopt* should be a character string of at least 33 characters. Remember that the maximum length of an NBP name is *MAX_ENTITY_LEN*, which is defined as 32. The extra character is required for the null terminator.

SO_LOOKUP_NAME

optval *Type*	*Get/Set*	*Winsock Version*	*Description*
WSH_LOOKUP_NAME	Get only	1	Looks up a specified NBP name and returns the matching tuples of names and NBP information

This option is used to look up a specified name on the network (for example, when a client wants to connect to a server). The well-known textual name must be resolved to an AppleTalk address before a connection can be established. See the section "Resolving an AppleTalk Name" in Chapter 6 for sample code on how to look up an AppleTalk name.

One thing to be aware of is that upon successful return, the *WSH_NBP_TUPLE* structures occupy the space in the supplied buffer after the *WSH_LOOKUP_NAME* information. That is, you should supply *getsockopt* with a buffer large enough to hold the *WSH_LOOKUP_NAME* information at the start of the buffer and a number of *WSH_NBP_TUPLE* structures in the remaining space. Figure 9-1 illustrates how the buffer should be prepared prior to the call (with respect to *WSH_LOOKUP_NAME*) and where the *WSH_NBP_TUPLE* structures are placed upon return.

Figure 9-1. SO_LOOKUP_NAME *buffer*

SO_LOOKUP_ZONES, SO_GETZONELIST

optval *Type*	*Get/Set*	*Winsock Version*	*Description*

WSH_LOOKUP_ZONES	Get only	1	Returns zone names from the Internet zone lists

This option requires a buffer large enough to contain a *WSH_LOOKUP_ZONES* structure at the head. Upon successful return, the space after the *WSH_LOOKUP_ZONES* structure contains the list of null-terminated zone names. The following code demonstrates how to use the *SO_LOOKUP_ZONES* option:

```
PWSH_LOOKUP_NAME     atlookup;
PWSH_LOOKUP_ZONES    zonelookup;
char                 cLookupBuffer[4096],
                     *pTupleBuffer = NULL;

atlookup = (PWSH_LOOKUP_NAME)cLookupBuffer;
zonelookup = (PWSH_LOOKUP_ZONES)cLookupBuffer;
ret = getsockopt(s, SOL_APPLETALK, SO_LOOKUP_ZONES, (char *)atlookup,
    &dwSize);
pTupleBuffer = (char *)cLookupBuffer + sizeof(WSH_LOOKUP_ZONES);
for(i = 0; i < zonelookup->NoZones; i++)
{
    printf("%3d: '%s'\n", i + 1, pTupleBuffer);
    while (*pTupleBuffer++);
}
```

SO_LOOKUP_ZONES_ON_ADAPTER, SO_GETLOCALZONES

optval *Type*	*Get/Set*	*Winsock Version*	*Description*
WSH_LOOKUP_ZONES	Get only	1	Returns a list of zone names known to the given adapter name

This option is similar to *SO_LOOKUP_ZONES* except that you specify the adapter name for which you want to obtain a list of zones local to the network that that adapter is connected to. Again, you must supply a sufficiently large buffer that has a *WSH_LOOKUP_ZONES* structure at the head. The returned list of null-terminated zone names begins in the space after the *WSH_LOOKUP_ZONES* structure. Additionally, the name of the adapter must be passed in as a UNICODE string (*WCHAR*).

SO_LOOKUP_NETDEF_ON_ADAPTER, SO_GETNETINFO

optval *Type*	*Get/Set*	*Winsock Version*	*Description*
WSH_LOOKUP_NETDEF_ON_ADAPTER	Set only	1	Returns the seeded values for the network as well as the default zone

This option returns the seeded values for the network numbers and a null-terminated ANSI string containing the default zone for the network on the indicated adapter. The adapter is passed as a UNICODE (*WCHAR*) string following the structure and is overwritten by the default zone upon function return. If the network is not seeded, the network range 1-0xFFFE is returned and the null-terminated ANSI string contains the default zone "*."

SO_PAP_GET_SERVER_STATUS

optval *Type*	*Get/Set*	*Winsock Version*	*Description*
WSH_PAP_GET_SERVER_STATUS	Get only	1	Returns the PAP status from a given server

This option gets the Printer Access Protocol (PAP) status registered on the address specified in *ServerAddr* (usually obtained via an NBP lookup). The four reserved bytes correspond to the four reserved bytes in the PAP status packet. These will be in network byte order. A PAP status string can be arbitrary and is set with the option *SO_PAP_SET_SERVER_STATUS*, which we'll explain later in this chapter. The *WSH_PAP_GET_SERVER_STATUS* structure is defined as

```
#define    MAX_PAP_STATUS_SIZE        255
#define    PAP_UNUSED_STATUS_BYTES        4

typedef struct _WSH_PAP_GET_SERVER_STATUS
{
    SOCKADDR_AT            ServerAddr;
    UCHAR                  Reserved[PAP_UNUSED_STATUS_BYTES];
    UCHAR                  ServerStatus[MAX_PAP_STATUS_SIZE + 1];

} WSH_PAP_GET_SERVER_STATUS, *PWSH_PAP_GET_SERVER_STATUS;
```

