# Network Programming for Microsoft Windows



PUBLISHED BY
Microsoft Press
A Division of Microsoft Corporation
One Microsoft Way
Redmond, Washington 98052-6399
http://www.mspress.microsoft.com/books/2459.htm



ISBN 0-7356-0560-2

Printed and bound in the United States of America.
Distributed in Canada by Penguin Books Canada Limited.
A CIP catalogue record for this book is available from the British Library.

Microsoft Press books are available through booksellers and distributors worldwide. For further information about international editions, contact your local Microsoft Corporation office or contact Microsoft Press International directly at fax (425) 936-7329. Visit our Web site at mspress.microsoft.com.



**Acquisitions Editor:** Ben Ryan

**Project Editor:** Rebecca McKay

**Technical Editor:** Donnie Cameron

---

*Send feedback to MSDN. Look here for MSDN Online resources.*

# Part II: The Winsock API

Part II of this book is dedicated to Winsock programming on Win32 platforms. Winsock is the preferred interface for accessing a variety of underlying network protocols and is available in varying forms on every Win32 platform. Winsock is a network programming interface and not a protocol. The Winsock interface inherits a great deal from the Berkeley (BSD) Sockets implementation on UNIX platforms, which is capable of accessing multiple networking protocols. In Win32 environments, the interface has finally evolved into a truly protocol-independent interface, especially with the release of Winsock 2.

The next three chapters describe protocol and Winsock basics, including addressing for each protocol and a simple Winsock client/server sample. The later chapters describe new features in Winsock 2, such as transport service providers, name space providers, and Quality of Service (QOS). What might be confusing about some of these technologies is that they are in the Winsock 2 specification and Winsock 2 is supported on all the current Win32 platforms (except Windows CE, which we will discuss in this section), but not all these features are implemented on the given platforms. These limitations are pointed out wherever applicable. This section assumes you have basic knowledge of Winsock (or BSD sockets) and are somewhat familiar with basic client/server Winsock terminology.

---

# Chapter 5: Network Principles and Protocols

The main motivation for creating the Winsock 2 specification was to provide a protocol-independent transport interface. This is wonderful in that it provides one familiar interface for network programming over a variety of network transports, but you should still be aware of the network protocol characteristics. This chapter covers the traits that you should be aware of when utilizing a particular protocol, including some basic networking principles. Additionally, we'll discuss how to programmatically query Winsock for protocol information, and we'll examine the basic steps necessary to create a socket for a given protocol.

## Protocol Characteristics

The first part of this chapter discusses the basic characteristics found in the world's available transport protocols. This information is meant to provide a bit of background on the type of behavior that protocols adhere to as well as to inform you, the programmer, of how a protocol will behave in an application.

### Message-Oriented

A protocol is said to be message-oriented if for each discrete write command it transmits those and only those bytes as a single message on the network. This also means that when the receiver requests data, the data returned is a discrete message written by the sender. The receiver will not get more than one message. In Figure 5-1, for example, the workstation on the left submits messages of 128, 64, and 32 bytes destined for the workstation on the right. The receiving workstation issues three read commands with a 256-byte buffer. Each call in succession returns 128, 64, and 32 bytes. The first call to read does not return all three packets, even if all the packets have been received. This logic is known as preserving message boundaries and is often desired when structured data is exchanged. A network game is a good example of preserving message boundaries. Each player sends all other players a packet with positional information. The code behind such communication is simple: one player requests a packet of data, and that player gets exactly one packet of positional information from

another player in the game.

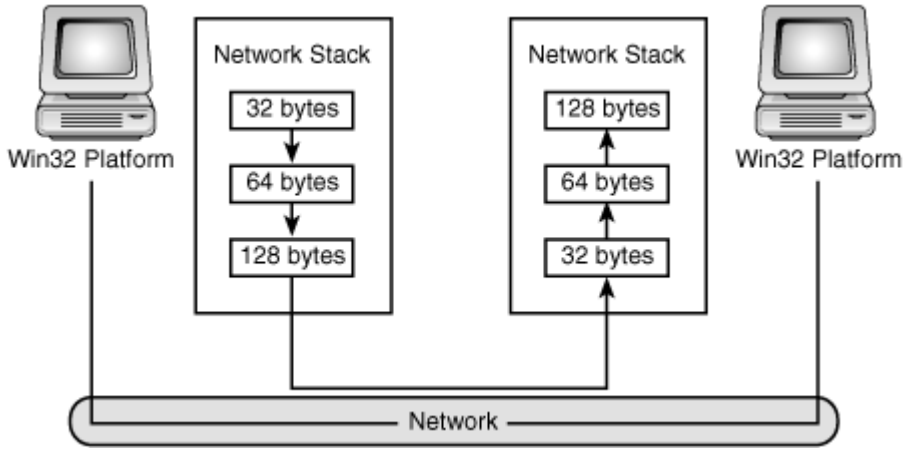


**Figure 5-1. Datagram services**

A protocol that does not preserve these message boundaries is often referred to as a stream-based protocol. Be aware that the term "stream-based" is often loosely applied to imply additional characteristics. Stream service is defined as the transmitting of data in a continual process: the receiver reads as much data as is available with no respect to message boundaries. For the sender, this means that the system is allowed to break up the original message into pieces or lump several messages together to form a bigger packet of data. On the receiving end, the network stack reads data as it arrives and buffers it for the process. When the process requests an amount of data, the system returns as much data as possible without overflowing the buffer supplied by the client call. In Figure 5-2, the sender submits packets of 128, 64, and 32 bytes; however, the local system stack is free to gather the data into larger packets. In this case, the second two packets are transmitted together. The decision to lump discrete packets of data together is based on a number of factors, such as the maximum transmit unit or the Nagle algorithm. In TCP/IP, the Nagle algorithm consists of a host waiting for data to accumulate before sending it on the wire. The host will wait until it has a large enough chunk of data to send or until a predetermined amount of time elapses. When implementing the Nagle algorithm the host's peer waits a predetermined amount of time for outgoing data before sending an acknowledgement to the host so that the peer doesn't have to send a packet with only the acknowledgement. Sending many packets of a small size is inefficient and adds substantial overhead for error checking and acknowledgments.

On the receiving end, the network stack pools together all incoming data for the given process. Take a look at Figure 5-2. If the receiver performs a read with a 256-byte buffer, all 224 bytes are returned at once. If the receiver requests that only 20 bytes be read, the system returns only 20 bytes.

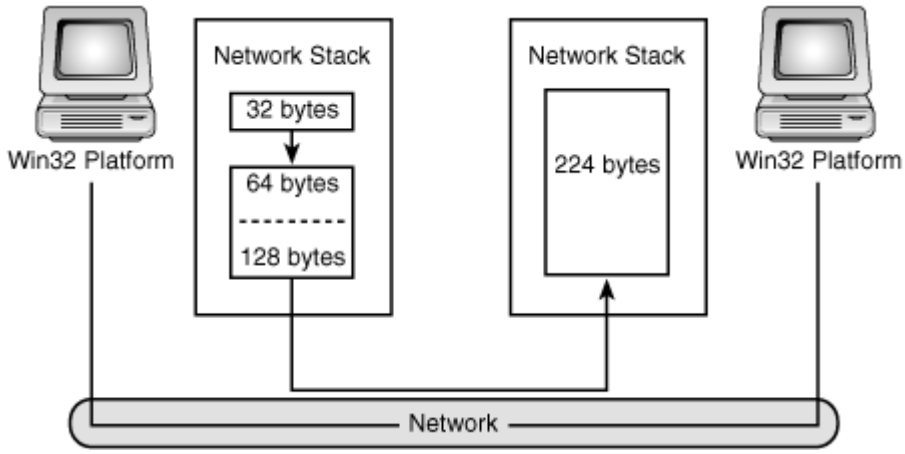


**Figure 5-2. Stream services**

### Pseudo-stream

Pseudo-stream is a term often applied to a system with a message-based protocol that sends data in discrete packets, which the receiver reads and buffers in a pool so that the receiving application reads data chunks of any size. Combining the sender in Figure 5-1 with the receiver in Figure 5-2 illustrates how pseudo-streams work. The sender must send each individual packet separately, but the receiver is free to gather them. For the most part, treat pseudo-streaming as you would a normal stream-oriented protocol.

## Connection-Oriented and Connectionless

A protocol usually provides either connection-oriented services or connectionless services. In connection-oriented services, a path is established between the two communicating parties before any data is exchanged. This ensures that there is a route between the two parties in addition to ensuring that both parties are alive and responding. This also means that establishing a communication channel between two participants requires a great deal of overhead. Additionally, most connection-oriented protocols guarantee delivery, further increasing overhead as additional computations are performed to verify correctness. On the other hand, a connectionless protocol makes no guarantees that the recipient is

listening. A connectionless service is similar to the postal service: the sender addresses a letter to a particular person and puts it in the mail. The sender doesn't know whether the recipient is expecting to receive a letter or whether severe storms are preventing the post office from delivering the message.

## Reliability and Ordering

Reliability and ordering are perhaps the most critical characteristics to be aware of when designing an application to use a particular protocol. In most cases, reliability and ordering are closely tied to whether a protocol is connectionless or connection-oriented. Reliability, or guaranteed delivery, ensures that each byte of data from the sender will reach the intended recipient unaltered. An unreliable protocol does not ensure that each byte arrives, and it makes no guarantee as to the data's integrity.

Ordering has to do with the order in which the data arrives at the recipient. A protocol that preserves ordering ensures that the recipient receives the data in the exact order it was sent. Obviously, a protocol that does not preserve order makes no such guarantees.

Reliability and ordering are closely tied to whether a protocol is connectionless or connection-oriented. In the case of connection-oriented communications, if you are already making the extra effort to establish a clear communication channel between the two participants, you usually want to guarantee data integrity and data ordering. In most cases, connection-oriented protocols do guarantee reliability. The next section discusses actual protocols and their characteristics. Note that by assuring packet ordering, you do not automatically guarantee data integrity. Of course, the great benefit of connectionless protocols is their speed: they don't bother to establish a virtual connection to the recipient. Why slow this down with error checking? This is why connectionless protocols generally don't guarantee data integrity or ordering, while connection-oriented protocols do. Why would anyone use datagrams with all these faults? Generally, connectionless protocols are an order of magnitude faster than connection-oriented communications. No checks need to be made for factors such as data integrity and acknowledgments of received data, factors that add a great deal of complexity to sending even small amounts of data. Datagrams are useful for noncritical data transfers. Datagrams are well suited for applications like the game example that we discussed earlier: each

player can use datagrams to periodically send his or her positions within the game to every other player. If one client misses a packet, it quickly receives another, giving the player an appearance of seamless communication.

## Graceful Close

A graceful close is associated only with connection-oriented protocols. In a graceful close, one side initiates the shutting down of a communication session and the other side still has the opportunity to read pending data on the wire or the network stack. A connection-oriented protocol that does not support graceful closes causes an immediate termination of the connection and loss of any data not read by the receiving end whenever either side closes the communication channel. In the case of TCP, each side of a connection has to perform a close to fully terminate the connection. The initiating side sends a segment (datagram) with a FIN control flag to the peer. Upon receipt, the peer sends an ACK control flag back to the initiating side to acknowledge receipt of the FIN, but the peer is still able to send more data. The FIN control flag signifies that no more data will be sent from the side originating the close. Once the peer decides it no longer needs to send data, it too issues a FIN, which the initiator acknowledges with an ACK control flag. At this point, the connection has been completely closed.

## Broadcast Data

To broadcast data is to be able to send data from one workstation so that all other workstations on the local area network can receive it. This feature is available to connectionless protocols because all machines on the LAN can pick up and process a broadcast message. The drawback to using broadcast messages is that every machine has to process the message. For example, let’s say the user broadcasts a message on the LAN, and the network card on each machine picks up the message and passes it up to the network stack. The stack then cycles through all network applications to see whether they should receive this message. Usually a majority of the machines on the network are not interested and simply discard the data. However, each machine still has to spend time processing the packet to see whether any applications are interested in it. Consequently, high-broadcast traffic can bog down machines on a LAN as each workstation inspects the packet. In general, routers do

not propagate broadcast packets.

## Multicast Data

Multicasting is the ability of one process to send data that will be received by one or more recipients. The method by which a process joins a multicast session differs depending on the underlying protocol. For example, under the IP protocol, multicasting is a modified form of broadcasting. IP multicasting requires that all hosts interested in sending or receiving data join a special group. When a process wishes to join a multicast group, a filter is added on the network card so that data bound to only that group address will be picked up by the network hardware and propagated up the network stack to the appropriate process. Video conferencing applications often use multicasting. Chapter 11 covers multicast programming from Winsock as well as other critical multicasting issues.

## Quality of Service

Quality of service (QOS) is an application's ability to request certain bandwidth requirements to be dedicated for exclusive use. One good use for quality of service is real-time video streaming. In order for the receiving end of a real-time video streaming application to display a smooth, clear picture, the data being sent must fall within certain restrictions. In the past, an application would buffer data and play back frames from the buffer to maintain a smooth video. If there is a period during which data is not being received fast enough, the playback routine has a certain number of buffered frames that it can play. QOS allows bandwidth to be reserved on the network, so frames can be read off the wire within a set of guaranteed constraints. Theoretically this means that the same real-time video streaming application can use QOS and eliminate the need to buffer any frames. QOS is discussed in detail in Chapter 12.

## Partial Messages

Partial messages apply only to message-oriented protocols. Let's say an application wants to receive a message but the local computer has received only part of the data. This can be a common occurrence, especially if the sending computer is transmitting large messages. The local machine might not have enough resources free to contain the whole message. In reality, most message-oriented

protocols impose a reasonable limit on the maximum size of a datagram, so this particular event is not encountered often. However, most datagram protocols support messages of a size large enough to require being broken up into a number of smaller chunks for transmission on the physical medium. Thus the possibility exists that when a user's application requests to read a message, the user's system might have received only a portion of the message. If the protocol supports partial messages, the reader is notified that the data being returned is only a part of the whole message. If the protocol does not support partial messages, the underlying network stack holds onto the pieces until the whole message arrives. If for some reason the whole message does not arrive, most unreliable protocols that lack support for partial messages will simply discard the incomplete datagram.

## Routing Considerations

One important consideration is whether a protocol is routable. If a protocol is routable, a successful communication path can be set up (either a virtual connection-oriented circuit or a data path for datagram communication) between two workstations, no matter what network hardware lies between them. For example, machine A is on a separate subnet from machine B. A router linking the two subnets separates the two machines. A routable protocol realizes that machine B is not on the same subnet as machine A; therefore, it directs the packet to the router, which decides how to best forward the data so that it reaches machine B. A nonroutable protocol is not able to make such provisions: the router drops any packets of nonroutable protocols that it receives. The router does not forward a packet from a nonroutable protocol even if the packet's intended destination is on the connected subnet. NetBEUI is the only protocol supported by Win32 platforms that is not capable of being routed.

## Other Characteristics

Each protocol supported on Win32 platforms presents characteristics that are specialized or unique. Also, a myriad of other protocol characteristics, such as byte ordering and maximum transmission size, can be used to describe every protocol available on networks today. Not all of those characteristics are necessarily critical to writing a successful Winsock application. Winsock 2 provides a

facility to enumerate each available protocol provider and query its characteristics. The third section of this chapter explains this function and presents a code sample.

## Supported Protocols

One of the most useful features offered by Win32 platforms is the capability to simultaneously support many different network protocols. As you read in Chapter 2, the Windows redirector ensures that network requests are routed to the right protocols and subsystems; however, with Winsock you can write network applications that directly utilize any one of these protocols. Chapter 6 discusses how machines on a network are addressed using the different protocols available to a workstation. One of the benefits of using the Winsock programming interface is that it is a protocol-independent interface. A majority of all operations are common no matter which protocol is used. However, you must understand how workstations are addressed in order to locate and connect to a peer for network communication.

### Supported Win32 Network Protocols

Win32 platforms support a wide variety of protocols. Each protocol is usually capable of multiple behaviors. For example, Internet Protocol (IP) is capable of both connection-oriented stream services and connectionless datagram services. Table 5-1 provides most of the various protocols available and some of the behaviors each supports.

**Table 5-1. The characteristics of available protocols**

| Protocol | Name | Message Type | Connection Type | Reliable | Packet Ordering | Graceful Close | Supports Broadcast | Supports Multipoint | Quality of Service | Maximum Message Size (Bytes) |
|---|---|---|---|---|---|---|---|---|---|---|
| IP | MSAFD TCP | Stream | Connection | Yes | Yes | Yes | No | No | No | None |
| | MSAFD UDP | Message | Connectionless | No | No | No | Yes | Yes | No | 65467 |
| | RSVP TCP | Stream | Connection | Yes | Yes | Yes | No | No | Yes | None |
| | RSVP UDP | Message | Connectionless | No | No | No | Yes | Yes | Yes | 65467 |
| IPX/SPX | MSAFD nwlnkipx [IPX] | Message | Connectionless | No | No | No | Yes | Yes | No | 576 |
| | MSAFD nwlnkspx [SPX] | Message | Connection | Yes | Yes | No | No | No | No | None |
| | MSAFD nwlnkspx [SPX] [Pseudo-stream] | Message | Connection | Yes | Yes | No | No | No | No | None |
| | MSAFD nwlnkspx [SPXII] | Message | Connection | Yes | Yes | Yes | No | No | No | None |
| | MSAFD nwlnkspx [SPXII] [Pseudo-stream] | Message | Connection | Yes | Yes | Yes | No | No | No | None |
| NetBIOS | Sequential Packets | Message | Connection | Yes | Yes | No | No | No | No | 64 KB (65535) |
| | Datagrams | Message | Connectionless | No | No | No | Yes* [SP25] | No | No | 64 KB (65535) |
| AppleTalk | MSAFD AppleTalk [ADSP] | Message | Connection | Yes | Yes | Yes | No | No | No | 64 KB (65535) |
| | MSAFD AppleTalk [ADSP] [Pseudo-stream] | Message | Connection | Yes | Yes | Yes | No | No | No | None |
| | MSAFD AppleTalk [PAP] | Message | Connection | Yes | Yes | Yes | No | No | No | 4096 |
| | MSAFD AppleTalk [RTMP] | Stream | Connectionless | No | No | No | No | No | No | None |
| | MSAFD AppleTalk [ZIP] | Stream | Connectionless | No | No | No | No | No | No | None |
| ATM | MSAFD ATM AAL5 | Stream | Connection | No | Yes | No | No | Yes | Yes | None |
| | Native ATM (AAL5) | Message | Connection | No | Yes | No | No | Yes | Yes | None |
| Infrared Sockets | MSAFD Irda [IrDA] | Stream | Connection | Yes | Yes | Yes | No | No | No | None |

*NetBIOS supports datagrams sent to either unique or group NetBIOS clients. It does not support blanket broadcasts.

## Windows CE Network Protocols

Windows CE differs from the other Win32 platforms in that it supports only TCP/IP as a network transport protocol. Additionally, Windows CE supports only the Winsock 1.1 specification, which means that most of the new Winsock 2 features covered in this section don’t apply to this platform. Windows CE does support NetBIOS over TCP/IP through a redirector, but it does not offer any kind of programming interface to NetBIOS through the native NetBIOS API or through Winsock.

# Winsock 2 Protocol Information

Winsock 2 provides a method for determining which protocols are installed on a given workstation and returning a variety of characteristics for each protocol. If a protocol is capable of multiple behaviors, each distinct behavior type has its own catalog entry within the system. For example, if you install TCP/IP on your system, there will be two IP entries: one for TCP, which is reliable and connection-oriented, and one for UDP, which is unreliable and connectionless.