The following code snippet is a quick example of how to request the PAP status. The length of the status string is the first byte of the *ServerStatus* field.

```
WSH_PAP_GET_SERVER_STATUS  status;
int                        nSize = sizeof(status);

status.ServerAddr.sat_family = AF_APPLETALK;
ret = getsockopt(s, SOL_APPLETALK, SO_PAP_GET_SERVER_STATUS,
    (char *)&status, &nSize);
```

SO_PAP_PRIME_READ

optval *Type*	*Get/Set*	*Winsock Version*	*Description*
char []	Set only	1	This call primes a read on a PAP connection so that the sender can actually send the data.

When this option is called on a socket describing a PAP connection, it enables the remote client to send the data without the local application having called *recv* or *WSARecvEx*. After this option is set, the application can block on a *select* call and then the actual reading of the data can occur. The *optval* parameter to this call is the buffer that is to receive the data, which must be at least *MIN_PAP_READ_BUF_SIZE* (4096) bytes in length. This option allows support for nonblocking sockets on the read-driven PAP protocol. Note that for each buffer you want to read, you must make a call to *setsockopt* with the *SO_PAP_PRIME_READ* option.

SO_PAP_SET_SERVER_STATUS

optval *Type*	*Get/Set*	*Winsock Version*	*Description*
char []	Set only	1	Sets the status to be sent if another client requests the status

A client can request to obtain the PAP status by using *SO_PAP_GET_SERVER_STATUS*. This option can be used to set the status so that if clients request the PAP status, the buffer submitted to the set command will be returned on the get command. The status is a buffer of at most 255 bytes containing the status of the associated socket. If the set option is called with a null buffer, the previous status value set is erased.

SO_REGISTER_NAME

optval *Type*	*Get/Set*	*Winsock Version*	*Description*
WSH_REGISTER_NAME	Set only	1	Registers the given name on the AppleTalk network

This option is used to register the supplied name on the AppleTalk network. If the name already exists on the network, the error *WSAEADDRINUSE* is returned. Refer to Chapter 6 for a description of the *WSH_REGISTER_NAME* structure.

SOL_IRLMP Option Level

The *SOL_IRLMP* level deals with the IrDA protocol, whose address family is *AF_IRDA*. One important thing to keep in mind when using IrDA socket options is that the implementation of infrared sockets varies among platforms. Because Windows CE first offered IR support, it does not have all the options available that were introduced later, in Windows 98 or Windows 2000. In this section, each option is followed by the platforms it is supported on.

IRLMP_9WIRE_MODE

optval *Type*	*Get/Set*	*Winsock Version*	*Description*
BOOL	Both	1+	Gets/sets IP options within the IP header

This is another rarely used option needed to communicate with Windows 98 via IrCOMM, which is at a lower level than the level at which IrSock normally operates. In 9-wire mode, each TinyTP or IrLMP packet contains an additional 1-byte IrCOMM header. To accomplish this through the socket interface, you need to first get the maximum PDU size of an IrLMP packet with the *IRLMP_SEND_PDU_LEN* option. The socket is then put in 9-wire mode with *setsockopt* before connecting or accepting a connection. This tells the stack to add the 1-byte IrCOMM header (always set to 0) to each outgoing frame. Each *send* must be of a size less than the maximum PDU length to leave room for the added IrCOMM byte. IrCOMM is beyond the scope of this book. This option is available on Windows 98 and Windows 2000.

IRLMP_ENUMDEVICES

optval *Type*	*Get/Set*	*Winsock Version*	*Description*
DEVICELIST	Get only	1+	Returns a list of IrDA device IDs for IR-capable devices within range

Because of the nature of infrared networking, devices capable of communicating are mobile and can move in and out of range. This option "queries" which IR devices are within range, and to connect to another device you must perform this step to obtain the device ID for each device you want to connect to.

The *DEVICELIST* structures are different on the various platforms that support IrSock because the latest platforms that added support also added functionality. Recall that Windows CE offered IrSock support first, and Windows 98 and Windows 2000 added support shortly thereafter. The *DEVICELIST* structure definition for Windows 98 and Windows 2000 is

```
typedef struct _WINDOWS_DEVICELIST
{
    ULONG                       numDevice;
    WINDOWS_IRDA_DEVICE_INFO    Device[1];
} WINDOWS_DEVICELIST, *PWINDOWS_DEVICELIST, FAR *LPWINDOWS_DEVICELIST;

typedef struct _WINDOWS_IRDA_DEVICE_INFO
{
    u_char  irdaDeviceID[4];
    char      irdaDeviceName[22];
```

```
    u_char  irdaDeviceHints1;
    u_char  irdaDeviceHints2;
    u_char  irdaCharSet;
} WINDOWS_IRDA_DEVICE_INFO, *PWINDOWS_IRDA_DEVICE_INFO,
  FAR *LPWINDOWS_IRDA_DEVICE_INFO;
```