The function call to obtain information on installed network protocols is *WSAEnumProtocols* and is defined as

```
int WSAEnumProtocols (
    LPINT lpiProtocols,
    LPWSAPROTOCOL_INFO lpProtocolBuffer,
    LPDWORD lpdwBufferLength
);
```

This function supersedes the Winsock 1.1 function *EnumProtocols*, which is the necessary function for Windows CE. The only difference is that *WSAEnumProtocols* returns an array of *WSAPROTOCOL_INFO* structures, whereas *EnumProtocols* returns an array of *PROTOCOL_INFO* structures that contain fewer fields than the *WSAPROTOCOL_INFO* structure (but more or less the same information). The *WSAPROTOCOL_INFO* structure is defined as

```
typedef struct _WSAPROTOCOL_INFOW {
    DWORD             dwServiceFlags1;
    DWORD             dwServiceFlags2;
    DWORD             dwServiceFlags3;
    DWORD             dwServiceFlags4;
    DWORD             dwProviderFlags;
    GUID              ProviderId;
    DWORD             dwCatalogEntryId;
    WSAPROTOCOLCHAIN  ProtocolChain;
    int               iVersion;
    int               iAddressFamily;
    int               iMaxSockAddr;
    int               iMinSockAddr;
    int               iSocketType;
    int               iProtocol;
```

```
    int                 iProtocolMaxOffset;
    int                 iNetworkByteOrder;
    int                 iSecurityScheme;
    DWORD               dwMessageSize;
    DWORD               dwProviderReserved;
    WCHAR               szProtocol[WSAPROTOCOL_LEN + 1];
} WSAPROTOCOL_INFOW, FAR * LPWSAPROTOCOL_INFOW;
```

**Initializing Winsock** Before you can call a Winsock function, you must load the correct version of the Winsock library. The Winsock initialization routine is WSAStartup, defined as

```
int WSAStartup(WORD wVersionRequested, LPWSADATA
lpWSAData);
```

The first parameter is the version of the Winsock library that you want to load. For current Win32 platforms, the latest Winsock 2 library is version 2.2. The only exception is Windows CE, which supports only Winsock version 1.1. If you wanted Winsock version 2.2, you could either specify the value (0x0202) or use the macro *MAKEWORD(2, 2)*. The high-order byte specifies the minor version number, while the low-order byte specifies the major version number.

The second parameter is a structure, *WSADATA*, that is returned upon completion. *WSADATA* contains information about the version of Winsock that *WSAStartup* loaded. Table 5-2 lists the individual fields of the *WSADATA* structure, which is actually defined as

```
typedef struct WSAData {
    WORD            wVersion;
    WORD            wHighVersion;
    char            szDescription[WSADESCRIPTION_LEN +
1];
    char            szSystemStatus[WSASYS_STATUS_LEN +
1];
    unsigned short  iMaxSockets;
    unsigned short  iMaxUdpDg;
    char FAR *      lpVendorInfo;
} WSADATA, FAR * LPWSADATA;
```

For the most part, the only useful information returned in the *WSADATA* structure is *wVersion* and *wHighVersion*. The entries pertaining to maximum sockets and maximum UDP size should

be obtained from the catalog entry for the specific protocol you are using. The above section on *WSAEnumProtocols* discusses this.

The easiest way to call *WSAEnumProtocols* is to make the first call with *lpProtocolBuffer* equal to *NULL* and *lpdwBufferLength* set to 0. The call fails with *WSAENOBUFS*, but *lpdwBufferLength* then contains the correct size of the buffer required to return all the protocol information. Once you allocate the correct buffer size and make another call with the supplied buffer, the function returns the number of *WSAPROTOCOL_INFO* structures returned. At this point, you can step through the structures to find the protocol entry with your required attributes. The sample program Enum.c on the companion CD-ROM enumerates all installed protocols and prints out the characteristics of each protocol.

**Table 5-2. Member fields of the WSADATA structure**

| Field | Description |
|---|---|
| *wVersion* | The Winsock version the caller is expected to use |
| *wHighVersion* | The highest Winsock version supported by the loaded library, usually the same value as *wVersion* |
| *szDescription* | A text description of the loaded library |
| *szSystemStatus* | A text string containing relevant status or configuration information |
| *iMaxSockets* | Maximum number of sockets (ignore this field for Winsock 2 or later) |
| *iMaxUdpDg* | Maximum UDP datagram size (ignore this field for Winsock 2 or later) |
| *lpVendorInfo* | Vendor-specific information (ignore this field for Winsock 2 or later) |

When you are finished with the Winsock library and no longer want to call any Winsock functions, the companion routine *WSACleanup* unloads the library and frees any resources. This function is defined as

```
int WSACleanup (void);
```

Keep in mind that for each call to *WSAStartup* a matching call to *WSACleanup* will be needed because each startup call increments the reference count to the loaded Winsock DLL, requiring an equal number of calls to *WSACleanup* to decrement the count.

Note that Winsock 2 is fully compatible with all Winsock 1.1 function calls. Thus an application written to the Winsock 1.1 specification will be able to run if it loads the Winsock 2 library, as the Winsock 1.1 functions are mapped through their Winsock 2 equivalents.

The most commonly used field of the *WSAPROTOCOL_INFO* structure is *dwServiceFlags1*, which is a bit field for the various protocol attributes. Table 5-3 lists the various bit flags that can be set in the field and describes the meaning of each property. To check for the presence of a particular property, select the appropriate property flag and perform a bitwise AND of the property and the *dwServiceFlags1* field. If the resultant value is nonzero, that property is present in the given protocol; otherwise, it isn't.

**Table 5-3. Protocol flags**

| Property | Meaning |
|---|---|
| *XP1_CONNECTIONLESS* | This protocol provides connectionless service. If not set, the protocol supports connection-oriented data transfers. |
| *XP1_GUARANTEED_DELIVERY* | This protocol guarantees that all data sent will reach the intended recipient. |
| *XP1_GUARANTEED_ORDER* | This protocol guarantees that the data will arrive in the order in which it was sent and that it will |

| | |
|---|---|
| | not be duplicated. However, this does not guarantee delivery. |
| *XP1_MESSAGE_ORIENTED* | This protocol honors message boundaries. |
| *XP1_PSEUDO_STREAM* | This protocol is message-oriented, but the message boundaries are ignored on the receiver side. |
| *XP1_GRACEFUL_CLOSE* | This protocol supports two-phase closes: each party is notified of the other's intent to close the communication channel. If not set, only abortive closes are performed. |
| *XP1_EXPEDITED_DATA* | This protocol supports urgent data (out-of-band data). |
| *XP1_CONNECT_DATA* | This protocol supports transferring data with the connection request. |
| *XP1_DISCONNECT_DATA* | This protocol supports transferring data with the disconnect request. |
| *XP1_SUPPORT_BROADCAST* | This protocol supports the broadcast mechanism. |
| *XP1_SUPPORT_MULTIPOINT* | This protocol supports multipoint or |

| | |
|---|---|
| | multicast mechanisms. |
| *XP1_MULTIPOINT_CONTROL_PLANE* | If this flag is set, the control plane is rooted. Otherwise, it is nonrooted. |
| *XP1_MULTIPOINT_DATA_PLANE* | If this flag is set, the data plane is rooted. Otherwise, it is nonrooted. |
| *XP1_QOS_SUPPORTED* | This protocol supports QOS requests. |
| *XP1_UNI_SEND* | This protocol is unidirectional in the send direction. |
| *XP1_UNI_RECV* | This protocol is unidirectional in the receive direction. |
| *XP1_IFS_HANDLES* | The socket descriptors returned by the provider are Installable File System (IFS) handles and can be used in API functions such as *ReadFile* and *WriteFile*. |
| *XP1_PARTIAL_MESSAGE* | The *MSG_PARTIAL* flag is supported in *WSASend* and *WSASendTo*. |

Most of these flags will be discussed in one or more of the following chapters, so we won't go into detail about the full meaning of each flag now. The other fields of importance are *iProtocol*, *iSocketType*, and *iAddressFamily*. The *iProtocol* field defines which protocol this entry belongs to. The *iSocketType* field is important if the protocol is capable of multiple behaviors, such as stream-oriented connections

or datagram connections. Finally, *iAddressFamily* is used to distinguish the correct addressing structure to use for the given protocol. These three entries are of great importance when creating a socket for a given protocol and will be discussed in detail in the next section.

## Windows Sockets

Now that you are familiar with the various protocols available and their attributes, we'll take a look at using these protocols from Winsock. If you're familiar with Winsock, you know that the API is based on the concept of a socket. A socket is a handle to a transport provider. In Win32, a socket is not the same thing as a file descriptor and therefore is a separate type, *SOCKET*. Two functions create a socket:

```
SOCKET WSASocket (
    int af,
    int type,
    int protocol,
    LPWSAPROTOCOL_INFO lpProtocolInfo,
    GROUP g,
    DWORD dwFlags
);

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

The first parameter, *af*, is the address family of the protocol. For example, if you want to create either a UDP or a TCP socket, use the constant *AF_INET* to indicate the Internet Protocol (IP). The second parameter, *type*, is the socket type of the protocol. A socket type can be one of five values: *SOCK_STREAM*, *SOCK_DGRAM*, *SOCK_SEQPACKET*, *SOCK_RAW*, and *SOCK_RDM*. The third parameter is *protocol*. This field is used to qualify a specific transport if there are multiple entries for the given address family and socket type. Table 5-4 shows the values used for the address family, socket type, and protocol fields for a given network transport.

**Table 5-4. Socket parameters**

| Protocol | Address Family | Socket Type | | Pro |
|---|---|---|---|---|
| Internet Protocol (IP) | *AF_INET* | TCP | *SOCK_STREAM* | *IPP* |
| | | UDP | *SOCK_DGRAM* | *IPP* |
| | | Raw sockets | *SOCK_RAW* | *IPP* |
| | | | | *IPP* |
| IPX/SPX | *AF_NS* | MSAFD nwlnkipx [IPX] | *SOCK_DGRAM* | *NSP* |
| | *AF_IPX* | MSAFD nwlnkspx [SPX] | *SOCK_SEQPACKET* | *NSP* |
| | | MSAFD nwlnkspx [SPX] [Pseudo-stream] | *SOCK_STREAM* | *NSP* |
| | | MSAFD nwlnkspx [SPXII] | *SOCK_SEQPACKET* | *NSP* |
| | | MSAFD nwlnkspx [SPXII] [Pseudo-stream] | *SOCK_STREAM* | *NSP* |
| NetBIOS | *AF_NETBIOS* | Sequential Packets | *SOCK_SEQPACKET* | LAN |
| | | Datagrams | *SOCK_DGRAM* | LAN |
| AppleTalk | *AF_APPLETALK* | MSAFD AppleTalk [ADSP] | *SOCK_RDM* | *ATP* |

| | | | | |
|---|---|---|---|---|
| | | MSAFD AppleTalk [ADSP] [Pseudo-stream] | *SOCK_STREAM* | *ATP* |
| | | MSAFD AppleTalk [PAP] | *SOCK_RDM* | *ATP* |
| | | MSAFD AppleTalk [RTMP] | *SOCK_DGRAM* | *DDP* |
| | | MSAFD AppleTalk [ZIP] | *SOCK_DGRAM* | *DDP* |
| ATM | *AF_ATM* | MSAFD ATM AAL5 | *SOCK_RAW* | *ATM* |
| | | Native ATM (AAL5) | *SOCK_RAW* | *ATM* |
| Infrared Sockets | *AF_IRDA* | MSAFD Irda [IrDA] | *SOCK_STREAM* | *IRD* |

The first three parameters for creating a socket are organized in three tiers. The first and most important parameter is the address family. This specifies which protocol is being used. It also dictates the valid options for the second and third parameters. For example, choosing the ATM address family (*AF_ATM*) limits you to only raw sockets (*SOCK_RAW*) for the socket type. Likewise, by selecting an address family and a socket type, you are limited as to the protocol you choose. However, it is possible to pass a 0 for the *protocol* parameter. The system then chooses a transport provider based on the other two parameters, *af* and *type*. When enumerating the catalog entries for protocols, check the *dwProviderFlags* entry of the *WSAPROTOCOL_INFO* structure. If this field is set to *PFL_MATCHES_PROTOCOL_ZERO*, this is the default transport that will be used if the protocol parameter to *socket* or *WSASocket* is 0.

If you are using the *WSASocket* function and have enumerated all protocols using *WSAEnumProtocols*, you can select a

*WSAPROTOCOL_INFO* structure and pass that to the *WSASocket* as the *lpProtocolInfo* parameter. If you then specify the constant *FROM_PROTOCOL_INFO* in all of the first three parameters (*af*, *type*, and *protocol*), the values from the *WSAPROTOCOL_INFO* structure you supplied are used instead. This is how you specify an exact protocol entry.

The last two flags of *WSASocket* are simple. The group parameter is always 0 because no version of Winsock supports socket groups. The *dwFlags* parameter is used to specify one or more of the following flags:

- *WSA_FLAG_OVERLAPPED*
- *WSA_FLAG_MULTIPOINT_C_ROOT*
- *WSA_FLAG_MULTIPOINT_C_LEAF*
- *WSA_FLAG_MULTIPOINT_D_ROOT*
- *WSA_FLAG_MULTIPOINT_D_LEAF*

The first flag, *WSA_FLAG_OVERLAPPED*, is used to specify that this socket is capable of overlapped I/O, which is one of the possible communication models available in Winsock. This topic is covered in detail in Chapter 8. If you create a socket using the *socket* call, *WSA_FLAG_OVERLAPPED* is set by default. In general, it is a good idea to always set this flag when using *WSASocket*. The last four flags deal with multicast sockets.

## Raw Sockets

When creating a socket with *WSASocket*, you can pass a *WSAPROTOCOL_INFO* structure into the call to define the kind of socket you want to create; however, you can create socket types that don’t have an entry in the transport provider catalog. The best example of this is raw sockets under IP. Raw sockets are a form of communication that allows you to encapsulate other protocols within the UDP packet, such as the Internet Control Message Protocol (ICMP). ICMP’s purpose is to deliver control, error, and informational messages among Internet hosts. Because ICMP does not provide any data transfer facilities, it is not considered to be at the same level as UDP or TCP, but at the same level as IP itself. Chapter 13

covers raw sockets in further detail.

## Platform-Specific Information

Windows 95 out of the box supports the Winsock 1.1 specification. Microsoft has made freely available a Winsock 2 update that can be downloaded from its Web site (*http://www.microsoft.com/windows95/downloads/).* Also, a Winsock 2 SDK is available that includes the necessary headers and libraries to compile a Winsock 2 application. Windows 98, Windows NT 4, and Windows 2000 all support Winsock 2 natively without any necessary add-ons. Windows CE supports only the Winsock 1.1 specification.

As far as support for various transport providers goes, a few limitations should be mentioned. Windows CE supports only TCP/IP and Infrared Sockets. For Windows 95 and Windows 98, the NetBIOS transport providers (transports with the address family *AF_NETBIOS*) are not exposed to the Winsock API. If you perform a *WSAEnumProtocols*, none of the NetBIOS providers will be listed, even though they are installed on the machine. However, NetBIOS is still available by using the native NetBIOS interface, as described in Chapter 1. Last, the RSVP (which offers QOS) and ATM providers listed are natively available on Windows 98 and Windows 2000.

## The Winsock API and the OSI Model

Let's look at how some of the concepts presented in this chapter relate to the OSI model (see Figure 1-1 on page 4). The transport providers in the Winsock catalog that are enumerated by *WSAEnumProtocols* are at the Transport layer of the OSI model. That is, each of these transports provides a method of transferring data; however, each is a member of a protocol, and a network protocol is at the Network layer because it is the protocol that provides a method of addressing each node on a network. For example, UDP and TCP are transports, but both belong to the Internet Protocol.

The Winsock API fits between the Session and Transport layers. Winsock provides the ability to open, manipulate, and close data sessions for a given transport. Under Windows, the top three layers (Application, Presentation, and Session) relate for the most part to

your Winsock application. In other words, your Winsock application controls all aspects of the session and if necessary formats the data for the purpose of your program.

## Selecting the Right Protocol

When you develop a network application, you can choose from a number of available protocols to base your application on. If your application needs to communicate over a certain protocol, you don't have many choices; however, if you are developing an application from scratch, TCP/IP is the way to go, at least from the point of supportability and commitment from Microsoft. AppleTalk, NetBIOS, and IPX/SPX are protocols included by Microsoft to provide compatibility with other operating systems and were at one time the staples for network programming. This was evident when Windows 95 was installed on network computers: the default protocols installed were NetBEUI and IPX/SPX.

With the explosive growth in the Internet, the great majority of companies, educational institutions, and others have made TCP/IP their protocol of choice. In Windows 2000, Microsoft also stresses TCP/IP. Now TCP/IP is the default protocol installed, and most networking services will be based on this protocol, lessening the reliance on NetBIOS. Additionally, when you find a bug in Microsoft's implementation of TCP/IP there is a quick response to post hot fixes, whereas for bugs in other protocols there might or might not be a fix, depending on demand.

This said, TCP/IP is the safe way to go when choosing a protocol to use in a networked application. Additionally, Microsoft is showing strong support for ATM networks. If you have the luxury of developing an application that will run exclusively on ATM networks, it should be relatively safe to develop it using native ATM from Winsock. Of course, users of TCP/IP should note that ATM networks can be configured to run in TCP/IP emulation mode, which runs exceptionally well. These factors are just a few among the many to consider, in addition to the protocol characteristics needed by any application you develop.

## Conclusion

In this chapter, we covered the basic characteristics to be aware of

when choosing a network transport for an application. Knowledge of these characteristics is vital when it comes to successfully developing a network application based on a particular protocol. We also looked into programmatically obtaining a list of transport providers installed on a system and how to query for a particular property. Finally, we learned how to create a socket for a given network transport by specifying the correct parameters to either the *WSASocket* function or the *socket* function, and also by querying for the catalog entry using *WSAEnumProtocols* and passing in a *WSAPROTOCOL_INFO* structure to *WSASocket*. In the next chapter, we will investigate the addressing methods for each of the major protocols.

---

# Chapter 6: Address Families and Name Resolution

In order to establish communication through Winsock, you must understand how to address a workstation using a particular protocol. This chapter explains the protocols supported by Winsock and how each protocol resolves an address specific to that family to an actual machine on the network. Winsock 2 introduces several new protocol-independent functions that can be used with sockets of any address families; in most cases, however, each address family provides its own mechanisms for resolving addresses, either through a function or as an option passed to *getsockopt*. This chapter covers only the basic knowledge necessary to form an address structure for each protocol family. Chapter 10 covers the registration and name resolution functions, which advertise a service of a given protocol family. (This is a bit different from just resolving a name.) See Chapter 10 for more details on the differences between straight name resolution and service advertising and resolution.

For each covered address family, we will discuss the basics of how to address a machine on a network. We will then create a socket for each family. Additionally, we will cover the protocol-specific options for name resolution.

## IP

The Internet Protocol (IP) is commonly known as the network protocol that is used on the Internet. IP is widely available on most computer operating systems and can be used on most local area networks (LANs), such as a small network in your office, and on wide area networks (WANs), such as the Internet. By design, IP is a connectionless protocol and doesn’t guarantee delivery of data. Two higher-level protocols—TCP and UDP—are used for data communication over IP.

## TCP

Connection-oriented communication is accomplished through the Transmission Control Protocol (TCP). TCP provides reliable error-free data transmission between two computers. When applications communicate using TCP, a virtual connection is established between

the source computer and the destination computer. Once a connection is established, data can be exchanged between the computers as a two-way stream of bytes.

## UDP

Connectionless communication is accomplished through the User Datagram Protocol (UDP). UDP doesn't guarantee reliable data transmission and is capable of sending data to multiple destinations and receiving data from multiple sources. For example, if a client sends data to a server, the data is transmitted immediately, whether or not the server is ready to receive the data. If the server receives data from the client, it doesn't acknowledge the receipt. Data is transmitted using datagrams.

Both TCP and UDP use IP for data transmission and are normally referred to as TCP/IP and UDP/IP. Winsock addresses IP communication through the *AF_INET* address family, which is defined in Winsock.h and Winsock2.h.

## Addressing

In IP, computers are assigned an IP address that is represented as a 32-bit quantity, formally known as an IP version 4 (IPv4) address. When a client wants to communicate with a server through TCP or UDP, it must specify the server's IP address along with a service port number. Also, when servers want to listen for incoming client requests, they must specify an IP address and a port number. In Winsock, applications specify IP addresses and service port information through the *SOCKADDR_IN* structure, which is defined as

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

The *sin_family* field must be set to *AF_INET*, which tells Winsock we are using the IP address family.

**IP Version 6** IP version 6 is an updated specification of IP that allows for a larger address space, which will become necessary in the near future, as IP version 4 addresses become scarce. Many of the Winsock header files contain conditional definitions for IPv6 structures; however, no current Win32 platform provides an IPv6 network stack (including Windows 2000). Microsoft Research has made available an experimental IPv6 stack that you can download and use (*http://research.microsoft.com/msripv6/*); however, it isn't supported, and we do not address any version 6–specific issues in this text.

The *sin_port* field defines which TCP or UDP communication port will be used to identify a server service. Applications should be particularly careful in choosing a port because some of the available port numbers are reserved for well-known services such as File Transfer Protocol (FTP) and Hypertext Transfer Protocol (HTTP). The ports used by well-known services are controlled and assigned by the Internet Assigned Numbers Authority (IANA) and are described in RFC 1700. Essentially, the port numbers are divided into the three ranges explained below: well-known, registered, and dynamic and/or private ports.

- 0–1023 are controlled by the IANA and are reserved for well-known services.
- 1024–49151 are registered ports listed by the IANA and can be used by ordinary user processes or programs executed by ordinary users.
- 49152–65535 are dynamic and/or private ports.

Ordinary user applications should choose the registered ports in the range 1024–49151 to avoid the possibility of using a port already in use by another application or a system service. Ports in the range 49152–65535 can also be used freely because no services are registered on these ports with the IANA. If, when using the *bind* API function, your application binds to a port that is already in use by another application on your host, the system will return the Winsock error *WSAEADDRINUSE*. Chapter 7 describes the Winsock bind process in greater detail.

The *sin_addr* field of the *SOCKADDR_IN* structure is used for storing

an IP address as a 4-byte quantity, which is an unsigned long integer data type. Depending on how this field is used, it can represent a local or remote IP address. IP addresses are normally specified in Internet standard dotted notation as "a.b.c.d." Each letter represents a number for each byte and is assigned, from left to right, to the four bytes of the unsigned long integer. The final field, *sin_zero*, functions only as padding to make the *SOCKADDR_IN* structure the same size as the *SOCKADDR* structure.

A useful support function named *inet_addr* converts a dotted IP address to a 32-bit unsigned long integer quantity. The *inet_addr* function is defined as

```
unsigned long inet_addr(
    const char FAR *cp
);
```

The *cp* field is a null-terminated character string that accepts an IP address in dotted notation. Note that this function returns an IP address as a 32-bit unsigned long integer in network-byte order. (Network-byte order is described shortly, under "Byte ordering.")

## Special addresses

Two special IP addresses affect the behavior of a socket in certain situations. The special address *INADDR_ANY* allows a server application to listen for client activity over every network interface on a host computer. Typically, server applications use this address when they bind a socket to the local interface to listen for connections. If you have a multihomed system, this address allows a single application to accept responses from multiple interfaces.

The special address *INADDR_BROADCAST* can be used to send broadcast UDP datagrams over an IP network. Using this special address requires an application to set the socket option *SO_BROADCAST*. Chapter 9 explains this option in greater detail.

## Byte ordering

Different computer processors represent numbers in *big-endian* and *little-endian* form, depending on how they are designed. For example, on Intel x86 processors, multibyte numbers are represented in little-endian form: the bytes are ordered from least significant byte to most significant byte. When an IP address and

port number are specified as multibyte quantities in a computer, they are represented in *host-byte* order. However, when IP addresses and port numbers are specified over a network, Internet networking standards specify that multibyte values must be represented in big-endian form (most significant byte to least significant byte), normally referred to as *network-byte* order.

A series of functions can be used to convert a multibyte number from host-byte order to network-byte order and vice versa. The following four API functions convert a number from host-byte to network-byte order:

```
u_long htonl(u_long hostlong);

int WSAHtonl(
    SOCKET s,
    u_long hostlong,
     u_long FAR * lpnetlong
);

u_short htons(u_short hostshort);

int WSAHtons(
    SOCKET s,
    u_short hostshort,
    u_short FAR * lpnetshort
);
```

The *hostlong* parameter of *htonl* and *WSAHtonl* is a 4-byte number in host-byte order. The *htonl* function returns the number in network-byte order, whereas the *WSAHtonl* function returns the number in network-byte order through the *lpnetlong* parameter. The *hostshort* parameter of *htons* and *WSAHtons* is a 2-byte number in host-byte order. The *htons* function returns the number as a 2-byte value in network-byte order, whereas the *WSAHtons* function returns the number through the *lpnetshort* parameter.

The next four functions are the opposite of the preceding four functions: they convert from network-byte order to host-byte order.

```
u_long ntohl(u_long netlong);

int WSANtohl(
    SOCKET s,
    u_long netlong,
```

```
    u_long FAR * lphostlong
);

u_short ntohs(u_short netshort);

int WSANtohs(
    SOCKET s,
    u_short netshort,
    u_short FAR * lphostshort
);
```

We will now demonstrate how to create a *SOCKADDR_IN* structure using the *inet_addr* and *htons* functions described above.

```
SOCKADDR_IN InternetAddr;
INT nPortId = 5150;

InternetAddr.sin_family = AF_INET;

// Convert the proposed dotted Internet address 136.149.3.
// to a 4-byte integer, and assign it to sin_addr

    (continued) InternetAddr.sin_addr.s_addr = inet_addr("

// The nPortId variable is stored in host-byte order. Conv
// nPortId to network-byte order, and assign it to sin_por

InternetAddr.sin_port = htons(nPortId);
```

Now that you have the basics of addressing IP through a *SOCKADDR_IN* structure, you can prepare to set up communication for TCP or UDP by creating a socket.

## Creating a Socket

Creating an IP socket offers applications the capability to communicate over the TCP, UDP, and IP protocols. To open an IP socket using the TCP protocol, call the *socket* function or the *WSASocket* function with the address family *AF_INET* and the socket type *SOCK_STREAM*, and set the protocol field to 0, as follows:

```
s = socket(AF_INET, SOCK_STREAM, 0);

s = WSASocket(AF_INET, SOCK_STREAM, 0, NULL, 0,
    WSA_FLAG_OVERLAPPED);
```

To open an IP socket using the UDP protocol, simply specify the socket type *SOCK_DGRAM* instead of *SOCK_STREAM* in the *socket* and *WSASocket* calls above. It is also possible to open a socket to communicate directly over IP. This is accomplished by setting the socket type to *SOCK_RAW*. Chapter 13 describes the *SOCK_RAW* option in greater detail.

## Name Resolution

When a Winsock application wants to communicate with a host over IP, it must know the host's IP address. From an application user's point of view, IP addresses aren't easy to remember. Most people would much rather refer to a machine by using an easy-to-remember, user-friendly host name instead of an IP address. Winsock provides two support functions that can help you resolve a host name to an IP address.

The Windows Sockets *gethostbyname* and *WSAAsyncGetHostByName* API functions retrieve host information corresponding to a host name from a host database. Both functions return a *HOSTENT* structure that is defined as

```
struct hostent
{
    char FAR *       h_name;
    char FAR * FAR * h_aliases;
    short            h_addrtype;
    short            h_length;
    char FAR * FAR * h_addr_list;
};
```

The *h_name* field is the official name of the host. If your network uses the Domain Name System (DNS), it is the Fully Qualified Domain Name (FQDN) that causes the name server to return a reply. If your network uses a local "hosts" file, it is the first entry after the IP address. The *h_aliases* field is a null-terminated array of alternative names for the host. The *h_addrtype* represents the address family being returned. The *h_length* field defines the length in bytes of each address in the *h_addr_list* field. The *h_addr_list* field is a null-terminated array of IP addresses for the host. (A host can have more than one IP address assigned to it.) Each address in the array is returned in network-byte order. Normally, applications use the first address in the array. However, if more than one address is returned, applications should randomly choose an

available address rather than always use the first address.

The *gethostbyname* API function is defined as

```
struct hostent FAR * gethostbyname (
    const char FAR * name
);
```

The *name* parameter represents a friendly name of the host you are looking for. If this function succeeds, a pointer to a *HOSTENT* structure is returned. Note that the memory where the *HOSTENT* structure is stored is system memory. An application shouldn't rely on this to remain static. Since this memory is maintained by the system, your application doesn't have to free the returned structure.

The *WSAAsyncGetHostByName* API function is an asynchronous version of the *gethostbyname* function that uses Windows messages to inform an application when this function completes. *WSAAsyncGetHostByName* is defined as

```
HANDLE WSAAsyncGetHostByName(
    HWND hWnd,
    unsigned int wMsg,
    const char FAR * name,
    char FAR * buf,
    int buflen
);
```

The *hWnd* parameter is the handle of the window that will receive a message when the asynchronous request completes. The *wMsg* parameter is the Windows message to be received when the asynchronous request completes. The *name* parameter represents a user-friendly name of the host we are looking for. The *buf* parameter is a pointer to the data area to receive the *HOSTENT* data. This buffer must be larger than a *HOSTENT* structure and should be set to the size defined in *MAXGETHOSTSTRUCT*.

Two more functions that retrieve host information are worth mentioning: the *gethostbyaddr* and *WSAAsyncGetHostByAddr* API functions, which are designed to retrieve host information corresponding to an IP network address. These functions are useful when you have the IP address of a host and want to look up its user-friendly name. The *gethostbyaddr* function is defined as

```
struct HOSTENT FAR * gethostbyaddr(
```

```
    const char FAR * addr,
    int len,
    int type
);
```

The *addr* parameter is a pointer to an IP address in network-byte order. The *len* parameter specifies the byte length of the *addr* parameter. The *type* parameter should specify the value *AF_INET,* which indicates that this is an IP address type. The *WSAAsyncGetHostByAddr* API function is an asynchronous version of *gethostbyaddr*.

## Port numbers

In addition to the IP address of a remote computer, an application must know the service's port number to communicate with a service running on a local or remote computer. When using TCP and UDP, applications must decide which ports they plan to communicate over. There are well-known port numbers reserved by server services that support protocols of a level higher than TCP. For example, port 21 is reserved for FTP, and port 80 is reserved for HTTP. As mentioned earlier, well-known services typically use ports 1–1023 to set up communication. If you are developing a TCP application that doesn't use one of the well-known services, consider using ports above 1023 to avoid using a port already being used. You can retrieve port numbers for well-known services by calling the *getservbyname* and *WSAAsyncGetServByName* functions. These functions simply retrieve static information from a file named *services*. In Windows 95 and Windows 98, the services file is located under %WINDOWS%; and in Windows NT and Windows 2000, it is located under %WINDOWS%\System32\Drivers\Etc. The *getservbyname* function is defined as follows:

```
struct servent FAR * getservbyname(
    const char FAR * name,
    const char FAR * proto
);
```

The *name* parameter represents the name of the service you are looking for. For example, if you are trying to locate the port for FTP, you should set the *name* parameter to point to the string "ftp". The *proto* parameter optionally points to a string that indicates the protocol that the service in *name* is registered under. The *WSAAsyncGetServByName* function is an asynchronous version of

*getservbyname*.

Windows 2000 has a new dynamic method to register and query service information for TCP and UDP. Server applications can register the service name, IP address, and port number of a service by using the *WSASetService* function. Client applications can query this service information by using a combination of the following API functions: *WSALookupServiceBegin*, *WSALookupServiceNext*, and *WSALookupServiceEnd*. Chapter 10 covers the details of using these capabilities.

# Infrared Sockets

Infrared sockets, or IrSock, are an exciting new technology first introduced on the Windows CE platform. Infrared sockets allow two PCs to communicate with each other through an infrared serial port. Infrared sockets are now available on Windows 98 and Windows 2000. Infrared sockets differ from traditional sockets in that infrared sockets are designed to take into account the transient nature of portable computing. Infrared sockets present a new name resolution model that will be discussed in the next section.

## Addressing

Because most computers with Infrared Data Association (IrDA) devices are likely to move around, traditional name-resolution schemes don't work well. Conventional resolution methods assume static resources such as name servers, which cannot be used when a person is moving a handheld PC or laptop computer running a network client.  To circumvent this problem, IrSock is designed to browse in-range resources in an ad hoc manner without the overhead of a large network, and it doesn't use standard Winsock name service functions or even IP addressing. Instead, the name service has been incorporated into the communication stream, and a new address family has been introduced to support services bound to infrared serial ports. The IrSock address structure includes a service name that describes the application used in bind and connect calls, and a device identifier that describes the device on which the service runs. This pair is analogous to the IP address and port number tuple used by conventional TCP/IP sockets. The IrSock address structure is defined as

```
typedef struct sockaddr_irda {
    u_short    irdaAddressFamily;
    u_char     irdaDeviceID[4];
    char       irdaServiceName[25];
} SOCKADDR_IRDA;
```

The *irdaAddressFamily* field is always set to *AF_IRDA*. The *irdaDeviceID* is a four-character string that uniquely identifies the device on which a particular service is running. This field is ignored when an IrSock server is created. However, the field is significant for a client because it specifies the IrDA device to connect to. (There can be multiple devices in range.) Finally, the *irdaServiceName* field is the name of the service that the application either will register itself with or is trying to connect to.

## Name Resolution

Addressing can be based on IrDA Logical Service Access Point Selectors (LSAP-SELs) or on services registered with the Information Access Services (IAS). The IAS abstracts a service from an LSAP-SEL into a user-friendly text service name, in much the same way that an Internet domain name server maps names to numeric IP addresses. You can use either an LSAP-SEL or a user-friendly name to successfully connect, but user-friendly names require name resolution. For the most part, you shouldn't use the direct LSAP-SEL "address" because the address space for IrDA services is limited. The Win32 implementation allows LSAP-SEL integer identifiers in the range of 1 to 127. Essentially an IAS server can be thought of as a WINS server because it associates an LSAP-SEL with a textual service name.

An actual IAS entry has three fields of importance: class name, attribute, and attribute value. For example, let's say a server wishes to register itself under the service name *MyServer*. This is accomplished when the server issues the bind call with the appropriate SOCKADDR_IRDA structure. Once this occurs, an IAS entry is added with the class name *MyServer*, the attribute IrDA:TinyTP:LsapSel, and an attribute value of, say, 3. The attribute value is the next unused LSAP-SEL assigned by the system upon registration. The client, on the other hand, passes in a SOCKADDR_IRDA structure to the connect call. This initiates an IAS lookup for a service with the class name *MyServer* and the attribute IrDA:TinyTP:LsapSel. The IAS query will return the value 3. You can

formulate your own IAS query by using the socket option *IRLMP_IAS_QUERY* in the *getsockopt* call.

If you want to bypass IAS altogether (which is not recommended), you can specify and LSAP-SEL address for a server name or an endpoint to which a client wants to connect. You should bypass IAS only to communicate with legacy IrDA devices that don't provide any kind of IAS registration (Such as infrared-capable printers). You can bypass the IAS registration and lookup by specifying the service name in the SOCKADDR_IRDA structure as LSAP-SEL-*xxx*, where *xxx* is the attribute value between 1 and 127. For a server, this would directly assign the server to the given LSAP-SEL address (assuming the LSAP-SEL address is unused). For a client, this bypasses the IAS lookup and causes an immediate attempt to connect to whatever service is running on that LSAP-SEL.

## Enumerating IrDA Devices

Because infrared devices move in and out of range, a method of dynamically listing all available infrared devices within range is necessary. This section describes how to accomplish that. Let's begin with a few platform discrepancies between the Windows CE implementation and the Windows 98 and Windows 2000 implementation. Windows CE supported IrSock before the platforms and provided minimal information about infrared devices. Later, Windows 98 and Windows 2000 provided support for IrSock, but they added additional “hint” information returned by the enumeration request. (This hint information will be discussed shortly.) As a result, the Af_irda.h header file for Windows CE contains the original, minimal structure definitions; however, the new header file for the other platforms contains conditional structure definitions for each platform that now supports IrSock. We recommend that you use the later Af_irda.h header file for consistency.

The way to enumerate nearby infrared devices is by the *IRLMP_ENUM_DEVICES* command for *getsockopt*. A *DEVICELIST* structure is passed as the *optval* parameter. There are two structures, one for Windows 98 and Windows 2000 and one for Windows CE. They are defined as

```
typedef struct _WINDOWS_DEVICELIST
{
```

```
    ULONG                           numDevice;
    WINDOWS_IRDA_DEVICE_INFO    Device[1];
} WINDOWS_DEVICELIST, *PWINDOWS_DEVICELIST, FAR *LPWINDOWS

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

The only difference between the Windows 98 and Window 2000 structure and the Windows CE structure is that the Windows 98 and Windows 2000 structure contains an array of *WINDOWS_IRDA_DEVICE_INFO* structures as opposed to an array of *WCE_IRDA_DEVICE_INFO* structures. A conditional *#define* directive declares *DEVICELIST* as the appropriate structure depending on the target platform. Likewise, two declarations for the *IRDA_DEVICE_INFO* structures exist:

```
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;

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;
```

Again, a conditional *#define* directive declares *IRDA_DEVICE_INFO* to the correct structure definition depending on the target platform.

As mentioned earlier, the function to use for the actual enumeration of infrared devices is *getsockopt* with the option *IRLMP_ENUM_DEVICES*. The following piece of code lists the device IDs of all infrared devices nearby.

```
SOCKET      sock;
```

```
DEVICELIST  devList;
DWORD       dwListLen=sizeof(DEVICELIST);

sock = WSASocket(AF_IRDA, SOCK_STREAM, 0, NULL, 0,
    WSA_FLAG_OVERLAPPED);
...
devList.numDevice = 0;
dwRet = getsockopt(sock, SOL_IRLMP, IRLMP_ENUMDEVICES,
    (char *)&devList, &dwListLen);
```

Before you pass a *DEVICELIST* structure into the *getsockopt* call, don't forget to set the *numDevice* field to 0. A successful enumeration will set the *numDevice* field to a value greater than 0 and set an equal number of *IRDA_DEVICE_INFO* structures in the *Device* field. Also, in an actual application you probably want to perform *getsockopt* more than once in order to check for devices that just moved into range. For example, attempting to discover an infrared device in five tries or less is a good heuristic. Simply place the call in a loop with a short call to the *Sleep* function after each unsuccessful enumeration.

Now that you know how to enumerate infrared devices, creating a client or a server is simple. The server side of the equation is a bit simpler because it looks like a "normal" server. That is, no extra steps are required. The general steps for an IrSock server are

1. Create a socket of address family *AF_IRDA* and socket type *SOCK_STREAM*.

2. Fill out a *SOCKADDR_IRDA* structure with the service name of the server.

3. Call *bind* with the socket handle and the *SOCKADDR_IRDA* structure.

4. Call *listen* with the socket handle and the backlog limit.

5. Block on an *accept* call for incoming clients.

The steps for a client are a bit more involved, as you must enumerate infrared devices. The following steps are necessary for an IrSock client:

1. Create a socket of address family *AF_IRDA* and socket type

*SOCK_STREAM*.

2. Enumerate available infrared devices by calling *getsockopt* with the *IRLMP_ENUM_DEVICES* option.

3. For each device returned, fill out a *SOCKADDR_IRDA* structure with the device ID returned and the service name you want to connect to.

4. Call the *connect* function with the socket handle and the *SOCKADDR_IRDA* structure. Do this for each structure filled out in step 3 until a connect succeeds.