In Windows CE, the *DEVICELIST* structure is defined as

```
typedef struct _WCE_DEVICELIST
{
    ULONG                       numDevice;
    WCE_IRDA_DEVICE_INFO        Device[1];
} WCE_DEVICELIST, *PWCE_DEVICELIST;

typedef struct _WCE_IRDA_DEVICE_INFO
{
    u_char  irdaDeviceID[4];
    char    irdaDeviceName[22];
    u_char  Reserved[2];
} WCE_IRDA_DEVICE_INFO, *PWCE_IRDA_DEVICE_INFO;
```

As you can see, the device information structure varies as well: *WCE_IRDA_DEVICE_INFO* for Windows CE and *WINDOWS_IRDA_DEVICE_INFO* for Windows 98 and Windows 2000. Each of these structures contains a field, *irdaDeviceID*, which is a 4-byte identification tag used to uniquely identify that device. You need this field to fill out the *SOCKADDR_IRDA* structure used to connect to a specific device or to manipulate or obtain an Information Access Service (IAS) entry with the options *IRLMP_IAS_SET* and *IRLMP_IAS_QUERY*.

When you call *getsockopt* to enumerate infrared devices, the *optval* parameter must be a *DEVICELIST* structure. The only requirement is that the *numDevice* field be set to 0 at first. The call to *getsockopt* does not return an error if no IR devices are discovered. After a call, the *numDevice* field should be checked to see whether it is greater than 0, which means that one or more devices were found. The *Device* field returns with a number of structures equal to the value returned in *numDevice*.

IRLMP_EXCLUSIVE_MODE

optval *Type*	***Get/Set***	***Winsock Version***	***Description***
BOOL	Both	1+	If *TRUE*, socket connection is in exclusive mode.

This option isn't normally used by user applications, as it bypasses the TinyTP layer in the IrDA stack and communicates directly with IrLMP. If you are really interested in using this option, you should consult the IrDA specification at *http://www.irda.org*. This option is available on Windows CE and Windows 2000.

IRLMP_IAS_QUERY

optval *Type*	***Get/Set***	***Winsock Version***	***Description***
IAS_QUERY	Get only	1+	Queries IAS on a given service and class name for its attributes

This socket option is the complement of *IRLMP_IAS_SET*, as it retrieves information about a class name and its service. Before making the call to *getsockopt*, you must first fill out the *irdaDeviceID* field to the device you are querying. Set the *irdaAttribName* field to the property string on which you want to retrieve its value. The most common query would be for the LSAP-SEL number; its property string is "IrDA:IrLMP:LsapSel." Next, you need to set the *irdaClassName* field to the name of the service that the given property string applies to. Once these fields are filled, make the call to *getsockopt.* Upon success, the *irdaAttribType* field indicates which field in the union to obtain

the information from. Use the identifiers in Table 9-2 to decode this entry. The most common error is *WSASERVICE_NOT_FOUND*, which is returned when the given service is not found on that device. This option is available on Windows CE, Windows 98, and Windows 2000.

IRLMP_IAS_SET

optval *Type*	***Get/Set***	***Winsock Version***	***Description***
IAS_QUERY	Set only	1+	Sets an attribute value for a given class name and attribute

IAS is a dynamic service registration entity that can be queried and modified. The *IRLMP_IAS_SET* option allows you to set a single attribute for a single class within the local IAS. As with *IRLMP_ENUMDEVICES*, there are separate structures for Windows CE and for Windows 98 and Windows 2000. The structures for Windows 98 and Windows 2000 are

```
typedef struct _WINDOWS_IAS_QUERY
{
    u_char     irdaDeviceID[4];
    char       irdaClassName[IAS_MAX_CLASSNAME];
    char       irdaAttribName[IAS_MAX_ATTRIBNAME];
    u_long     irdaAttribType;
    union
    {
        LONG   irdaAttribInt;
        struct
        {
            u_long   Len;
            u_char   OctetSeq[IAS_MAX_OCTET_STRING];
        } irdaAttribOctetSeq;
        struct
        {
            u_long   Len;
            u_long   CharSet;
            u_char   UsrStr[IAS_MAX_USER_STRING];
        } irdaAttribUsrStr;
    } irdaAttribute;
} WINDOWS_IAS_QUERY, *PWINDOWS_IAS_QUERY, FAR *LPWINDOWS_IAS_QUERY;
```