## Querying IAS

There are two ways to find out whether a given service is running on a particular device. The first method is to actually attempt a connection to the service; the other is to query IAS for the given service name. Both methods require you to enumerate all infrared devices, and attempt a query (or connection) with each device until one of them succeeds or you have exhausted every device. Perform a query by calling *getsockopt* with the *IRLMP_IAS_QUERY* option. A pointer to an *IAS_QUERY* structure is passed as the *optval* parameter. Again, there are two *IAS_QUERY* structures, one for Windows 98 and Windows 2000, and another for Windows CE. Here are the definitions of each structure:

```
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;

typedef struct _WCE_IAS_QUERY
{
    (continued)    (continued) Socket Options
```

Many *SO_* socket options aren't meaningful to IrDA. Only *SO_LINGER* is specifically supported. The IrSock-specific socket options are of course supported only on sockets of the address family *AF_IRDA*. These options are also covered in Chapter 9, which summarizes all socket options and their parameters.

# IPX/SPX

The Internetwork Packet Exchange (IPX) protocol is commonly known as the protocol used with computers featuring Novell NetWare client/server networking services. IPX provides connectionless communication between two processes; therefore, if a workstation transmits a data packet, there is no guarantee that the packet will be delivered to the destination. If an application needs guaranteed delivery of data and insists on using IPX, it can use a higher-level protocol over IPX, such as the Sequence Packet Exchange (SPX) and SPX II protocols, in which SPX packets are transmitted through IPX. Winsock provides applications with the capability to communicate through IPX on Windows 95, Windows 98, Windows NT, and Windows 2000 but not on Windows CE.

## Addressing

In an IPX network, network segments are bridged together using an IPX router. Each network segment is assigned a unique 4-byte network number. As more network segments are bridged together, IPX routers manage communication between different network segments using the unique network segment numbers. When a computer is attached to a network segment, it is identified using a unique 6-byte node number, which is usually the physical address of the network adapter. A node (which is a computer) is typically capable of having one or more processes forming communication

over IPX. IPX uses socket numbers to distinguish communication for processes on a node.

To prepare a Winsock client or server application for IPX communication, you have to set up a *SOCKADDR_IPX* structure. The *SOCKADDR_IPX* structure is defined in the Wsipx.h header file, and your application must include this file after including Winsock2.h. The *SOCKADDR_IPX* structure is defined as

```
typedef struct sockaddr_ipx
{
    short          sa_family;
    char           sa_netnum[4];
    char           sa_nodenum[6];
    unsigned short sa_socket;
} SOCKADDR_IPX, *PSOCKADDR_IPX, FAR *LPSOCKADDR_IPX;
```

The *sa_family* field should always be set to the *AF_IPX* value. The *sa_netnum* field is a 4-byte number representing a network number of a network segment on an IPX network. The *sa_nodenum* field is a 6-byte number representing a node number of a computer's physical address. The *sa_socket* field represents a socket or port used to distinguish IPX communication on a single node.

## Creating a Socket

Creating a socket using IPX offers several possibilities. To open an IPX socket, call the *socket* function or the *WSASocket* function with the address family *AF_IPX*, the socket type *SOCK_DGRAM*, and the protocol *NSPROTO_IPX*, as follows:

```
s = socket(AF_IPX, SOCK_DGRAM, NSPROTO_IPX);

s = WSASocket(AF_IPX, SOCK_DGRAM, NSPROTO_IPX,
    NULL, 0, WSA_FLAG_OVERLAPPED);
```

Note that the third parameter protocol must be specified and cannot be 0. This is important because this field can be used to set specific IPX packet types.

As we mentioned earlier, IPX provides unreliable connectionless communication using datagrams. If an application needs reliable communication using IPX, it can use higher-level protocols over IPX, such as SPX and SPX II. This can be accomplished by setting the

type and protocol fields of the *socket* and *WSASocket* calls to the socket type *SOCK_SEQPACKET* or *SOCK_STREAM*, and the protocol *NSPROTO_SPX* or *NSPROTO_SPXII*.

If *SOCK_STREAM* is specified, data is transmitted as a continuous stream of bytes with no message boundaries—similar to the behavior of sockets in TCP/IP. On the other hand, if *SOCK_SEQPACKET* is specified, data is transmitted with message boundaries. For example, if a sender transmits 2000 bytes, the receiver won't return until all 2000 bytes have arrived. SPX and SPX II accomplish this by setting an end-of-message bit in an SPX header. When *SOCK_SEQPACKET* is specified, this bit is respected—meaning Winsock *recv* and *WSARecv* calls won't complete until a packet is received with this bit set. If *SOCK_STREAM* is specified, the end-of-message bit isn't respected, and *recv* completes as soon as any data is received, regardless of the setting of the end-of-message bit. From the sender's perspective (using the *SOCK_SEQPACKET* type), sends smaller than a single packet are always sent with the end-of-message bit set. Sends larger than single packets are packetized with the end-of-message bit set on only the last packet of the send.

## Binding a socket

When an IPX application associates a local address with a socket using *bind*, you shouldn't specify a network number and a node address in a *SOCKADDR_IPX* structure. The *bind* function populates these fields using the first IPX network interface available on the system. In the case of a machine with multiple network interfaces (a multihomed machine), it isn't necessary to bind to a specific interface. Windows 95, Windows 98, Windows NT, and Windows 2000 provide a virtual internal network in which each network interface can be reached regardless of the physical network it is attached to. We will describe internal network numbers in greater detail later in this chapter. After your application successfully binds to a local interface, you can retrieve local network number and node number information using the *getsockname* function, as in the following code fragment:

```
SOCKET sdServer;
SOCKADDR_IPX IPXAddr;
int addrlen = sizeof(SOCKADDR_IPX);
```

```
if ((sdServer = socket (AF_IPX, SOCK_DGRAM, NSPROTO_IPX))
    == INVALID_SOCKET)
{
    printf("socket failed with error %d\n",
        WSAGetLastError());
    return;
}

ZeroMemory(&IPXAddr, sizeof(SOCKADDR_IPX));
IPXAddr.sa_family = AF_IPX;
IPXAddr.sa_socket = htons(5150);

if (bind(sdServer, (PSOCKADDR) &IPXAddr, sizeof(SOCKADDR_I
    == SOCKET_ERROR)
{
    printf("bind failed with error %d\n",
        WSAGetLastError());
    return;
}

if (getsockname((unsigned) sdServer, (PSOCKADDR) &IPXAddr,
    == SOCKET_ERROR)
{
    printf("getsockname failed with error %d",
        WSAGetLastError());
    return;
}

// Print out SOCKADDR_IPX information returned from
// getsockname()
```

### Network number vs. internal network number

A network number (known as an external network number) identifies network segments in IPX and is used for routing IPX packets between network segments. Windows 95, Windows 98, Windows NT, and Windows 2000 also feature an internal network number that is used for internal routing purposes and to uniquely identify the computer on an inter-network (several networks bridged together). The internal network number is also known as a virtual network number—the internal network number identifies another (virtual) segment on the inter-network. Thus, if you configure an internal network number for a computer running Windows 95, Windows 98, Windows NT, or Windows 2000, a NetWare server or an IPX router will add an extra hop in its route to that computer.

The internal virtual network serves a special purpose in the case of a multihomed computer. When applications bind to a local network interface, they shouldn't specify local interface information but instead should set the *sa_netnum* and *sa_nodenum* fields of a *SOCKADDR_IPX* structure to 0. This is because IPX is able to route packets from any external network to any of the local network interfaces using the internal virtual network. For example, even if your application explicitly binds to the network interface on Network A, and a packet comes in on Network B, the internal network number will cause the packet to be routed internally so that your application receives it.

### Setting IPX packet types through Winsock

Winsock allows your application to specify IPX packet types when you create a socket using the *NSPROTO_IPX* protocol specification. The packet type field in an IPX packet indicates the type of service offered or requested by the IPX packet. In Novell, the following IPX packet types are defined:

- **01h** Routing Information Protocol (RIP) Packet
- **04h** Service Advertising Protocol (SAP) Packet
- **05h** Sequenced Packet Exchange (SPX) Packet
- **11h** NetWare Core Protocol (NCP) Packet
- **14h** Propagated Packet for Novell NetBIOS

To modify the IPX packet type, simply specify *NSPROTO_IPX* + *n* as the protocol parameter of the *socket* API, with *n* representing the packet type number. For example, to open an IPX socket that sets the packet type to 04h (SAP Packet), use the following *socket* call:

```
s = socket(AF_IPX, SOCK_DGRAM, NSPROTO_IPX + 0x04);
```

## Name Resolution

As you can probably tell, addressing IPX in Winsock is sort of ugly since you must supply multibyte network and node numbers to form an address. IPX provides applications with the ability to locate services by using user-friendly names to retrieve network number,

node number, and port number in an IPX network through the SAP protocol. As we will see in Chapter 10, Winsock 2 provides a protocol-independent method for name registration using the *WSASetService* API function. Through the SAP protocol, IPX server applications can use *WSASetService* to register under a user-friendly name the network number, node number, and port number they are listening on. Winsock 2 also provides a protocol-independent method of name resolution through the following API functions: *WSALookupServiceBegin*, *WSALookupServiceNext*, and *WSALookupServiceEnd*.

It is possible to perform your own name-service registration and lookups by opening an IPX socket and specifying an SAP packet type. After opening the socket, you can begin broadcasting SAP packets to the IPX network to register and locate services on the network. This requires that you understand the SAP protocol in great detail and that you deal with the programming details of decoding an IPX SAP packet.

# NetBIOS

The NetBIOS address family is yet another protocol family accessible from Winsock. You will be familiar with many of the topics and caveats discussed here from the NetBIOS discussion in Chapter 1. Addressing NetBIOS from Winsock still requires the knowledge of NetBIOS names and LANA numbers. We'll assume you've read those sections in Chapter 1, and we'll continue on with the specifics of accessing NetBIOS from Winsock.

> **Note** The NetBIOS address family is exposed by Winsock only on Windows NT and Windows 2000. It is not available on the Windows 95 and Windows 98 platforms or on Windows CE.

## Addressing

The basis for addressing a machine under NetBIOS is a NetBIOS name, which we covered in Chapter 1. To review, a NetBIOS name is 16 characters long, with the last character reserved as a qualifier to define what type of service the name belongs to. There are two types of NetBIOS names: unique and group. A unique name can be registered by only one process on the entire network. For example, a session-based server would register the name FOO, and clients

who wanted to contact that server would attempt a connection to FOO. Group names allow a group of applications to register the same name, so datagrams sent to that name will be received by all processes that registered that name.

In Winsock, the NetBIOS addressing structure is defined in Wsnetbs.h, as follows:

```
#define NETBIOS_NAME_LENGTH 16

typedef struct sockaddr_nb
{
    short   snb_family;
    u_short snb_type;
    char    snb_name[NETBIOS_NAME_LENGTH];
} SOCKADDR_NB, *PSOCKADDR_NB, FAR *LPSOCKADDR_NB;
```

The *snb_family* field specifies the address family of this structure and should always be set to *AF_NETBIOS*. The *snb_type* field is used to specify a unique or a group name. The following defines can be used for this field:

```
#define NETBIOS_UNIQUE_NAME          (0x0000)
#define NETBIOS_GROUP_NAME           (0x0001)
```

Finally, the *snb_name* field is the actual NetBIOS name.

Now that you know what each field means and what it should be set to, the following handy macro defined in the header file sets all of this for you:

```
#define SET_NETBIOS_SOCKADDR(_snb, _type, _name, _port)
    {
        int _i;
        (_snb)->snb_family = AF_NETBIOS;
        (_snb)->snb_type = (_type);
        for (_i = 0; _i < NETBIOS_NAME_LENGTH - 1; _i++) {
            (_snb)->snb_name[_i] = ' ';
        }
        for (_i = 0;
             *((_name) + _i) != '\0'
             && _i < NETBIOS_NAME_LENGTH - 1;
             _i++)
        {
            (_snb)->snb_name[_i] = *((_name)+_i);
```

```
        }
        (_snb)->snb_name[NETBIOS_NAME_LENGTH - 1] = (_port
    }
```

The first parameter to the macro, *_snb*, is the address of the *SOCKADDR_NB* structure you are filling in. As you can see, it automatically sets the *snb_family* field to *AF_NETBIOS*. For the *_type* parameter to the macro, specify *NETBIOS_UNIQUE_NAME* or *NETBIOS_GROUP_NAME*. The *_name* parameter is the NetBIOS name. The macro assumes it is either at least *NETBIOS_NAME_LENGTH* – 1 characters in length or is null-terminated if shorter. Notice that the *snb_name* field is prefilled with spaces. Finally, the macro sets the 16th character of the *snb_name* character string to the value of the *_port* parameter.

You can see that the NetBIOS name structure in Winsock is straightforward and shouldn't present any particular difficulties. The name resolution is performed under the hood, so unlike with TCP and IrDA, you don't have to resolve a name into a physical address before any operations. This becomes clear when you consider that NetBIOS is implemented over multiple protocols and each protocol has its own addressing scheme. In the next chapter, we'll present a simple client/server using the NetBIOS interface in Winsock.

## Creating a Socket

The most important consideration when you create a NetBIOS socket is the LANA number. Just as in the native NetBIOS API, you have to be aware of which LANA numbers concern your application. Remember that in order for a NetBIOS client and server to communicate, they must have a common transport protocol on which they both listen or connect. There are two ways to create a NetBIOS socket. The first is to call *socket* or *WSASocket*, as follows:

```
s = WSASocket(AF_NETBIOS, SOCK_DGRAM | SOCK_SEQPACKET, -la
              NULL, 0, WSA_FLAG_OVERLAPPED);
```

The *type* parameter of *WSASocket* is either *SOCK_DGRAM* or *SOCK_SEQPACKET* (don't specify both), depending on whether you want a connectionless datagram or a connection-oriented session socket. The third parameter, *protocol*, is the LANA number on which the socket should be created, except that you have to make it negative. The fourth parameter is null because you are specifying

your own parameters, not using a *WSAPROTOCOL_INFO* structure. The fifth parameter isn't used. Finally, the *dwFlags* parameter is set to *WSA_FLAG_OVERLAPPED*; you should specify *WSA_FLAG_OVERLAPPED* on all calls to *WSASocket*.

The drawback to the first method of socket creation is that you need to know which LANA numbers are valid to begin with. Unfortunately, Winsock doesn't have a nice, short method of enumerating available LANA numbers. The alternative in Winsock is to enumerate all transport protocols with *WSAEnumProtocols*. Of course, you could call the *Netbios* function with the *NCBENUM* command to get the valid LANAs. Chapter 5 described how to call *WSAEnumProtocols*. The following sample enumerates all transport protocols, searches for a NetBIOS transport, and creates a socket for each one.

```
dwNum = WSAEnumProtocols(NULL, lpProtocolBuf, &dwBufLen);
if (dwNum == SOCKET_ERROR)
{
    // Error
}
    (continued) for (i = 0; i < dwNum; i++)
{
    // Look for those entries in the AF_NETBIOS address fa
    if (lpProtocolBuf[i].iAddressFamily == AF_NETBIOS)
    {
        // Look for either SOCK_SEQPACKET or SOCK_DGRAM
        if (lpProtocolBuf[i].iSocketType == SOCK_SEQPACKET
        {
             s[j++] = WSASocket(FROM_PROTOCOL_INFO,
                FROM_PROTOCOL_INFO, FROM_PROTOCOL_INFO,
                   &lpProtocolBuf[i], 0, WSA_FLAG_OVERLAPP
        }
    }
}
```

In the above pseudocode, we enumerate the available protocols and iterate through them looking for those belonging to the *AF_NETBIOS* address family. Next we check the socket type, and in this case, look for entries of type *SOCK_SEQPACKET*. Otherwise, if we wanted datagrams we would check for *SOCK_DGRAM*. If this matches, we have a NetBIOS transport we can use. If you need the LANA number, take the absolute value of the *iProtocol* field in the *WSAPROTOCOL_INFO* structure. The only exception is for LANA 0. The *iProtocol* field for this LANA is 0x80000000 because 0 is

reserved for use by Winsock. The variable *j* will contain the number of valid transports.

# AppleTalk

AppleTalk support in Winsock has been around for a while, although few people are aware of it. AppleTalk probably won't be a protocol you choose to use unless you are communicating with Macintosh computers. AppleTalk is somewhat similar to NetBIOS in that it is name-based on a per-process basis. That is, a server dynamically registers a particular name that it will be known as. Clients use this name to establish a connection. However, AppleTalk names are substantially more complicated than NetBIOS names. The next section will discuss how computers using the AppleTalk protocol are addressed on the network.

## Addressing

An AppleTalk name is actually based on three separate names: the name, the type, and the zone. Each name can be up to 32 characters in length. The name identifies the process and its associated socket on a machine. The type is a subgrouping mechanism for zones. Traditionally, a zone is a network of AppleTalk-enabled computers physically located on the same loop. Microsoft's implementation of AppleTalk allows a Windows machine to specify the default zone it is located within. Multiple networks can be bridged together. These human-friendly names map to a socket number, a node number, and a network number. An AppleTalk name must be unique within the given type and zone. This requirement is enforced by the Name Binding Protocol (NBP), which broadcasts a query to see whether the name is already in use. Under the hood, AppleTalk uses the Routing Table Maintenance Protocol (RTMP) to dynamically discover routes to the different AppleTalk networks linked together.

The following structure provides the basis for addressing AppleTalk hosts from Winsock:

```
typedef struct sockaddr_at
{
    USHORT  sat_family;
    USHORT  sat_net;
    UCHAR   sat_node;
```

```
    UCHAR   sat_socket;
} SOCKADDR_AT, *PSOCKADDR_AT;
```

Notice that the address structure contains only characters or short integers and not friendly names. The *SOCKADDR_AT* structure is passed into Winsock calls such as *bind*, *connect*, and *WSAConnect*, but in order to translate the human-readable names you must query the network to either resolve or register that name first. This is done by using a call to *getsockopt* or *setsockopt*, respectively.

### Registering an AppleTalk name

A server that wants to register a particular name so that clients can easily connect to it calls *setsockopt* with the *SO_REGISTER_NAME* option. For all socket options involving AppleTalk names, use the *WSH_NBP_NAME* structure, which is defined as

```
typedef struct
{
    CHAR    ObjectNameLen;
    CHAR    ObjectName[MAX_ENTITY];
    CHAR    TypeNameLen;
    CHAR    TypeName[MAX_ENTITY];
    CHAR    ZoneNameLen;
    CHAR    ZoneName[MAX_ENTITY];
} WSH_NBP_NAME, *PWSH_NBP_NAME;
```

A number of types—such as *WSH_REGISTER_NAME*, *WSH_DEREGISTER_NAME*, and *WSH_REMOVE_NAME*—are defined based on the *WSH_NBP_NAME* structure. Using the appropriate type depends on whether you look up a name, register a name, or remove a name.

The following code sample illustrates how to register an AppleTalk name.

```
#define MY_ZONE    "*"
#define MY_TYPE    "Winsock-Test-App"
    (continued) #define MY_OBJECT    "AppleTalk-Server"

WSH_REGISTER_NAME    atname;
SOCKADDR_AT          ataddr;
SOCKET               s;
//
// Fill in the name to register
```

```
//
strcpy(atname.ObjectName, MY_OBJECT);
atname.ObjectNameLen = strlen(MY_OBJECT);
strcpy(atname.TypeName, MY_TYPE);
atname.TypeNameLen = strlen(MY_TYPE);
strcpy(atname.ZoneName, MY_ZONE);
atname.ZoneNameLen = strlen(MY_ZONE);

s = socket(AF_APPLETALK, SOCK_STREAM, ATPROTO_ADSP);
if (s == INVALID_SOCKET)
{
    // Error
}
ataddr.sat_socket = 0;
ataddr.sat_family = AF_APPLETALK;
if (bind(s, (SOCKADDR *)&ataddr, sizeof(ataddr)) == SOCKET
{
    // Unable to open an endpoint on the AppleTalk network
}
if (setsockopt(s, SOL_APPLETALK, SO_REGISTER_NAME,
               (char *)&atname, sizeof(WSH_NBP_NAME)) == S
{
    // Name registration failed!
}
```

The first thing you'll notice is the *MY_ZONE*, *MY_TYPE*, and *MY_OBJECT* strings. Remember that an AppleTalk name is three-tiered. Notice that the zone is an asterisk (*). This is a special character used in the zone field to specify the "current" zone the computer is located in. Next we create a socket of type *SOCK_STREAM* of the AppleTalk protocol ADSP. Following socket creation, you'll notice a call to the *bind* function with an address structure that has a zeroed-out *sat_socket* field and only the protocol family field set. This is important because it creates an endpoint on the AppleTalk network for your application to make requests from. Note that while this call to *bind* allows you to perform simple actions on the network, it doesn't by itself allow your application to accept incoming connection requests from clients. To accept client connections, you must register your name on the network, which is the next step.

Registering an AppleTalk name is simple. Make the call to *setsockopt* by passing *SOL_APPLETALK* as the *level* parameter and *SO_REGISTER_NAME* as the *optname* parameter. The last two parameters are a pointer to our *WSH_REGISTER_NAME* structure

and its size. If the call succeeds, our server name was successfully registered. If the call fails, the requested name is probably already in use by someone else. The Winsock error returned is *WSAEADDRINUSE* (10048 or 0x02740h). Note that for both datagram-oriented and stream-oriented AppleTalk protocols, a process that wants to receive data must register a name that clients can either send datagrams to or connect to.

## Resolving an AppleTalk name

On the client side of the equation, an application usually knows a server by its friendly name and must resolve that into the network, node, and socket numbers used by Winsock calls. This is accomplished by calling *getsockopt* with the *SO_LOOKUP_NAME* option. Performing a name lookup relies on the *WSH_LOOKUP_NAME* structure. This structure and its dependent structure are defined as

```
typedef struct
{
    WSH_ATALK_ADDRESS   Address;
    USHORT              Enumerator;
    WSH_NBP_NAME        NbpName;
} WSH_NBP_TUPLE, *PWSH_NBP_TUPLE;

typedef struct _WSH_LOOKUP_NAME
{
    //  Array of NoTuple WSH_NBP_TUPLEs
    WSH_NBP_TUPLE       LookupTuple;
    ULONG               NoTuples;
} WSH_LOOKUP_NAME, *PWSH_LOOKUP_NAME;
```

When we call *getsockopt* with the *SO_LOOKUP_NAME* option, we pass a buffer cast as a *WSH_LOOKUP_NAME* structure and fill in the *WSH_NBP_NAME* field within the first *LookupTuple* member. Upon a successful call, *getsockopt* returns an array of *WSH_NBP_TUPLE* elements containing physical address information for that name. Figure 6-1 contains the file Atalknm.c*,* which illustrates how to look up a name. In addition, it shows how to list all "discovered" AppleTalk zones and how to find your default zone. Zone information can be obtained by using the *getsockopt* options *SO_LOOKUP_ZONES* and *SO_LOOKUP_MYZONE*.

```
#include <winsock.h>
```

```
#include <atalkwsh.h>

#include <stdio.h>
#include <stdlib.h>

#define DEFAULT_ZONE       "*"
#define DEFAULT_TYPE       "Windows Sockets"
#define DEFAULT_OBJECT     "AppleTalk-Server"
char szZone[MAX_ENTITY],
     szType[MAX_ENTITY],
     szObject[MAX_ENTITY];

BOOL bFindName = FALSE,
     bListZones = FALSE,
     bListMyZone = FALSE;

void usage()
{
    printf("usage: atlookup [options]\n");
    printf("         Name Lookup:\n");
    printf("           -z:ZONE-NAME\n");
    printf("           -t:TYPE-NAME\n");
    printf("           -o:OBJECT-NAME\n");
    printf("         List All Zones:\n");
    printf("           -lz\n");
    printf("         List My Zone:\n");
    printf("           -lm\n");
    ExitProcess(1);
}

void ValidateArgs(int argc, char **argv)
{
    int                i;

    strcpy(szZone, DEFAULT_ZONE);
    strcpy(szType, DEFAULT_TYPE);
    strcpy(szObject, DEFAULT_OBJECT);

    for(i = 1; i < argc; i++)
    {
        if (strlen(argv[i]) < 2)
            continue;
        if ((argv[i][0] == '-') || (argv[i][0] == '/'))
        {
            switch (tolower(argv[i][1]))
            {
```

```
                case 'z':        // Specify a zone name
                    if (strlen(argv[i]) > 3)
                        strncpy(szZone, &argv[i][3], MAX_E
                    bFindName = TRUE;
                    break;
                case 't':        // Specify a type name
                    if (strlen(argv[i]) > 3)
                        strncpy(szType, &argv[i][3], MAX_E
                    bFindName = TRUE;
                    break;
                 case 'o':        // Specify an object nam
                    if (strlen(argv[i]) > 3)
                        strncpy(szObject, &argv[i][3], MAX
                    bFindName = TRUE;
                    break;
                case 'l':        // List zones information
                    if (strlen(argv[i]) == 3)
                        // List all zones
                        if (tolower(argv[i][2]) == 'z')
                            bListZones = TRUE;
                        // List my zone
                        else if (tolower(argv[i][2]) == 'm
                            bListMyZone = TRUE;
                    break;
                default:
                    usage();
            }
        }
    }
}

int main(int argc, char **argv)
{
    WSADATA              wsd;
    char                 cLookupBuffer[16000],
                        *pTupleBuffer = NULL;
    PWSH_NBP_TUPLE       pTuples = NULL;
    PWSH_LOOKUP_NAME     atlookup;
    PWSH_LOOKUP_ZONES    zonelookup;
    SOCKET               s;
    DWORD                dwSize = sizeof(cLookupBuffer);
    SOCKADDR_AT          ataddr;
    int                  i;

    // Load the Winsock library
    //
```

```
if (WSAStartup(MAKEWORD(2, 2), &wsd) != 0)
{
    printf("Unable to load Winsock library!\n");
    return 1;
}

ValidateArgs(argc, argv);

atlookup = (PWSH_LOOKUP_NAME)cLookupBuffer;
zonelookup = (PWSH_LOOKUP_ZONES)cLookupBuffer;
if (bFindName)
{
    // Fill in the name to look up
    //
    strcpy(atlookup->LookupTuple.NbpName.ObjectName, s
    atlookup->LookupTuple.NbpName.ObjectNameLen =
        strlen(szObject);
    strcpy(atlookup->LookupTuple.NbpName.TypeName, szT
    atlookup->LookupTuple.NbpName.TypeNameLen = strlen
    strcpy(atlookup->LookupTuple.NbpName.ZoneName, szZ
    atlookup->LookupTuple.NbpName.ZoneNameLen = strlen
}
// Create the AppleTalk socket
//
s = socket(AF_APPLETALK, SOCK_STREAM, ATPROTO_ADSP);
if (s == INVALID_SOCKET)
{
    printf("socket() failed: %d\n", WSAGetLastError())
    return 1;
}
// We need to bind in order to create an endpoint on t
// AppleTalk network to make our query from
//
ZeroMemory(&ataddr, sizeof(ataddr));
ataddr.sat_family = AF_APPLETALK;
ataddr.sat_socket = 0;
if (bind(s, (SOCKADDR *)&ataddr, sizeof(ataddr)) ==
        INVALID_SOCKET)
{
    printf("bind() failed: %d\n", WSAGetLastError());
    return 1;
}

if (bFindName)
{
    printf("Looking up: %s:%s@%s\n", szObject, szType,
```

```
    if (getsockopt(s, SOL_APPLETALK, SO_LOOKUP_NAME,
                (char *)atlookup, &dwSize) == INVALID_S
    {
        printf("getsockopt(SO_LOOKUP_NAME) failed: %d\
            WSAGetLastError());
        return 1;
    }
    printf("Lookup returned: %d entries\n",
        atlookup->NoTuples);
    //
    // Our character buffer now contains an array of
    // WSH_NBP_TUPLE structures after our WSH_LOOKUP_N
    // structure
     //
    pTupleBuffer = (char *)cLookupBuffer +
        sizeof(WSH_LOOKUP_NAME);
    pTuples = (PWSH_NBP_TUPLE) pTupleBuffer;

    for(i = 0; i < atlookup->NoTuples; i++)
    {
        ataddr.sat_family = AF_APPLETALK;
        ataddr.sat_net    = pTuples[i].Address.Network
        ataddr.sat_node   = pTuples[i].Address.Node;
        ataddr.sat_socket = pTuples[i].Address.Socket;
        printf("server address = %lx.%lx.%lx.\n",
            ataddr.sat_net,
            ataddr.sat_node,
            ataddr.sat_socket);
    }
}
else if (bListZones)
{
    // It is very important to pass a sufficiently big
    // for this option. Windows NT 4 SP3 blue screens
    // is too small.
    //
    if (getsockopt(s, SOL_APPLETALK, SO_LOOKUP_ZONES,
                (char *)atlookup, &dwSize) == INVALID_S
    {
        printf("getsockopt(SO_LOOKUP_NAME) failed: %d\
            WSAGetLastError());
        return 1;
    }
    printf("Lookup returned: %d zones\n", zonelookup->
    //
    // The character buffer contains a list of null-se
```

```
        //  strings after the WSH_LOOKUP_ZONES structure
        //
        pTupleBuffer = (char *)cLookupBuffer +
            sizeof(WSH_LOOKUP_ZONES);
        for(i = 0; i < zonelookup->NoZones; i++)
        {
            printf("%3d: '%s'\n", i+1, pTupleBuffer);
            while (*pTupleBuffer++);
        }
    }
    else if (bListMyZone)
    {
        // This option returns a simple string
        //
```

**Table 6-1. AppleTalk protocols and parameters**

| Protocol | Address Family | Socket Type | Protocol Type |
|---|---|---|---|
| MSAFD AppleTalk [ADSP] | | *SOCK_RDM* | *ATPROTO_ADSP* |
| *MSAFD AppleTalk [ADSP] [Pseudo-Stream]* | | *SOCK_STREAM* | *ATPROTO_ADSP* |
| MSAFD AppleTalk [PAP] | *AF_APPLETALK* | *SOCK_RDM* | *ATPROTO_PAP* |
| MSAFD AppleTalk [RTMP] | | *SOCK_DGRAM* | *DDPPROTO_RTMP* |
| MSAFD AppleTalk [ZIP] | | *SOCK_DGRAM* | *DDPPROTO_ZIP* |

# ATM

The Asynchronous Transfer Mode (ATM) protocol is one of the

newest protocols available that is supported by Winsock 2 on Windows 98 and Windows 2000. ATM is usually used for high-speed networking on LANs and WANs and can be used for all types of communication, such as voice, video, and data requiring high-speed communication. In general, ATM provides guaranteed quality of service (QOS) using Virtual Connections (VCs) on a network. As you will see in a moment, Winsock is capable of using VCs on an ATM network through the ATM address family. An ATM network—as shown in Figure 6-2—typically comprises endpoints (or computers) that are interconnected by switches that bridge an ATM network together.

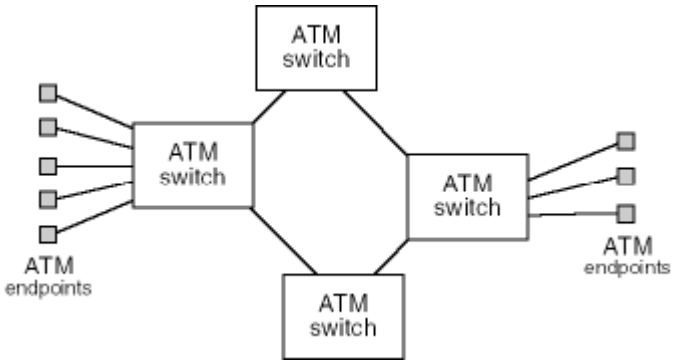


**Figure 6-2. ATM network**

There are a few things to be aware of when programming for the ATM protocol. First, ATM is a media type and not really a protocol per se. That is, ATM is similar to writing Ethernet frames directly on an Ethernet network. Like Ethernet, the ATM protocol doesn't provide flow control. It is a connection-oriented protocol that provides either message or stream modes. This also means that a sending application might overrun the local buffers if data cannot be sent quickly enough. Likewise, a receiving application must post receives frequently; otherwise, when the receiving buffers become full, any additional incoming data might be dropped. If your application requires flow control, one alternative is to use IP over ATM, which is simply the IP protocol running over an ATM network. As a result, the application follows the IP address family described above. Of course, ATM does offer some advantages over IP, such as a rooted multicast scheme (described in Chapter 12); however, the protocol that best suits you depends on your application's needs.

> **Note** Since ATM is new to Winsock 2, the information in this section was tested against ATM implementations on Windows 2000 Beta 3 only. Windows 98 (Service Pack 1) wasn't

available for testing at the time of this writing, and it is possible that some of the information might not conform to the final implementation details of both Windows 2000 and Windows 98 (Service Pack 1).

## Addressing

An ATM network has two network interfaces: the user network interface (UNI) and the network node interface (NNI). The UNI interface is the communication established between an endpoint and an ATM switch, while the NNI interface is the communication established between two switches. Each of these interfaces has a related communication protocol, described below.

- **UNI signaling protocol** Allows an endpoint to establish communication on an ATM network by sending setup and control information between an endpoint and an ATM switch. Note that this protocol is limited to transmissions between an endpoint and an ATM switch and isn't directly transmitted over an ATM network through switches.

- **NNI signaling protocol** Allows an ATM switch to communicate routing and control information between two switches.

For purposes of setting up an ATM connection through Winsock, we will only discuss certain information elements in the UNI signaling protocol. Winsock on Windows 2000 and Windows 98 (service pack 1) currently supports the UNI version 3.1 signaling protocol.

Winsock allows a client/server application to communicate over an ATM network by setting up a Service Access Point (SAP) to form connections using the ATM UNI signaling protocol. ATM is a connection-oriented protocol that requires endpoints to establish virtual connections across an ATM network for communication. An SAP simply allows Winsock applications to register and identify a socket interface for communication on an ATM network through a *SOCKADDR_ATM* address structure. Once an SAP is established, Winsock uses the SAP to establish a virtual connection between a Winsock client and server over ATM by making calls to the ATM network using the UNI signaling protocol. The *SOCKADDR_ATM* structure is defined as

```
typedef struct sockaddr_atm
```

```
{
    u_short       satm_family;
    ATM_ADDRESS   satm_number;
    ATM_BLLI      satm_blli;
    ATM_BHLI      satm_bhli;
} sockaddr_atm, SOCKADDR_ATM, *PSOCKADDR_ATM, *LPSOCKADDR_
```

The *satm_family* field should always be set to *AF_ATM*. The *satm_number* field represents an actual ATM address represented as an *ATM_ADDRESS* structure using one of two basic ATM addressing schemes: E.164 and Network Service Access Point (NSAP). NSAP addresses are also referred to as an NSAP-style ATM Endsystem Address (AESA). The *ATM_ADDRESS* structure is defined as

```
typedef struct
{
    DWORD AddressType;
    DWORD NumofDigits;
    UCHAR Addr[ATM_ADDR_SIZE];
} ATM_ADDRESS;
```

The *AddressType* field defines the specified addressing scheme. This should be set to *ATM_E164* for the E.164 addressing scheme and *ATM_NSAP* for the NSAP-style addressing scheme. Additionally, the *AddressType* field can be set to other values defined in Table 6-2 on the following page when an application tries to bind a socket to an SAP, which we will discuss in more detail later in this chapter. The *NumofDigits* field should always be set to *ATM_ADDR_SIZE*. The *Addr* field represents an actual ATM 20-byte E.164 or NSAP address.

The *satm_blli* and *satm_bhli* fields of the *SOCKADDR_ATM* structure represent Broadband Lower Layer Information (BLLI) and Broadband Higher Layer Information (BHLI) in ATM UNI signaling, respectively. In general, these structures are used to identify the protocol stack that operates over an ATM connection. Several well-known combinations of BHLI and BLLI values are described in ATM Form/IETF documents. (A particular combination of values identifies a connection as being used by LAN Emulation over ATM, another combination identifies native IP over ATM, and so on.) Complete ranges of values for the fields in these structures are given in the ATM UNI 3.1 standards book. ATM Form/IETF documents can be found at http://www.ietf.org.

**Table 6-2. ATM socket address types**

| ATM_ADDRESS AddressType Setting | Type of Address |
|---|---|
| *ATM_E164* | An E.164 address; applies when connecting to an SAP |
| *ATM_NSAP* | An NSAP-style ATM Endsystem Address (AESA); applies when connecting to an SAP |
| *SAP_FIELD_ANY_AESA_SEL* | An NSAP-style ATM Endsystem Address with the selector octet wildcarded; applies to binding a socket to an SAP |
| *SAP_FIELD_ANY_AESA_REST* | An NSAP-style ATM Endsystem Address with all the octets except for the selector octet wildcarded; applies to binding a socket to an SAP |

The BHLI and BLLI data structures are defined as

```
typedef struct
{
    DWORD HighLayerInfoType;
    DWORD HighLayerInfoLength;
    UCHAR HighLayerInfo[8];
} ATM_BHLI;

typedef struct
{
    DWORD Layer2Protocol;
    DWORD Layer2UserSpecifiedProtocol;
```

```
    DWORD Layer3Protocol;
    DWORD Layer3UserSpecifiedProtocol;
    DWORD Layer3IPI;
    UCHAR SnapID[5];
} ATM_BLLI;
```

Further details of the definition and use of these fields are beyond the scope of this book. An application that simply wants to form Winsock communication over an ATM network should set the following fields in the BHLI and BLLI structures to the *SAP_FIELD_ABSENT* value:

- ATM_BLLI—Layer2Protocol
- ATM_BLLI—Layer3Protocol
- ATM_BHLI—HighLayerInfoType

When these fields are set to this value, none of the other fields in both structures are used. The following pseudocode demonstrates how an application might use the *SOCKADDR_ATM* structure to set up an SAP for an NSAP address:

```
 SOCKADDR_ATM atm_addr;
UCHAR MyAddress[ATM_ADDR_SIZE];

atm_addr.satm_family                  = AF_ATM;
atm_addr.satm_number.AddressType      = ATM_NSAP;
atm_addr.satm_number.NumofDigits      = ATM_ADDR_SIZE;
atm_addr.satm_blli.Layer2Protocol     = SAP_FIELD_ABSENT;
atm_addr.satm_blli.Layer3Protocol     = SAP_FIELD_ABSENT;
atm_addr.satm_bhli.HighLayerInfoType = SAP_FIELD_ABSENT;

memcpy(&atm_addr.satm_number.Addr, MyAddress, ATM_ADDR_SIZ
```

ATM addresses are normally represented as a hexadecimal ASCII string of 40 characters, which corresponds to the 20 bytes that make up either an NSAP-style or an E.164 address in an *ATM_ADDRESS* structure. For example, an ATM NSAP-style address might look like this:

```
47000580FFE1000000F21A1D540000D10FED5800
```