The IAS query structure for Windows CE is

```
typedef struct _WCE_IAS_QUERY
{
    u_char     irdaDeviceID[4];
    char       irdaClassName[61];
    char       irdaAttribName[61];
    u_short    irdaAttribType;
    union
    {
        int    irdaAttribInt;
        struct
        {
            int      Len;
            u_char   OctetSeq[1];
            u_char   Reserved[3];
        } irdaAttribOctetSeq;
```

```
        struct
        {
            int      Len;
            u_char   CharSet;
            u_char   UsrStr[1];
            u_char   Reserved[2];
        } irdaAttribUsrStr;
    } irdaAttribute;
} WCE_IAS_QUERY, *PWCE_IAS_QUERY;
```

Table 9-2 provides the different constants for the *irdaAttribType* field, which indicates which type the attribute belongs to. The last two entries are not values that you can set, but values that a call to *getsockopt* with an *IRLMP_IAS_QUERY* socket option can return in the *irdaAttribType* field. These are included in the table for the sake of completeness.

Table 9-2. *IAS attribute types*

irdaAttribType *Value*	***Field to Set***
IAS_ATTRIB_INT	*IrdaAttribInt*
IAS_ATTRIB_OCTETSEQ	*IrdaAttribOctetSeq*
IAS_ATTRIB_STR	*IrdaAttribUsrStr*
IAS_ATTRIB_NO_CLASS	None
IAS_ATTRIB_NO_ATTRIB	None

In order to set a value, you must fill in *irdaDeviceID* to the IR device on which to modify the IAS entry. Also, *irdaAttribName* must be set to the class on which to set the attribute, while *irdaClassName* usually refers to the service on which to set the attribute. Remember that with IrSock, socket servers are services registered with IAS that have an associated LSAP-SEL number used by clients to connect to the server. The LSAP-SEL number is an attribute associated with that service. To modify the LSAPSEL number in the service's IAS entry, set the *irdaDeviceID* field to the device ID on which the service is running. Set the *irdaAttribName* field to the property string "IrDA:IrLMP:LsapSel" and the *irdaClassName* field to the name of the service (for example, "MySocketServer"). From there, set *irdaAttribType* to *IAS_ATTRIB_INT* and *irdaAttribInt* to the new LSAP-SEL number. Of course, changing the service's LSAP-SEL number is a bad idea, but this example is for illustration only.

IRLMP_IRLPT_MODE

optval *Type*	***Get/Set***	***Winsock Version***	***Description***
BOOL	Both	1+	If *TRUE*, socket is configured to communicate to IR capable printers.

It is possible to connect to an infrared printer using Winsock and send data to be printed. This is accomplished by putting the socket in IRLPT mode before establishing the connection. Simply pass the Boolean value *TRUE* to this option after socket creation. You can use the option *IRLMP_ENUMDEVICES* to find infrared-capable printers within range. Note that some legacy IR printers do not register themselves with IAS; you might need to connect to them directly using the "LSAP-SEL-*xxx*" identifier. See Chapter 6 and its discussion of IrSock for more details on bypassing IAS. This option is available on Windows CE and Windows 2000.

IRLMP_SEND_PDU_LEN

optval *Type*	***Get/Set***	***Winsock Version***	***Description***

int	Get only	1+	Gets the maximum PDU length

This option retrieves the maximum Protocol Data Unit (PDU) size needed when using the option *IRLMP_9WIRE_MODE*. See the description of *IRLMP_9WIRE_MODE* for more information about this option, which is available on Windows CE and Windows 2000.

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. Note that if you load Winsock 1, you must include the correct header and link with Wsock32.lib. Likewise for Winsock 2, you should include the Winsock 2 header file and link with Ws2_32.lib. This is especially relevant to multicasting, which is available under both versions. Multicasting is supported on all Win32 platforms except Windows CE, in which it is available on versions 2.1 and later.

IP_OPTIONS

optval *Type*	***Get/Set***	***Winsock Version***	***Description***
char []	Both	1+	Gets/sets IP options within the IP header

This flag allows you to set various IP options within the IP header. Some of the possible options are

- **Security and handling restrictions** RFC 1108.
- **Record route** Each router adds its IP address to the header (see the ping sample in Chapter 13).
- **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.

When setting an IP option, the data that you pass into the *setsockopt* call follows the structure shown in Figure 9-2. The IP option header can be up to 40 bytes in length.