Converting this string to a 20-byte address can be a rather tedious task. However, Winsock provides a protocol-independent API

function, *WSAStringToAddress*, which allows you to convert a 40-character ATM hexadecimal ASCII string to an *ATM_ADDRESS* structure. We describe this API function in more detail at the end of this chapter. Another way to convert a hexadecimal ASCII string to hexadecimal (binary) format is to use the function *AtoH* defined in Figure 6-3. This function isn't a part of Winsock. However, it is simple enough to develop, and you will see it in the samples in Chapter 7.

```
//
// Function: AtoH
//
// Description: This function coverts the ATM
// address specified in string (ASCII) format to
// binary (hexadecimal) format
//
void AtoH(CHAR *szDest, CHAR *szSource, INT iCount)
{
    while (iCount--)
    {
        *szDest++ = ( BtoH ( *szSource++ ) << 4 )
            + BtoH ( *szSource++ );
    }
    return;
}
//
// Function: BtoH
//
// Description: This function returns the equivalent
// binary value for an individual character specified
// in ASCII format
//
UCHAR BtoH( CHAR ch )
{
    if ( ch >= '0' && ch <= '9' )
    {
        return ( ch - '0' );
    }

    if ( ch >= 'A' && ch <= 'F' )
    {
        return ( ch - 'A' + 0xA );
    }

    if (ch >= 'a' && ch <= 'f' )
```

```
    {
        return ( ch - 'a' + 0xA );
    }
    //
    // Illegal values in the address will not be
    // accepted
    //
    return -1;
}Binding a Socket to an SAP
```

## Creating a Socket

In ATM, applications can create only connection-oriented sockets because ATM allows communication only over a VC. Therefore, data can be transmitted either as a stream of bytes or in a message-oriented fashion. To open a socket using the ATM protocol, call the *socket* function or the *WSASocket* function with the address family *AF_ATM* and the socket type *SOCK_RAW*, and set the protocol field to *ATMPROTO_AAL5*. For example:

```
s = socket(AF_ATM, SOCK_RAW, ATMPROTO_AAL5);

s = WSASocket(AF_ATM, SOCK_RAW, ATMPROTO_AAL5, NULL, 0,
    WSA_FLAG_OVERLAPPED);
```

By default, opening a socket (as in the example) creates a stream-oriented ATM socket. Windows also features an ATM provider that can perform message-oriented data transfers. Using the message-oriented provider requires you to explicitly specify the native ATM protocol provider to the *WSASocket* function by using a *WSAPROTOCOL_INFO* structure, as described in Chapter 5. This is necessary because the three elements in the *socket* call and the *WSASocket* call (address family, socket type, and protocol) match every ATM provider available in Winsock. By default, Winsock returns the protocol entry that matches those three attributes and that is marked as default, which in this case is the stream-oriented provider. The following pseudocode demonstrates how to retrieve the ATM message-oriented provider and establish a socket:

```
dwRet = WSAEnumProtocols(NULL, lpProtocolBuf, &dwBufLen);

for (i = 0; i < dwRet; i++)
{
    if ((lpProtocolBuf[i].iAddressFamily == AF_ATM) &&
```

```
        (lpProtocolBuf[i].iSocketType == SOCK_RAW) &&
        (lpProtocolBuf[i].iProtocol == ATMPROTO_AAL5) &&
        (lpProtocolBuf[i].dwServiceFlags1 &
          XP1_MESSAGE_ORIENTED))
    {
         s = WSASocket(FROM_PROTOCOL_INFO, FROM_PROTOCOL_IN
             FROM_PROTOCOL_INFO, lpProtocolBuf[i], 0,
             WSA_FLAG_OVERLAPPED);
    }
}
```

## Binding a Socket to an SAP

ATM addresses are actually quite complicated because the 20 bytes they comprise contain many informational elements. Winsock application programmers need not worry about all the specific details of these elements with the exception of the last byte. The last byte in NSAP-style and E.164 addresses represents a selector value that uniquely allows your application to define and specify a particular SAP on an endpoint. As we described earlier, Winsock uses an SAP to form communication over an ATM network.

When Winsock applications want to communicate over ATM, a server application must register an SAP on an endpoint and wait for a client application to connect on the registered SAP. For a client application, this simply involves setting up a *SOCKADDR_ATM* structure with the *ATM_E164* or *ATM_NSAP* address type and supplying the ATM address associated with the server's SAP. To create an SAP to listen for connections, your application must first create a socket for the *AF_ATM* address family. Once the socket is created, your application must define a *SOCKADDR_ATM* structure using the *SAP_FIELD_ANY_AESA_SEL*, *SAP_FIELD_ANY_AESA_REST*, *ATM_E164*, or *ATM_NSAP* address type as defined in Table 6-2 on page 168. For an ATM socket, an SAP will be created once your application calls the Winsock *bind* API function (which we describe in Chapter 7), and these address types define how Winsock creates an SAP on your endpoint.

The address type *SAP_FIELD_ANY_AESA_SEL* tells Winsock to create an SAP that is capable of listening for any incoming ATM Winsock connection, which is known as wildcarding an ATM address and the selector. This means that only one socket can be bound to this endpoint listening for any connection—if another socket tries to

bind with this address type, it will fail with Winsock error *WSAEADDRINUSE*. However, you can have another socket bound explicitly to your endpoint on a particular selector. The address type *SAP_FIELD_ANY_AESA_REST* can be used to create an SAP that is explicitly bound to a specified selector on an endpoint. This is known as wildcarding only the ATM address and not the selector. You can have only one socket at a time bound to a particular selector on an endpoint, or the *bind* call will fail with error *WSAEADDRINUSE*. When you use the *SAP_FIELD_ANY_AESA_SEL* type, you should specify an ATM address of all zeros in the *ATM_ADDRESS* structure. If you use *SAP_FIELD_ANY_AESA_REST*, you should specify all zeros for the first 19 bytes of the ATM address and the last byte should indicate what selector number you plan to use.

Sockets that are bound to explicit selectors (*SAP_FIELD_ANY_AESA_REST*) take higher precedence than those sockets that are bound to a wildcarded selector (*SAP_FIELD_ANY_AESA_SEL*). Those sockets that are bound to explicit selectors (*SAP_FIELD_ANY_AESA_REST*) or explicit interfaces (*ATM_NSAP* and *ATM_E164*) will get first choice at connections. (That is, if a connection comes in on the endpoint and the selector that a socket is explicitly listening on, that socket gets the connection.) Only when no explicitly bound socket is available will a wildcarded selector socket get the connection. Chapter 7 further demonstrates how to set up a socket that listens for connections on an SAP.

Finally, a utility named Atmadm.exe allows you to retrieve all ATM address and virtual connection information on an endpoint. This utility can be useful when you are developing an ATM application and need to know which interfaces are available on an endpoint. The command line options listed in the following table are available.

| Parameter | Description |
|---|---|
| -c | List all connections (VC). Lists the remote address and the local interface. |
| -a | Lists all registered addresses (i.e., all local ATM interfaces and their addresses). |
| -s | Prints statistics (current number of calls, number of signaling and ILMI packets sent/received, etc.). |

## Name Resolution

Currently no name providers are available for ATM under Winsock. This unfortunately requires applications to specify the 20-byte address ATM to establish socket communication over an ATM network. Chapter 10 discusses the Windows 2000 domain name space that can be generically used to register ATM addresses with user-friendly service names.

# Additional Winsock 2 Support Functions

Winsock 2 provides two useful support functions named *WSAAddressToString* and *WSAStringToAddress* that provide a protocol-independent method to convert a *SOCKADDR* structure of a protocol to a formatted character string and vice versa. Since these functions are protocol-independent, they require the transport protocol to support the string conversions. Currently these functions work only for the *AF_INET* and *AF_ATM* address families. The *WSAAddressToString* function is defined as

```
INT WSAAddressToString(
    LPSOCKADDR lpsaAddress,
    DWORD dwAddressLength,
    LPWSAPROTOCOL_INFO lpProtocolInfo,
    OUT LPTSTR lpszAddressString,
    IN OUT LPDWORD lpdwAddressStringLength
);
```

The *lpsaAddress* parameter represents a *SOCKADDR* structure for a particular protocol that contains the address to convert to a string. The *dwAddressLength* parameter specifies the size of the first parameter's structure, which can vary in size with different protocols. The *lpProtocolInfo* is an optional parameter that represents a protocol provider. Protocol providers can be retrieved from the *WSAEnumProtocols* API function, as described in Chapter 5. If you specify *NULL*, the call uses the provider of the first protocol supporting the address family indicated in *lpsaAddress.* The *lpszAddressString* parameter is a buffer that receives the human-readable address string. The *lpdwAddressStringLength* parameter represents the size of *lpszAddressString*. On output, it returns the length of the string actually copied into *lpszAddressString*. If the supplied buffer isn't large enough, the function fails with error

*WSAEFAULT* and the *lpdwAddressStringLength* parameter is updated with the required size in bytes.

Conversely, the *WSAStringToAddress* API function takes a human-readable address string and converts it to a *SOCKADDR* structure. *WSAStringToAddress* is defined as

```
INT WSAStringToAddress(
    LPTSTR AddressString,
    INT AddressFamily,
    LPWSAPROTOCOL_INFO lpProtocolInfo,
    LPSOCKADDR lpAddress,
    LPINT lpAddressLength
);
```

The *AddressString* parameter is a human-readable address string. Table 6-3 describes the format for this string for the current supported protocols.

**Table 6-3. Address string formats**

| Address Family | String Format |
|---|---|
| IP | *XXX.XXX.XXX.XXX*:*Y*—where *X* represents an octe address string and *Y* represents a port number |
| ATM | *NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN* where the 40 *N* characters represent a 20-byte AT in hexadecimal notation |

The *AddressFamily* parameter represents the address family type for the *AddressString* parameter. The *lpProtocolInfo* parameter is an optional parameter that represents a protocol provider. If you set this parameter to *NULL*, Winsock will search for the first available protocol provider for the address family type specified in the *AddressFamily* parameter. If you want to select a particular provider, the *WSAEnumProtocols* API function will supply you with a list of available protocol providers installed on your system. The *Address* buffer parameter takes a *SOCKADDR* structure that receives the information in the address string. The *lpAddressLength* parameter represents the size of the resultant *SOCKADDR* structure.

# Conclusion

In this chapter, we described the protocol address families supported by Winsock and explained addressing attributes specific to each family. For each address family, we discussed how to create a socket and how to set up a socket address structure to begin communication over a protocol. The next chapter will describe basic communication techniques available in Winsock, which apply to all of the address families described in this chapter.

---

# Chapter 7: Winsock Basics

This chapter is dedicated to learning the basic techniques and API calls necessary for writing successful network applications. In the last chapter, you learned how each protocol accessible from Winsock addresses machines and services on those machines. In this chapter, we'll look at establishing a connection from one machine on the network to another, along with how to send and receive data. For simplicity's sake, and to prevent repetition, the discussion in this chapter is limited to the TCP/IP protocol. However, this book's companion CD contains client/server samples for each of the protocols covered in Chapter 6. The only protocol-dependent operation is socket creation. Most of the remaining Winsock calls that are required for establishing a connection and for sending and receiving data are independent of the underlying protocol. The exceptions were noted in Chapter 6 along with each protocol discussion.

The examples presented in this chapter help to provide an understanding of the Winsock calls that are required for accepting connections, establishing connections, and sending and receiving data. Because the purpose of this chapter is to learn these Winsock calls, the examples presented use straight blocking Winsock calls. Chapter 8 presents the different I/O models available in Winsock, including code examples.

Additionally, in this chapter we will present both the Winsock 1 and Winsock 2 versions of the various API functions. You can differentiate the two functions with the *WSA* prefix. If Winsock 2 updated or added a new API function in its specification, the function name is prefixed with *WSA*. For example, the Winsock 1 function to create a socket is simply *socket*. Winsock 2 introduces a newer version named *WSASocket* that is capable of using some of the new features made available in Winsock 2. There are a few exceptions to this naming rule. *WSAStartup*, *WSACleanup*, *WSARecvEx*, and *WSAGetLastError* are in the Winsock 1.1 specification.

## Initializing Winsock

Every Winsock application must load the appropriate version of the Winsock DLL. If you fail to load the Winsock library before calling a

Winsock function, the function will return a *SOCKET_ERROR* and the error will be *WSANOTINITIALISED*. Loading the Winsock library is accomplished by calling the *WSAStartup* function, which is defined as

```
int WSAStartup(
    WORD wVersionRequested,
    LPWSADATA lpWSAData
);
```

The *wVersionRequested* parameter is used to specify the version of the Winsock library you want to load. The high-order byte specifies the minor version of the requested Winsock library, while the low-order byte is the major version. You can use the handy macro *MAKEWORD(x, y)*, in which *x* is the high byte and *y* is the low byte, to obtain the correct value for *wVersionRequested*.

The *lpWSAData* parameter is a pointer to a *LPWSADATA* structure that *WSAStartup* fills with information related to the version of the library it loads:

```
typedef struct WSAData
{
    WORD           wVersion;
    WORD           wHighVersion;
    char           szDescription[WSADESCRIPTION_LEN + 1];
    char           szSystemStatus[WSASYS_STATUS_LEN + 1];
    unsigned short iMaxSockets;
    unsigned short iMaxUdpDg;
    char FAR *     lpVendorInfo;
} WSADATA, FAR * LPWSADATA;
```

*WSAStartup* sets the first field, *wVersion*, to the Winsock version you will be using. The *wHighVersion* parameter holds the highest version of the Winsock library available. Remember that in both of these fields, the high-order byte represents the Winsock minor version, while the low-order byte is the major version. The *szDescription* and *szSystemStatus* fields are set by the particular implementation of Winsock and aren't really useful. Do not use the next two fields, *iMaxSockets* and *iMaxUdpDg*. They are supposed to be the maximum number of concurrently open sockets and the maximum datagram size; however, to find the maximum datagram size you should query the protocol information through *WSAEnumProtocols*. The maximum number of concurrent sockets

isn't some magic number—it depends more on how much physical memory is available. Finally, the *lpVendorInfo* field is reserved for vendor-specific information regarding the implementation of Winsock. This field is not used on any Win32 platforms.

Table 7-1 lists the latest versions of Winsock that the various Microsoft Windows platforms support. What's important to remember is the difference between major versions. Winsock 1.x does not support many of the advanced Winsock features detailed in this section. Additionally, for applications using Winsock 1, the include file Winsock.h is necessary; otherwise, for Winsock 2, Winsock2.h should be included.

**Table 7-1. Supported Winsock versions**

| Platform | Winsock Version |
|---|---|
| Windows 95 | 1.1 (2.2) |
| Windows 98 | 2.2 |
| Windows NT 4.0 | 2.2 |
| Windows 2000 | 2.2 |
| Windows CE | 1.1 |

> **Note** A Winsock 2 upgrade for Windows 95 is available for download from *http://www.microsoft.com/windows95/downloads/*.

Note that even though a platform supports Winsock 2, you do not have to request the latest version. That is, if you want to write an application that is supported on a majority of platforms, you should write it to the Winsock 1.1 specification. This application will run perfectly well on Windows NT 4.0 because all Winsock 1.1 calls are mapped through the Winsock 2 DLL. Also, if a newer version of the Winsock library becomes available for a platform that you use, it is often in your best interest to upgrade. These new versions contain bug fixes, and your old code should run without a problem—at least theoretically. In some cases, the behavior of the Winsock stack is different from what the specification defines. As a result, many programmers write their applications according to the behavior of the particular platform they are targeting instead of the

specification. For example, under Windows NT 4.0, when a program is using the asynchronous window event model, an *FD_WRITE* is posted after every successful *send* or *WSASend* to indicate that you can write data. However, the specification says that an *FD_WRITE* is posted when the system is able to send data, such as when the application starts, and that a posted *FD_WRITE* means you should keep writing until you receive the error *WSAEWOULDBLOCK*. In fact, after the system sends all pending data and can process more *send* and *WSASend* calls, it will post an *FD_WRITE* event to your application window, at which time you can resume writing data to the network (Knowledge Base Article Q186245). This problem has been fixed in Service Pack 4 for Windows NT 4.0 as well as in Windows 2000.

For the most part, however, when writing new applications you will load the latest version of the Winsock library currently available. Remember that if, for example, Winsock 3 is released, your application that loads version 2.2 should run as expected. If you request a Winsock version later than that which the platform supports, *WSAStartup* will fail. Upon return, the *wHighVersion* of the *WSADATA* structure will be the latest version supported by the library on the current system.

## Error Checking and Handling

We'll first cover error checking and error handling, as they are vital to writing a successful Winsock application. It is actually common for Winsock functions to return an error; however, many times the error is not critical and communication can still take place on that socket. The most common return value for an unsuccessful Winsock call is *SOCKET_ERROR*, although this is certainly not always the case. When covering each API call in detail, we'll point out the return value corresponding to an error. The constant *SOCKET_ERROR* actually is -1. If you make a call to a Winsock function and an error condition occurs, you can use the function *WSAGetLastError* to obtain a code that indicates specifically what happened. This function is defined as

```
int WSAGetLastError (void);
```

A call to the function after an error occurs will return an integer code for the particular error that occurred. These error codes returned from *WSAGetLastError* all have predefined constant values

that are declared in either Winsock.h or Winsock2.h, depending on the version of Winsock. The only difference between the two header files is that Winsock2.h contains more error codes for some of the newer API functions and capabilities introduced in Winsock 2. The constants defined for the various error codes (with *#define* directives) generally begin with *WSAE*.

# Connection-Oriented Protocols

In this first section, we'll cover the Winsock functions necessary for both receiving connections and establishing connections. We'll first discuss how to listen for client connections and explore the process for accepting or rejecting a connection. Then we'll describe how to initiate a connection to a server. Finally, we will discuss how data is transferred in a connection session.

## Server API Functions

A server is a process that waits for any number of client connections with the purpose of servicing their requests. A server must listen for connections on a well-known name. In TCP/IP, this name is the IP address of the local interface and a port number. Every protocol has a different addressing scheme and therefore a different naming method. The first step in Winsock is to bind a socket of the given protocol to its well-known name, which is accomplished with the *bind* API call. The next step is to put the socket into listening mode, which is performed (appropriately enough) with the *listen* API function. Finally, when a client attempts a connection, the server must accept the connection with either the *accept* or the *WSAAccept* call. In the next few sections, we will discuss each API call that is required for binding and listening and for accepting a client connection. Figure 7-1 illustrates the basic calls a server and a client must perform in order to establish a communication channel.

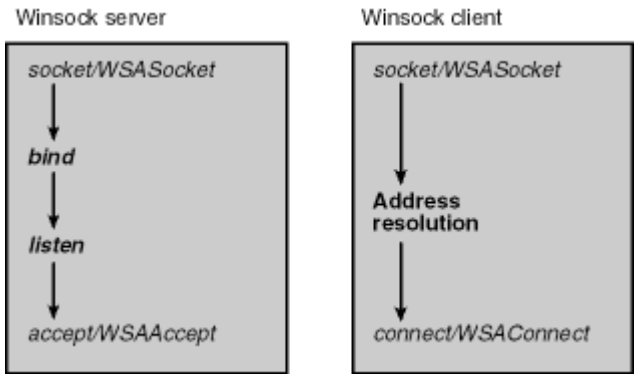

**Figure 7-1. Winsock basics for server and client**

## bind

Once the socket of a particular protocol is created, you must bind the socket to a well-known address. The *bind* function associates the given socket with a well-known address. This function is declared as

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

The first parameter, *s*, is the socket on which you want to wait for client connections. The second parameter is of type *struct sockaddr*, which is simply a generic buffer. You must actually fill out an address buffer specific to the protocol you are using and cast that as a *struct sockaddr* when calling *bind*. The Winsock header file defines the type *SOCKADDR* as *struct sockaddr*. We'll use this type throughout the chapter for brevity. The third parameter is simply the size of the protocol-specific address structure being passed. For example, the following code illustrates how this is done on a TCP connection:

```
SOCKET              s;
struct sockaddr_in  tcpaddr;
int                 port = 5150;

    (continued) s = socket(AF_INET, SOCK_STREAM, IPPROTO_T

tcpaddr.sin_family = AF_INET;
tcpaddr.sin_port = htons(port);
tcpaddr.sin_addr.s_addr = htonl(INADDR_ANY);

bind(s, (SOCKADDR *)&tcpaddr, sizeof(tcpaddr));
```

If the structure *sockaddr_in* looks mysterious to you, consult the TCP/IP addressing section in Chapter 6. From the example, you'll see a stream socket being created, followed by setting up the TCP/IP address structure on which client connections will be accepted. In this case, the socket is being bound to the default IP interface on port number 5150. The call to *bind* formally establishes this association of the socket with the IP interface and port.

On error, *bind* returns *SOCKET_ERROR*. The most common error encountered with *bind* is *WSAEADDRINUSE*. When you're using TCP/IP, the *WSAEADDRINUSE* error indicates that another process is already bound to the local IP interface and port number or that the IP interface and port number are in the *TIME_WAIT* state. If you call *bind* again on a socket that is already bound, *WSAEFAULT* will be returned.

## listen

The next piece of the equation is to put the socket into listening mode. The *bind* function merely associates the socket with a given address. The API function that tells a socket to wait for incoming connections is *listen*, which is defined as

```
int listen(
    SOCKET s,
    int    backlog
);
```

Again, the first parameter is a bound socket. The *backlog* parameter specifies the maximum queue length for pending connections. This is important when several simultaneous requests are made to the server. For example, let's say the backlog parameter is set to 2. If three client requests are made at the same time, the first two will be placed in a "pending" queue so that the application can service their requests. The third connection request will fail with *WSAECONNREFUSED*. Note that once the server accepts a connection, the connection request is removed from the queue so that others can make a request. The *backlog* parameter is silently limited to a value determined by the underlying protocol provider. Illegal values are replaced with their nearest legal values. Additionally, there is no standard provision for finding the actual backlog value.

The errors associated with *listen* are fairly straightforward. By far the most common is *WSAEINVAL,* which usually indicates that you forgot to call *bind* before *listen*. Otherwise, it is possible to receive the *WSAEADDRINUSE* error on the *listen* call as opposed to the *bind* call. This error occurs most often on the *bind* call.

## accept and WSAAccept

Now you're ready to accept client connections. This is accomplished

with either the *accept* or the *WSAAccept* function. The prototype for *accept* is

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

Parameter *s* is the bound socket that is in a listening state. The second parameter should be the address of a valid *SOCKADDR_IN* structure, while *addrlen* should be a reference to the length of the *SOCKADDR_IN* structure. For a socket of another protocol, substitute the *SOCKADDR_IN* with the *SOCKADDR* structure corresponding to that protocol. A call to *accept* services the first connection request in the queue of pending connections. When the *accept* function returns, the *addr* structure contains the IP address information of the client making the connection request, while the *addrlen* parameter indicates the size of the structure. Additionally, *accept* returns a new socket descriptor that corresponds to the accepted client connection. For all subsequent operations with this client, the new socket should be used. The original listening socket is still used to accept other client connections and is still in listening mode.

Winsock 2 introduced the function *WSAAccept*, which has the ability to conditionally accept a connection based on the return value of a condition function. The prototype for this new function is

```
SOCKET WSAAccept(
    SOCKET s,
    struct sockaddr FAR * addr,
    LPINT addrlen,
    LPCONDITIONPROC lpfnCondition,
    DWORD dwCallbackData
);
```

The first three parameters are the same as the Winsock 1 version of *accept*. The *lpfnCondition* argument is a pointer to a function that is called upon a client request. This function determines whether to accept the client's connection request. The prototype for this function is

```
int CALLBACK ConditionFunc(
    LPWSABUF lpCallerId,
```

```
    LPWSABUF lpCallerData,
    LPQOS lpSQOS,
    LPQOS lpGQOS,
    LPWSABUF lpCalleeId,
    LPWSABUF lpCalleeData,
    GROUP FAR * g,
    DWORD dwCallbackData
);
```

The *lpCallerId* parameter is a value parameter that contains the address of the connecting entity. The *WSABUF* structure is commonly used by many Winsock 2 functions. It is declared as

```
typedef struct __WSABUF {
    u_long     len;
    char FAR * buf;
} WSABUF, FAR * LPWSABUF;
```

Depending on its use, the *len* field refers either to the size of the buffer pointed to by the *buf* field or to the amount of data contained in the data buffer *buf*.

For *lpCallerId*, the *buf* pointer points to an address structure for the given protocol on which the connection is made. To correctly access the information, simply cast the *buf* pointer to the appropriate *SOCKADDR* type. In the case of TCP/IP, this is, of course, a *SOCKADDR_IN* structure that will contain the IP address of the client making the connection. Most network protocols can be expected to support caller ID information at connection-request time.

The *lpCallerData* parameter contains any connection data sent by the client along with the connection request. If caller data was not specified, this parameter is *NULL*. Be aware that most network protocols, such as TCP, do not support connect data. Whether a protocol supports connect or disconnect data can be determined by consulting its entry in the Winsock catalog with the *WSAEnumProtocols* function. See Chapter 5 for the specifics.

The next two parameters, *lpSQOS* and *lpGQOS*, specify any quality of service (QOS) parameters that are being requested by the client. Both parameters reference a QOS structure that contains information regarding bandwidth requirements for both sending and receiving data. If the client is not requesting QOS, these parameters

will be *NULL*. The difference between these two parameters is that *lpSQOS* refers to a single connection, while *lpGQOS* is used for socket groups. Socket groups are not implemented or supported in Winsock 1 or 2. (See Chapter 12 for further details about QOS.)

The *lpCalleeId* is another *WSABUF* structure containing the local address to which the client has connected. Again, the *buf* field of this structure points to a *SOCKADDR* object of the appropriate address family. This information is useful in the event that the server is running on a multihomed machine. Remember that if a server binds to the address *INADDR_ANY*, connection requests are serviced on any network interface. This parameter will contain the specific interface on which the connection occurred.

The *lpCalleeData* parameter is the complement of *lpCallerData*. The *lpCalleeData* parameter points to a *WSABUF* structure that the server can use to send data back to the client as a part of the connection request process. If the service provider supports this option, the *len* field indicates the maximum number of bytes the server can send back to the client as a part of this connection request. In this case, the server would copy any number of bytes up to this amount into the *buf* portion of the *WSABUF* structure and update the *len* field to indicate the number of bytes being transferred. If the server does not want to return any connect data, the conditional accept function should set the *len* field to 0 before returning. If the provider does not support connect data, the *len* field will be 0. Again, most protocols do not support data exchange upon accept. In fact, none of the currently supported protocols on any Win32 platform support this feature.

Once the server has processed parameters passed into the conditional function, the server must indicate whether to accept, reject, or defer the client's connection request. If the server is accepting the connection, the conditional function should return *CF_ACCEPT*. Upon rejection, the function should return *CF_REJECT*. If for some reason the decision cannot be made at this time, *CF_DEFER* can be returned. When the server is prepared to handle this connection request, it should call *WSAAccept*. Note that the condition function runs in the same thread as the *WSAAccept* function and should return as soon as possible. Also be aware that for the protocols supported by the current Win32 platforms, the conditional accept function does not imply that the client's connection request is delayed until a value is returned from this

conditional function. In most cases, the underlying network stack has already accepted the connection at the time the conditional accept function is called. If the value *CF_REJECT* is returned, the underlying stack simply closes the connection. We won't go into the detailed usage of the conditional acceptance function now, as this information will be more useful in Chapter 12.

If an error occurs, *INVALID_SOCKET* is returned. The most common error encountered is *WSAEWOULDBLOCK* if the listening socket is in asynchronous or nonblocking mode and there is no connection to be accepted. When a conditional function returns *CF_DEFER*, *WSAAccept* returns the error *WSATRY_AGAIN*. If the condition function returns *CF_REJECT*, the *WSAAccept* error is *WSAECONNREFUSED*.

## Client API Functions

The client is much simpler and involves fewer steps to set up a successful connection. There are only three steps for a client:

1. Create a socket with *socket* or *WSASocket*.

2. Resolve the server's name (dependent on underlying protocol).

3. Initiate the connection with *connect* or *WSAConnect*.

You already know from Chapter 6 how to create the socket and resolve an IP host name, so the only remaining step is establishing a connection. Chapter 6 also covers the various name-resolution methods for other protocol families.

> **TCP States** As a Winsock programmer, you are not required to know the actual TCP states, but by knowing them you will gain a better understanding of how the Winsock API calls effect change in the underlying protocol. Additionally, many programmers run into a common problem when closing sockets; the TCP states surrounding a socket closure are of the most interest.
>
> The start state of every socket is the CLOSED state. When a client initiates a connection, it sends a SYN packet to the server and puts the client socket in the SYN_SENT state. When the server receives the SYN packet, it sends a SYN-and-ACK packet, which the client responds to with an ACK packet. At this point,

the client's socket is in the ESTABLISHED state. If the server never sends a SYN-ACK packet, the client times out and reverts to the CLOSED state.

When a server's socket is bound and is listening on a local interface and port, the state of the socket is LISTEN. When a client attempts a connection, the server receives a SYN packet and responds with a SYN-ACK packet. The state of the server's socket changes to SYN_RCVD. Finally, the client sends an ACK packet, which causes the state of the server's socket to change to ESTABLISHED.

Once the application is in the ESTABLISHED state, there are two paths for closure. If your application initiates the closure, the closure is known as an active socket closure; otherwise, the socket closure is passive. Figure 7-2 illustrates both an active and a passive closure. If you actively initiate a closure, your application sends a FIN packet. When your application calls *closesocket* or *shutdown* (with *SD_SEND* as its second argument), your application sends a FIN packet to the peer, and the state of your socket changes to FIN_WAIT_1. Normally, the peer responds with an ACK packet, and your socket's state becomes FIN_WAIT_2. If the peer also closes the connection, it sends a FIN packet and your computer responds by sending an ACK packet and placing your socket in the TIME_WAIT state.

The TIME_WAIT state is also called the 2MSL wait state. MSL stands for Maximum Segment Lifetime and represents the amount of time a packet can exist on the network before being discarded. Each IP packet has a time-to-live (TTL) field, which when decremented to 0 causes the packet to be discarded. Each router on the network that handles the packet decrements the TTL by 1 and passes the packet on. Once an application enters the TIME_WAIT state, it remains there for twice the MSL time. This allows TCP to re-send the final ACK in case it's lost, causing the FIN to be retransmitted. After the 2MSL wait state completes, the socket goes to the CLOSED state.

On an active close, two other paths lead to the TIME_WAIT state. In our previous discussion, only one side issues a FIN and receives an ACK response, but the peer is still free to send data until it too closes. This is where the other two paths come into play. In one path—the simultaneous close—a computer and its

peer at the other side of a connection issue a close at the same time: the computer sends a FIN packet to the peer and receives a FIN packet from

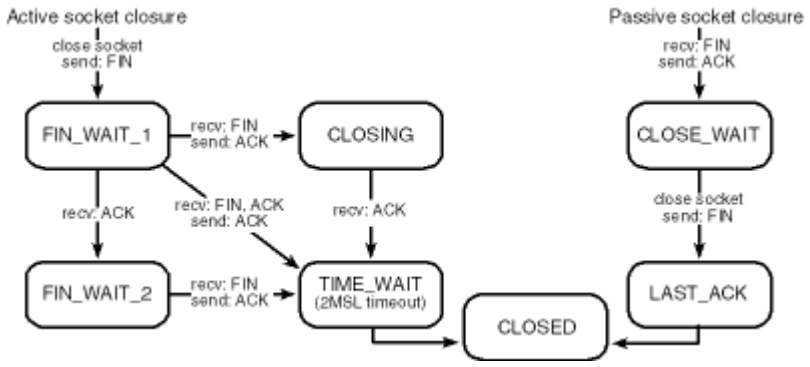


**Figure 7-2. TCP socket closure states**

the peer. Then the computer sends an ACK packet in response to the peer's FIN packet and changes its socket to the CLOSING state. Once the computer receives the last ACK packet from the peer, the computer's socket state becomes TIME_WAIT.

The other path for an active closure is just a variation on the simultaneous close: the socket transitions from the FIN_WAIT_1 state directly to the TIME_WAIT state. This occurs when an application sends a FIN packet but shortly thereafter receives a FIN-ACK packet from the peer. In this case, the peer is acknowledging the application's FIN packet and sending its own, to which the application responds with an ACK packet.

The major effect of the TIME_WAIT state is that while a TCP connection is in the 2MSL wait state, the socket pair defining that connection cannot be reused. A socket pair is the combination of local IP–local port and remote IP-remote port. Some TCP implementations do not allow the reuse of any port number in a socket pair in the TIME_WAIT state. Microsoft's implementation does not suffer from this deficiency. However, if a connection is attempted in which the socket pair is already in the TIME_WAIT state, the connection attempt will fail with error *WSAEADDRINUSE*. One way around this (besides waiting for the socket pair that is using that local port to leave the TIME_WAIT state) is to use the socket option *SO_REUSEADDR*. Chapter 9 covers the *SO_REUSEADDR* option in detail.

The last point of discussion for socket states is the passive closure. In this scenario, an application receives a FIN packet

from the peer and responds with an ACK packet. At this point, the application's socket changes to the CLOSE_WAIT state. Because the peer has closed its end, it can't send any more data, but the application still can until it also closes its end of the connection. To close its end of the connection, the application sends its own FIN, causing the application's TCP socket state to become LAST_ACK. After the application receives an ACK packet from the peer, the application's socket reverts to the CLOSED state.

For more information regarding the TCP/IP protocol, consult RFC 793. This RFC and others can be found at *http://www.rfc-editor.org*.

### connect and WSAConnect

The only new step is the connect. This is accomplished by calling either *connect* or *WSAConnect*. First we'll look at the Winsock 1 version of this function, which is defined as

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

The parameters are fairly self-explanatory: *s* is the valid TCP socket on which to establish the connection, *name* is the socket address structure (*SOCKADDR_IN*) for TCP that describes the server to connect to, and *namelen* is the length of the *name* variable. The Winsock 2 version is defined as

```
int WSAConnect(
    SOCKET s,
    const struct sockaddr FAR * name,
    int namelen,
    LPWSABUF lpCallerData,
    LPWSABUF lpCalleeData,
    LPQOS lpSQOS,
    LPQOS lpGQOS
);
```

The first three parameters are exactly the same as the *connect* API function. The next two, *lpCallerData* and *lpCalleeData,* are string buffers used to send and receive data at the time of the connection

request. The *lpCallerData* parameter is a pointer to a buffer that holds data the client sends to the server with the connection request. The *lpCalleeData* parameter points to a buffer that will be filled with any data sent back from the server at the time of connection setup. Both of these variables are *WSABUF* structures, so the *len* field needs to be set to the length of data in the *buf* field that is to be transferred in the case of *lpCallerData.* For *lpCalleeData*, the *len* field refers to the length of the buffer in *buf* that can receive data back from the server. The last two parameters, *lpSQOS* and *lpGQOS*, refer to QOS structures that define the bandwidth requirements for both sending and receiving data on the connection to be established. The parameter *lpSQOS* is used to specify requirements for the socket *s*, while *lpGQOS* specifies the requirements for socket groups. Socket groups are not currently supported. A null value for *lpSQOS* indicates no application-specific QOS.

If the computer you're attempting to connect to does not have a process listening on the given port, the *connect* call fails with the error *WSAECONNREFUSED*. The other error you might encounter is *WSAETIMEDOUT*, which occurs if the destination you're trying to reach is unavailable (either because of a communication-hardware failure on the route to the host or because the host is not currently on the network).

## Data Transmission

Sending and receiving data is what network programming is all about. For sending data on a connected socket, there are two API functions: *send* and *WSASend*. The second function is specific to Winsock 2. Likewise, two functions are for receiving data on a connected socket: *recv* and *WSARecv*. The latter is also a Winsock 2 call.

An important thing to keep in mind is that all buffers associated with sending and receiving data are of the simple *char* type. That is, there are no UNICODE versions of these functions. This is especially significant on Windows CE, as it uses UNICODE by default. In situations in which you are using UNICODE, you have the option of sending a character string as is or casting it as a *char* *. The catch is that if you use the string length function to tell the Winsock API functions how many characters to send or receive, you must multiply this value by 2 because each character occupies 2 bytes of

the string array. The other option is to use *WideCharToMultiByte* to convert UNICODE to ASCII before passing the string data to the Winsock API functions.

Additionally, the error code returned by all send and receive functions is *SOCKET_ERROR*. Once an error is returned, call *WSAGetLastError* to obtain extended error information. The most common errors encountered are *WSAECONNABORTED* and *WSAECONNRESET*. Both of these deal with the connection being closed—either through a timeout or through the peer closing the connection. Another common error is *WSAEWOULDBLOCK*, which is normally encountered when either nonblocking or asynchronous sockets are used. This error basically means that the specified function cannot be completed at this time. In Chapter 8, we will describe various Winsock I/O methods that can help you avoid some of these errors.

### *send* and *WSASend*

The first API function to send data on a connected socket is *send*, which is prototyped as

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

The *SOCKET* parameter is the connected socket to send the data on. The second parameter, *buf*, is a pointer to the character buffer that contains the data to be sent. The third parameter, *len*, specifies the number of characters in the buffer to send. Finally, the *flags* parameter can be either 0, *MSG_DONTROUTE*, or *MSG_OOB*. Alternatively, the *flags* parameter can be a bitwise ORing of any of those flags. The *MSG_DONTROUTE* flag tells the transport not to route the packets it sends. It is up to the underlying transport to honor this request (for example, if the transport doesn't support this option, it will be ignored). The *MSG_OOB* flag signifies that the data should be sent out of band.

On a good return, *send* returns the number of bytes sent; otherwise, if an error occurs, *SOCKET_ERROR* is returned. A common error is *WSAECONNABORTED,* which occurs when the

virtual circuit terminates because of a timeout failure or a protocol error. When this occurs, the socket should be closed, as it is no longer usable. The error *WSAECONNRESET* occurs when the application on the remote host resets the virtual circuit by executing a hard close or terminating unexpectedly, or when the remote host is rebooted. Again, the socket should be closed after this error occurs. The last common error is *WSAETIMEDOUT*, which occurs when the connection is dropped because of a network failure or the remote connected system going down without notice.

The Winsock 2 version of the *send* API function, *WSASend*, is defined as

```
int WSASend(
    SOCKET s,
    LPWSABUF lpBuffers,
    DWORD dwBufferCount,
    LPDWORD lpNumberOfBytesSent,
    DWORD dwFlags,
    LPWSAOVERLAPPED lpOverlapped,
    LPWSAOVERLAPPED_COMPLETION_ROUTINE lpCompletionROUTINE
);
```

The socket is a valid handle to a connection session. The second parameter is a pointer to one or more *WSABUF* structures. This can be either a single structure or an array of such structures. The third parameter indicates the number of *WSABUF* structures being passed. Remember that each *WSABUF* structure is itself a character buffer and the length of that buffer. You might wonder why you would want to send more than one buffer at a time. This is called scatter-gather I/O and will be discussed later in this chapter; however, in the case of data sent using multiple buffers on a connected socket, each buffer is sent from the first to the last *WSABUF* structure in the array. The *lpNumberOfBytesSent* is a pointer to a *DWORD* that on return from the *WSASend* call contains the total number of bytes sent. The *dwFlags* parameter is equivalent to its counterpart in *send*. The last two parameters, *lpOverlapped* and *lpCompletionROUTINE*, are used for overlapped I/O. Overlapped I/O is one of the asynchronous I/O models supported by Winsock and is discussed in detail in Chapter 8.

The *WSASend* function sets *lpNumberOfBytesSent* to the number of bytes written. The function returns 0 on success and *SOCKET_ERROR* on any error, and generally encounters the same

errors as the *send* function.

### WSASendDisconnect

This function is rather specialized and not generally used. The function prototype is

```
int WSASendDisconnect (
    SOCKET s,
    LPWSABUF lpOUT boundDisconnectData
);
```

**Out-of-Band Data** When an application on a connected stream socket needs to send data that is more important than regular data on the stream, it can mark the important data as out-of-band (OOB) data. The application on the other end of a connection can receive and process OOB data through a separate logical channel that is conceptually independent of the data stream.