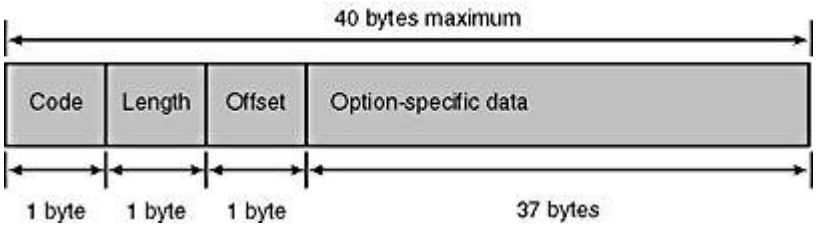

Figure 9-2. *IP option header format*

The code field indicates what type of IP option is present. For example, the value 0x7 represents the record route option. Length is simply the length of the option header, while offset is the offset value into the header where the data portion of the header begins. The data portion of the header is specific to the particular option. In the following code snippet, we set up the record route option. Notice that we declare a structure (*struct ip_option_hdr*) that contains the first three option values (code, length, offset), and then we declare the option-specific data as an array of nine unsigned long integers, as the data to be recorded is up to nine IP addresses. Remember that the maximum size of the IP option header is 40 bytes; however, our structure occupies only 39 bytes. The system will pad the header to a multiple of a 32-bit word for you (up to 40 bytes).

```
struct ip_option_hdr
{
    unsigned char    code;
    unsigned char    length;
    unsigned char    offset;
    unsigned long    addrs[9];
} opthdr;

...

ZeroMemory((char *)&opthdr, sizeof(opthdr));
opthdr.code = 0x7;
opthdr.length = 39;
opthdr.offset = 4;  // Offset to first address (addrs)
ret = setsockopt(s, IPPROTO_IP, IP_OPTIONS, (char *)&opthdr,
        sizeof(opthdr));
```

Once the option is set, it applies to any packets sent on the given socket. At any pointer thereafter, you 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

optval Type	***Get/Set***	***Winsock Version***	***Description***
BOOL	Both	2+	If *TRUE*, IP header is submitted with data to be sent and returned from data that is read.

Setting the *IP_HDRINCL* 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 you call a Winsock send function, you must include the entire IP header ahead of the data and fill each field of the IP header correctly. Figure 9-3 shows what the IP header should look like. This option is available only on Windows 2000.

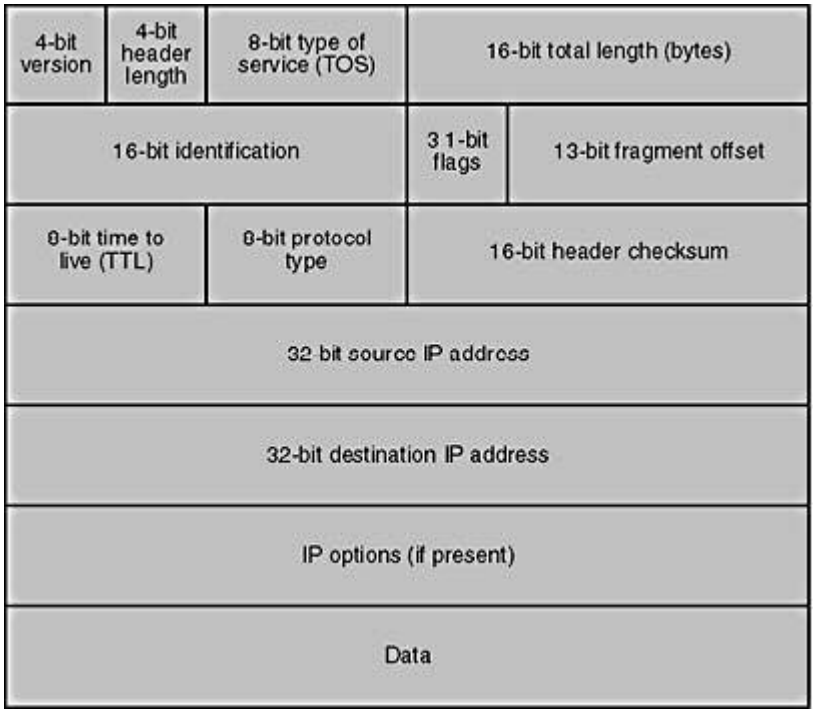

Figure 9-3. *The IP header*

The first field of the header is the IP version, which is currently version 4. The header length is the number of 32-bit words in the header. An IP header must always be a multiple of 32 bits. The next field is the type of service field. Consult the *IP_TOS* socket option, discussed next, for additional information. The total length field is the length, in bytes, of the IP header and data. The identification field is a unique value used to identify each IP packet sent. Normally, the system increments this value with each packet sent. The flags and fragmentation offset fields are used when IP packets are fragmented into smaller packets. The time to live field, or TTL, limits the number of routers through which the packet can pass. Each time a router forwards the packet, the TTL is decremented by 1. Once the TTL is 0, the packet is dropped. This limits the amount of time a packet can be live on the network. The protocol field is used to demultiplex incoming packets. Some of the valid protocols that use IP addressing are TCP, UDP, IGMP, and ICMP. The checksum is the 16-bit one's complement sum of the header. It is calculated over the header only and not the data. The next two fields are the 32-bit IP source and destination addresses. The IP options field is a variable length field that contains optional information, usually regarding security or routing.

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. See Chapter 13 for more details and an example of this option. This option works only on Windows 2000.

IP_TOS

optval *Type*	*Get/Set*	*Winsock Version*	*Description*
int	Both	1+	IP type of service

The type of service (TOS) is a field present in the IP header that is used to signify certain characteristics about a packet. 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. Additionally, RFC 1349 contains some corrections to the original RFC.

Interactive applications—such as Rlogin or Telnet—might want to minimize delay. Any kind of file transfer—such as FTP—is interested in maximum throughput. Maximum reliability is used by network management (SNMP) and routing protocols. Finally, Usenet news (NNTP) is an example of minimizing monetary costs. The *IP_TOS* option is not

available on Windows CE.

There is an additional issue when you attempt to set the TOS bits on a QOS-enabled socket. Because IP precedence is used by QOS to differentiate levels of service, it is undesirable to allow developers the ability to change these values. As a result, when you call *setsockopt* with *IP_TOS* on a QOS-enabled socket, the QOS service provider intercepts the call to verify whether the change can take place. See Chapter 12 for more information about QOS.

IP_TTL

optval *Type*	*Get/Set*	*Winsock Version*	*Description*
int	Both	1+	IP time-to-live parameter

The time-to-live (TTL) field is present in an IP header. Datagrams use the TTL field to limit the number of routers through which the datagram can pass. The purpose of this limitation is to prevent routing loops in which datagrams can spin in circles forever. 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. This option is not available on Windows CE.

IP_MULTICAST_IF

optval *Type*	*Get/Set*	*Winsock Version*	*Description*
unsigned long	Both	1+	Gets/sets the local interface for multicast data to be sent from

The IP multicast interface (IF) option 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 (network card, modem, and so on). The *optval* parameter should be an unsigned long integer representing the binary IP address of the local interface. The function *inet_addr* can be used to convert a string IP dotted decimal address to an unsigned long integer, as in the following sample:

```
DWORD    mcastIF;

// First join socket s to a multicast group
mcastIF = inet_addr("129.113.43.120");
ret = setsockopt(s, IPPROTO_IP, IP_MULTICAST_IF, (char *)&mcastIF,
    sizeof(mcastIF));
```

IP_MULTICAST_TTL

optval *Type*	*Get/Set*	*Winsock Version*	*Description*
int	Both	1+	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

optval *Type*	*Get/Set*	*Winsock Version*	*Description*

BOOL	Both	1+	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 set this option to *FALSE*, any data sent will not be posted to the incoming data queue for the socket.

IP_ADD_MEMBERSHIP

optval *Type*	*Get/Set*	*Winsock Version*	*Description*
struct ip_mreq	Set only	1+	Adds the socket to the given IP group membership

This option is the Winsock 1 method of 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;
};
```

In the *ip_mreq* structure, *imr_multiaddr* 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. See Chapter 11 for more information about valid multicast addresses. The *imr_interface* field is either the binary IP address of a local interface or the value *INADDR_ANY*, which can be used to select the default interface.

IP_DROP_MEMBERSHIP

optval *Type*	*Get/Set*	*Winsock Version*	*Description*
struct ip_mreq	Set only	1+	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 *s* will be removed from the given group. Again, Chapter 11 contains much more detailed information on IP multicasting.

IP_DONTFRAGMENT

optval *Type*	*Get/Set*	*Winsock Version*	*Description*
BOOL	Both	1+	If *TRUE*, do not fragment IP datagrams.

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. This option is not available on Windows CE.

IPPROTO_TCP Option Level

There is only one option belonging to the *IPPROTO_TCP* level. The option is valid only for sockets that are stream sockets (*SOCK_STREAM*) and belong to family *AF_INET*. This option is available on all versions of Winsock and is supported on all Win32 platforms.

TCP_NODELAY

optval *Type*	*Get/Set*	*Winsock Version*	*Description*
BOOL	Both	1+	If *TRUE*, the Nagle algorithm 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. Of course, 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. Telnet is an interactive application that allows the user to log on to a remote machine and send it commands. Typically the user hits only a few keystrokes per second. The Nagle algorithm would make such a session seem sluggish and unresponsive.

NSPROTO_IPX Option Level

These socket options are Microsoft-specific extensions to the Window IPX/SPX Windows Sockets interface, provided for use as necessary for compatibility with existing applications. They are otherwise not recommended for use, as they are guaranteed to work only over the Microsoft IPX/SPX stack. An application that uses these extensions might not work over other IPX/SPX implementations. These options are defined in WSNwLink.h, which should be included after Winsock.h and Wsipx.h.

IPX_PTYPE

optval *Type*	*Get/Set*	*Winsock Version*	*Description*
int	Both	1+	Gets/sets the IPX packet type

This option sets or gets the IPX packet type. The value specified in the *optval* argument will be set as the packet type on every IPX packet sent from this socket. The *optval* parameter is an integer.

IPX_FILTERPTYPE

optval *Type*	*Get/Set*	*Winsock Version*	*Description*
int	Both	1+	Gets/sets the IPX packet type to filter on

This option gets or sets the receive filter packet type. Only IPX packets with a packet type equal to the value specified in the *optval* argument are returned on any receive call; packets with a packet type that does not match are discarded.

IPX_STOPFILTERPTYPE

optval *Type*	*Get/Set*	*Winsock Version*	*Description*
int	Set only	1+	Removes the filter on the given IPX packet

You can use this option to stop filtering on packet types that are set with the *IPX_FILTERPTYPE* option.

IPX_DSTYPE

optval *Type*	*Get/Set*	*Winsock Version*	*Description*
int	Both	1+	Sets/gets the value of the datastream field in the SPX header

This option gets or sets the value of the datastream field in the SPX header of every packet sent.

IPX_EXTENDED_ADDRESS

optval *Type*	*Get/Set*	*Winsock Version*	*Description*
BOOL	Both	1+	If *TRUE*, enables extended addressing on IPX packets

This option enables or disables extended addressing. On sends, it adds the element *unsigned char sa_ptype* to the *SOCKADDR_IPX* structure, making the total length of the structure 15 bytes. On receives, the option adds both the *sa_ptype* and *unsigned char sa_flags* elements to the *SOCKADDR_IPX* structure, making the total length 16 bytes. The current bits defined in *sa_flags* are

- **0x01** The received frame was sent as a broadcast.
- **0x02** The received frame was sent from this machine.

IPX_RECVHDR

optval *Type*	*Get/Set*	*Winsock Version*	*Description*
BOOL	Both	1+	If *TRUE*, returns IPX header with receive call

If this option is set to true, any Winsock receive call returns the IPX header along with the data.

IPX_MAXSIZE

optval *Type*	*Get/Set*	*Winsock Version*	*Description*
int	Get only	1+	Returns the maximum IPX datagram size

Calling *getsockopt* with this option returns the maximum IPX datagram size possible.

IPX_ADDRESS

optval *Type*	*Get/Set*	*Winsock Version*	*Description*
IPX_ADDRESS_DATA	Get only	1+	Returns information regarding an IPX-capable adapter

This option queries for information about a specific adapter that IPX is bound to. In a system with *n* adapters, the adapters are numbered 0 through *n* - 1. To find the number of IPX-capable adapters on the system, use the *IPX_MAX_ADAPTER_NUM* option with *getsockopt*, or call *IPX_ADDRESS* with increasing values of *adapternum* until it fails. The *optval* parameter points to an *IPX_ADDRESS_DATA* structure defined as

```
typedef struct _IPX_ADDRESS_DATA
{
    INT     adapternum;  // Input: 0-based adapter number
```

```
    UCHAR   netnum[4];   // Output: IPX network number
    UCHAR   nodenum[6];  // Output: IPX node address
    BOOLEAN wan;         // Output: TRUE = adapter is on a WAN link
    BOOLEAN status;      // Output: TRUE = WAN link is up (or adapter
                         // is not WAN)
    INT     maxpkt;      // Output: max packet size, not including IPX
                         // header
    ULONG   linkspeed;   // Output: link speed in 100 bytes/sec
                         // (i.e., 96 == 9600 bps)
} IPX_ADDRESS_DATA, *PIPX_ADDRESS_DATA;
```

IPX_GETNE TINFO

optval *Type*	*Get/Set*	*Winsock Version*	*Description*
IPX_NETNUM_DATA	Get only	1+	Returns information regarding a specific IPX network number

This option obtains information about a specific IPX network number. If the network is in IPX's cache, the option returns the information directly; otherwise, it issues RIP requests to find it. The *optval* parameter points to a valid *IPX_NETNUM_DATA* structure defined as

```
typedef struct _IPX_NETNUM_DATA
{
    UCHAR  netnum[4];  // Input: IPX network number
    USHORT hopcount;   // Output: hop count to this network, in machine
                       // order
    USHORT netdelay;   // Output: tick count to this network, in machine
                       // order
    INT    cardnum;    // Output: 0-based adapter number used to route
                       // to this net; can be used as adapternum input
                       // to IPX_ADDRESS
    UCHAR  router[6];  // Output: MAC address of the next hop router,
                       // zeroed if the network is directly attached
} IPX_NETNUM_DATA, *PIPX_NETNUM_DATA;
```

IPX_GETNETINFO_NORIP

optval *Type*	*Get/Set*	*Winsock Version*	*Description*
IPX_NETNUM_DATA	Both	1+	If *TRUE*, do not fragment IP datagrams.

This option is similar to *IPX_GETNETINFO* except that it does not issue RIP requests. If the network is in IPX's cache, it returns the information; otherwise, it fails. (See also *IPX_RERIPNETNUMBER*, which always issues RIP requests.) Like *IPX_GETNETINFO*, this option requires passing an *IPX_NETNUM_DATA* structure as the *optval* parameter.

IPX_SPXGETCONNECTIONSTATUS

optval *Type*	*Get/Set*	*Winsock Version*	*Description*
IPX_SPXCONNSTATUS_DATA	Get only	1+	Returns information regarding a connected SPX socket

This option returns information on a connected SPX socket. The *optval* parameter points to an *IPX_SPXCONNSTATUS_DATA* structure defined below. All numbers are in network (high-low) byte order.

```
typedef struct _IPX_SPXCONNSTATUS_DATA
{
    UCHAR  ConnectionState;
    UCHAR  WatchDogActive;
    USHORT LocalConnectionId;
    USHORT RemoteConnectionId;
    USHORT LocalSequenceNumber;
    USHORT LocalAckNumber;
    USHORT LocalAllocNumber;
    USHORT RemoteAckNumber;
    USHORT RemoteAllocNumber;
    USHORT LocalSocket;
    UCHAR  ImmediateAddress[6];
    UCHAR  RemoteNetwork[4];
    UCHAR  RemoteNode[6];
    USHORT RemoteSocket;
    USHORT RetransmissionCount;
    USHORT EstimatedRoundTripDelay; /* In milliseconds */
    USHORT RetransmittedPackets;
    USHORT SuppressedPacket;
} IPX_SPXCONNSTATUS_DATA, *PIPX_SPXCONNSTATUS_DATA;
```

IPX_ADDRESS_NOTIFY

optval *Type*	*Get/Set*	*Winsock Version*	*Description*
IPX_ADDRESS_DATA	Get only	1+	Asynchronously notifies when the status of an IPX adapter changes

This option submits a request to be notified when the status of an adapter that IPX is bound to changes, which typically occurs when a WAN line goes up or down. This option requires the caller to submit an *IPX_ADDRESS_DATA* structure as the *optval* parameter. The exception, however, is that the *IPX_ADDRESS_DATA* structure is followed immediately by a handle to an unsignaled event. The following pseudo-code illustrates one method for calling this option.

```
char  buff[sizeof(IPX_ADDRESS_DATA) + sizeof(HANDLE)];
IPX_ADDRESS_DATA *ipxdata;
HANDLE *hEvent;

ipxdata = (IPX_ADDRESS_DATA *)buff;
hEvent = (HANDLE *)(buff + sizeof(IPX_ADDRESS_DATA));
ipxdata->adapternum = 0;  // Set to the appropriate adapter
*hEvent = CreateEvent(NULL, TRUE, FALSE, NULL);
setsockopt(s, NSPROTO_IPX, IPX_ADDRESS_NOTIFY, (char *)buff,
sizeof(buff));
```

When the *getsockopt* query is submitted, it completes successfully. However, the *IPX_ADDRESS_DATA* structure pointed to by *optval* will not be updated at that point. Instead, the request is queued internally inside the transport, and when the status of an adapter changes, IPX locates a queued *getsockopt* query and fills in all the fields in the *IPX_ADDRESS_DATA* structure. It then signals the event pointed to by the handle in the *optval* buffer. If multiple *getsockopt* calls are submitted at once, different events must be used. The event is used because the call needs to be asynchronous; *getsockopt* does not currently support this.

WARNING

In the current implementation, the transport signals only one queued query for each status change. Therefore, only one service that uses a queued query should run at once.

IPX_MAX_ADAPTER_NUM

optval *Type*	*Get/Set*	*Winsock Version*	*Description*
int	Get only	1+	Returns the number of IPX adapters present

This option returns the number of IPX-capable adapters present on the system. If this call returns *n* adapters, the adapters are numbered 0 through *n* - 1.

IPX_RERIPNETNUMBER

optval *Type*	*Get/Set*	*Winsock Version*	*Description*
IPX_NETNUM_DATA	Get only	1+	Returns information regarding a network number

This option is related to IPX_GETNETINFO except that it forces IPX to reissue RIP requests even if the network is in its cache (but not if it is directly attached to that network). Like *IPX_GETNETINFO*, it requires passing an *IPX_NETNUM_DATA* structure as the *optval* parameter.

IPX_RECEIVE_BROADCAST

optval *Type*	*Get/Set*	*Winsock Version*	*Description*
BOOL	Set only	1+	If *TRUE*, do not receive broadcast IPX packets.

By default, an IPX socket is capable of receiving broadcast packets. Applications that do not need to receive broadcast packets should set this option to *FALSE*, which can cause better system performance. Note, however, that setting the option to *FALSE* does not necessarily cause broadcasts to be filtered for the application.

IPX_IMMEDIATESPXACK

optval *Type*	*Get/Set*	*Winsock Version*	*Description*
BOOL	Both	1+	If *TRUE*, do not delay sending ACKs on SPX connections.

If you set this option to *true*, acknowledgement packets will not be delayed for SPX connections. Applications that do not tend to have back-and-forth traffic over SPX should set this—it increases the number of ACKs sent but prevents the appearance of slow performance as a result of delayed acknowledgments.