In TCP, OOB data is implemented via an urgent 1-bit marker (called URG) and a 16-bit pointer in the TCP segment header that identify a specific downstream byte as urgent data. Two specific ways of implementing urgent data currently exist for TCP. RFC 793, which describes TCP and introduces the concept of urgent data, indicates that the urgent pointer in the TCP header is a positive offset to the byte that follows the urgent data byte. However, RFC 1122 describes the urgent offset as pointing to the urgent byte itself.

The Winsock specification uses the term OOB to refer to both protocol-independent OOB data and TCP's implementation of OOB data (urgent data). In order to check whether pending data contains urgent data, you must call the *ioctlsocket* function with the *SIOCATMARK* option. Chapter 9 discusses how to use *SIOCATMARK*.

Winsock provides several methods for obtaining the urgent data. Either the urgent data is inlined so that it appears in the normal data stream, or in-lining can be turned off so that a discrete call to a receive function returns only the urgent data. The socket option *SO_OOBINLINE*, also discussed in detail in Chapter 9, controls the behavior of OOB data.

Telnet and Rlogin use urgent data for several reasons. However, unless you plan on writing your own Telnet or Rlogin, you should stay away from urgent data. It's not well defined and might be implemented differently on platforms other than Win32. If you require a method of signaling the peer for urgent reasons, implement a separate control socket for this urgent data and reserve the main socket connection for normal data transfers.

The function initiates a shutdown of the socket and sends disconnect data. Of course, this function is available only to those transport protocols that support graceful close and disconnect data. None of the transport providers currently support disconnect data. The *WSASendDisconnect* function behaves like a call to the *shutdown* function with an *SD_SEND* argument, but it also sends the data contained in its *boundDisconnectData* parameter. Subsequent sends are not allowed on the socket. Upon failure, *WSASendDisconnect* returns *SOCKET_ERROR*. This function can encounter some of the same errors as the *send* function.

### recv and WSARecv

The *recv* function is the most basic way to accept incoming data on a connected socket. This function is defined as

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

The first parameter, *s*, is the socket on which data will be received. The second parameter, *buf*, is the character buffer that will receive the data, while *len* is either the number of bytes you want to receive or the size of the buffer, *buf*. Finally, the *flags* parameter can be one of the following values: 0, *MSG_PEEK*, or *MSG_OOB*. Additionally, you can bitwise OR any one of these flags together. Of course, 0 specifies no special actions. *MSG_PEEK* causes the data that is available to be copied into the supplied receive buffer, but this data is not removed from the system's buffer. The number of bytes pending is also returned.

Message peeking is bad. Not only does it degrade performance, as you now need to make two system calls (one to peek and one

without the *MSG_PEEK* flag to actually remove the data), but it is also unreliable under certain circumstances. The data returned might not reflect the entire amount available. Also, by leaving data in the system buffers, the system has less and less space to contain incoming data. As a result, the system reduces the TCP window size for all senders. This prevents your application from achieving the maximum possible throughput. The best thing to do is to copy all the data you can into your own buffer and manipulate it there. You have seen the *MSG_OOB* flag before in the discussion on sending data. Refer to page 189 in that section for more information.

There are some considerations when using *recv* on a message- or datagram-based socket. In the event that the data pending is larger than the supplied buffer, the buffer is filled with as much data as it will contain. In this event, the *recv* call generates the error *WSAEMSGSIZE.* Note that the message-size error occurs with message-oriented protocols. Stream protocols buffer incoming data and will return as much data as the application requests, even if the amount of pending data is greater. Thus, for streaming protocols you will not encounter the *WSAEMSGSIZE* error.

The *WSARecv* function adds some new capabilities over *recv,* such as overlapped I/O and partial datagram notifications. The definition of *WSARecv* is

```
int WSARecv(
    SOCKET s,
    LPWSABUF lpBuffers,
    DWORD dwBufferCount,
    LPDWORD lpNumberOfBytesRecvd,
    LPDWORD lpFlags,
    LPWSAOVERLAPPED lpOverlapped,
    LPWSAOVERLAPPED_COMPLETION_ROUTINE lpCompletionROUTINE
);
```

Parameter *s* is the connected socket. The second and third parameters are the buffers to receive the data. The *lpBuffers* parameter is an array of *WSABUF* structures, while *dwBufferCount* indicates the number of *WSABUF* structures in the array. The *lpNumberOfBytesReceived* parameter points to the number of bytes received by this call if the receive operation completes immediately. The *lpFlags* parameter can be one of the values *MSG_PEEK*, *MSG_OOB*, or *MSG_PARTIAL* or a bitwise ORed combination of those values. The *MSG_PARTIAL* flag has several different meanings

depending on where it is used or encountered. For message-oriented protocols, this flag is set upon return from *WSARecv* (if the entire message could not be returned in this call because of insufficient buffer space). In this case, subsequent *WSARecv* calls set this flag until the entire message is returned, when the *MSG_PARTIAL* flag is cleared. If this flag is passed as an input parameter, the receive operation should complete as soon as data is available, even if it is only a portion of the entire message. The *MSG_PARTIAL* flag is used only with message-oriented protocols, not with streaming ones. Additionally, not all protocols support partial messages. The protocol entry for each protocol contains a flag indicating whether it supports this feature. See Chapter 5 for more information. The *lpOverlapped* and *lpCompletionROUTINE* parameters are used in overlapped I/O operations, discussed in Chapter 8.

### WSARecvDisconnect

This function is the opposite of *WSASendDisconnect* and is defined as follows:

```
int WSARecvDisconnect(
    SOCKET s,
    LPWSABUF lpInboundDisconnectData
);
```

Like its sending counterpart, the parameters are the connected socket handle and a valid *WSABUF* structure with the data to be received. The data received can only be disconnect data sent by a *WSASendDisconnect* on the other side; it cannot be used to receive normal data. Additionally, once the data is received, this function disables reception from the remote party, which is equivalent to calling the *shutdown* function with *SD_RECV*.

### WSARecvEx

The *WSARecvEx* function is a Microsoft-specific extension of Winsock 1 and is identical to the *recv* function except that the *flags* parameter is passed by reference. This allows the underlying provider to set the *MSG_PARTIAL* flag. The function prototype is as follows:

```
int PASCAL FAR WSARecvEx(
    SOCKET s,
```

```
    char FAR * buf,
    int len,
    int *flags
);
```

The *MSG_PARTIAL* flag is returned in the *flags* parameter if the data received is not a complete message. This flag is of interest for message-oriented (nonstream) protocols. If the *MSG_PARTIAL* flag is passed as a part of the *flags* parameter and a partial message is received, the call returns immediately with that data. If the supplied receive buffer is not large enough to hold an entire message, *WSARecvEx* fails with the *WSAEMSGSIZE* error and the remaining data is truncated. Note that the difference between a *MSG_PARTIAL* flag and a *WSAEMSGSIZE* error is that with the error, the whole message arrives but the supplied data buffer is too small to receive it. The *MSG_PEEK* and *MSG_OOB* flags can also be used with *WSARecvEx*.

## Stream Protocols

Because most connection-oriented protocols are also streaming protocols, we'll mention stream protocols here. The main thing to be aware of with *any* function that sends or receives data on a stream socket is that you are not guaranteed to read or write the amount of data you request. Let's say you have a character buffer with 2048 bytes of data you want to send with the *send* function. The code to send this is

```
char sendbuff[2048];
int  nBytes = 2048;

// Fill sendbuff with 2048 bytes of data

// Assume s is a valid, connected stream socket
ret = send(s, sendbuff, nBytes, 0);
```

It is possible for *send* to return having sent less than 2048 bytes. The *ret* variable will be set to the number of bytes sent because the system allocates a certain amount of buffer space for each socket to send and receive data. In the case of sending data, the internal buffers hold data to be sent until such time as the data can be placed on the wire. Several common situations can cause this. For example, simply transmitting a huge amount of data will cause these buffers to become filled quickly. Also, for TCP/IP, there is

what is known as the window size. The receiving end will adjust this window size to indicate how much data it can receive. If the receiver is being flooded with data, it might set the window size to 0 in order to catch up with the pending data. This will force the sender to stop until it receives a new window size greater than 0. In the case of our *send* call, there might only be buffer space to hold 1024 bytes, in which case you would have to resubmit the remaining 1024 bytes. The following code ensures that all your bytes are sent.

```
char sendbuff[2048];
int  nBytes = 2048,
     nLeft,
     idx;

// Fill sendbuff with 2048 bytes of data

// Assume s is a valid, connected stream socket
nLeft = nBytes;
idx = 0;
while (nLeft > 0)
{
    ret = send(s, &sendbuff[idx], nLeft, 0);
    if (ret == SOCKET_ERROR)
    {
        // Error
    }
    nLeft -= ret;
    idx += ret;
}
```

The foregoing holds true for receiving data on a stream socket but is less significant. Because stream sockets are a continuous stream of data, when an application reads it isn't generally concerned with how much data it should read. If your application requires discrete messages over a stream protocol, you might have to do a little work. If all the messages are the same size, life is pretty simple, and the code for reading, say, 512-byte messages would look like this:

```
char    recvbuff[1024];
int     ret,
        nLeft,
        idx;

nLeft = 512;
```

```
idx = 0;
    (continued) while (nLeft > 0)
{
    ret = recv(s, &recvbuff[idx], nLeft, 0);
    if (ret == SOCKET_ERROR)
    {
        // Error
    }
    idx += ret;
    nLeft -= ret;
}
```

Things get a little complicated if your message sizes vary. It is necessary to impose your own protocol to let the receiver know how big the forthcoming message will be. For example, the first 4 bytes written to the receiver will always be the integer size in bytes of the forthcoming message. The receiver will start every read by looking at the first 4 bytes, converting them to an integer, and determining how many additional bytes that message comprises.

**Scatter-Gather I/O** Scatter-gather support is a concept originally introduced in Berkeley Sockets with the functions *recv* and *writev*. This feature is available with the Winsock 2 functions *WSARecv*, *WSARecvFrom*, *WSASend*, and *WSASendTo*. It is most useful for applications that send and receive data that is formatted in a very specific way. For example, messages from a client to a server might always be composed of a fixed 32-byte header specifying some operation, followed by a 64-byte data block and terminated with a 16-byte trailer. In this example, *WSASend* can be called with an array of three *WSABUF* structures, each corresponding to the three message types. On the receiving end, *WSARecv* is called with three *WSABUF* structures, each containing data buffers of 32 bytes, 64 bytes, and 16 bytes.

When using stream-based sockets, scatter-gather operations simply treat the supplied data buffers in the *WSABUF* structures as one contiguous buffer. Also, the receive call might return before all buffers are full. On message-based sockets, each call to a receive operation receives a single message up to the buffer size supplied. If the buffer space is insufficient, the call fails with *WSAEMSGSIZE* and the data is truncated to fit the available space. Of course, with protocols that support partial messages, the *MSG_PARTIAL* flag can be used to prevent data loss.

## Breaking the Connection

Once you are finished with a socket connection, you must close the connection and release any resources associated with that socket handle. To actually release the resources associated with an open socket handle, use the *closesocket* call. Be aware, however, that *closesocket* can have some adverse affects—depending on how it is called—that can lead to data loss. For this reason, a connection should be gracefully terminated with the *shutdown* function before a call to the *closesocket* function. These two API functions are discussed next.

### *shutdown*

To ensure that all data an application sends is received by the peer, a well-written application should notify the receiver that no more data is to be sent. Likewise, the peer should do the same. This is known as a graceful close and is performed by the *shutdown* function, defined as

```
int shutdown(
    SOCKET s,
    int how
);
```

The *how* parameter can be *SD_RECEIVE*, *SD_SEND*, or *SD_BOTH*. For *SD_RECEIVE*, subsequent calls to any receive function on the socket are disallowed. This has no effect on the lower protocol layers. Additionally for TCP sockets, if data is queued for receive or if data subsequently arrives, the connection is reset. However, on UDP sockets incoming data is still accepted and queued. For *SD_SEND*, subsequent calls to any send function are disallowed. For TCP sockets, this causes a FIN packet to be generated after all data is sent and acknowledged by the receiver. Finally, specifying *SD_BOTH* disables both sends and receives.

### *closesocket*

The *closesocket* function closes a socket and is defined as

```
int closesocket (SOCKET s);
```

Calling *closesocket* releases the socket descriptor and any further

calls using the socket fail with *WSAENOTSOCK.* If there are no other references to this socket, all resources associated with the descriptor are released. This includes discarding any queued data.

Pending asynchronous calls issued by any thread in this process are canceled without posting any notification messages. Pending overlapped operations are also canceled. Any event, completion routine, or completion port that is associated with the overlapped operation is performed but will fail with the error *WSA_OPERATION_ABORTED*. Asynchronous and nonblocking I/O models are discussed in greater depth in Chapter 8. Additionally, one other factor influences the behavior of *closesocket*: whether the socket option *SO_LINGER* has been set. Consult the description for the *SO_LINGER* option in Chapter 9 for a complete explanation.

## Putting It All Together

You might be a bit overwhelmed by the multitude of functions for sending and receiving data, but in reality most applications only need either *recv* or *WSARecv* for receiving data and either *send* or *WSASend* for sending. The other functions are specialized with unique features not commonly used (or supported by the transport protocols). With this said, we'll discuss a simple client/server example using the principles and functions we've covered so far. Figure 7-3 contains the code for a simple echo server. This application creates a socket, binds to a local IP interface and port, and listens for client connections. Upon receipt of a client connection request, a new socket is created that is passed into a client thread that is spawned. The thread simply reads data and sends it back to the client.

```
// Module Name: Server.c
//
// Description:
//    This example illustrates a simple TCP server that ac
//    incoming client connections. Once a client connectio
//    established, a thread is spawned to read data from t
//    client and echo it back (if the echo option is not
//    disabled).
//
// Compile:
//    cl -o Server Server.c ws2_32.lib
//
// Command line options:
```

```
//    server [-p:x] [-i:IP] [-o]
//           -p:x      Port number to listen on
//           -i:str    Interface to listen on
//           -o        Receive only; don't echo the data b
//
#include <winsock2.h>

#include <stdio.h>
#include <stdlib.h>

#define DEFAULT_PORT        5150
#define DEFAULT_BUFFER      4096

int    iPort      = DEFAULT_PORT; // Port to listen for cl

 BOOL   bInterface = FALSE,   // Listen on the specified i
       bRecvOnly  = FALSE;   // Receive data only; don't e
char   szAddress[128];       // Interface to listen for cl

//
// Function: usage
//
// Description:
//    Print usage information and exit
//
void usage()
{
    printf("usage: server [-p:x] [-i:IP] [-o]\n\n");
    printf("       -p:x      Port number to listen on\n");
    printf("       -i:str    Interface to listen on\n");
    printf("       -o        Don't echo the data back\n\n"
    ExitProcess(1);
}

//
// Function: ValidateArgs
//
// Description:
//    Parse the command line arguments, and set some globa
//    to indicate what actions to perform
//
void ValidateArgs(int argc, char **argv)
{
    int i;

    for(i = 1; i < argc; i++)
```

```
    {
        if ((argv[i][0] == '-') || (argv[i][0] == '/'))
        {
            switch (tolower(argv[i][1]))
            {
                case 'p':
                    iPort = atoi(&argv[i][3]);
                    break;
                case 'i':
                    bInterface = TRUE;
                    if (strlen(argv[i]) > 3)
                        strcpy(szAddress, &argv[i][3]);
                    break;
                case 'o':
                    bRecvOnly = TRUE;
                    break;
                default:
                    usage();
                    break;
            }
        }
    }
}

//
// Function: ClientThread
//
// Description:
//    This function is called as a thread, and it handles
//    client connection.  The parameter passed in is the s
//    handle returned from an accept() call.  This functio
//    data from the client and writes it back.
//
DWORD WINAPI ClientThread(LPVOID lpParam)
{
    SOCKET        sock=(SOCKET)lpParam;
    char          szBuff[DEFAULT_BUFFER];
    int           ret,
                  nLeft,
                  idx;

    while(1)
    {
        // Perform a blocking recv() call
        //
        ret = recv(sock, szBuff, DEFAULT_BUFFER, 0);
```

```
        if (ret == 0)        // Graceful close
            break;
        else if (ret == SOCKET_ERROR)
        {
            printf("recv() failed: %d\n", WSAGetLastError(
            break;
        }
        szBuff[ret] = '\0';
        printf("RECV: '%s'\n", szBuff);
        //
        // If we selected to echo the data back, do it
        //
        if (!bRecvOnly)
        {
            nLeft = ret;
            idx = 0;
             //
            // Make sure we write all the data
            //
            while(nLeft > 0)
            {
                ret = send(sock, &szBuff[idx], nLeft, 0);
                if (ret == 0)
                    break;
                else if (ret == SOCKET_ERROR)
                {
                    printf("send() failed: %d\n",
                        WSAGetLastError());
                    break;
                }
                nLeft -= ret;
                idx += ret;
            }
        }
    }
    return 0;
}

//
// Function: main
//
// Description:
//    Main thread of execution. Initialize Winsock, parse
//    command line arguments, create the listening socket,
//    to the local address, and wait for client connection
//
```

```
int main(int argc, char **argv)
{
    WSADATA       wsd;
    SOCKET        sListen,
                  sClient;
    int           iAddrSize;
    HANDLE        hThread;
    DWORD         dwThreadId;
    struct sockaddr_in local,
                       client;

    ValidateArgs(argc, argv);
    if (WSAStartup(MAKEWORD(2,2), &wsd) != 0)
    {
        printf("Failed to load Winsock!\n");
        return 1;
    }
    // Create our listening socket
    //
    sListen = socket(AF_INET, SOCK_STREAM, IPPROTO_IP);
    if (sListen == SOCKET_ERROR)
    {
        printf("socket() failed: %d\n", WSAGetLastError())
        return 1;
    }
    // Select the local interface, and bind to it
    //
    if (bInterface)
    {
        local.sin_addr.s_addr = inet_addr(szAddress);
        if (local.sin_addr.s_addr == INADDR_NONE)
            usage();
    }
    else
        local.sin_addr.s_addr = htonl(INADDR_ANY);
    local.sin_family = AF_INET;
    local.sin_port = htons(iPort);

    if (bind(sListen, (struct sockaddr *)&local,
            sizeof(local)) == SOCKET_ERROR)
    {
        printf("bind() failed: %d\n", WSAGetLastError());
        return 1;
    }
    listen(sListen, 8);
    //
```

```
    // In a continuous loop, wait for incoming clients. On
    // is detected, create a thread and pass the handle of
    //
    while (1)
    {
        iAddrSize = sizeof(client);
        sClient = accept(sListen, (struct sockaddr *)&clie
                         &iAddrSize);
        if (sClient == INVALID_SOCKET)
        {
            printf("accept() failed: %d\n", WSAGetLastErro
            break;
        }
        printf("Accepted client: %s:%d\n",
            inet_ntoa(client.sin_addr), ntohs(client.sin_p

        hThread = CreateThread(NULL, 0, ClientThread,
                     (LPVOID)sClient, 0, &dwThreadId);
         if (hThread == NULL)
        {
            printf("CreateThread() failed: %d\n", GetLastE
            break;
        }
        CloseHandle(hThread);
    }
    closesocket(sListen);

    WSACleanup();
    return 0;
}
```

**Figure 7-3. Echo server code**

The client for this example, provided in Figure 7-4, is even more basic. The client creates a socket, resolves the server name passed into the application, and connects to the server. Once the connection is made, a number of messages are sent. After each send, the client waits for an echo response from the server. The client prints all data read from the socket.

The echo client and server don't fully illustrate the streaming nature of TCP. This is because a read operation follows every write operation, at least in the client's case. Of course, it is the other way around for the server. Thus, each call to the read function by the server will almost always return the full message that the client

sent. Don't be misled by this. If the client's messages become large enough to exceed the maximum transmission unit for TCP, the message will be broken up into separate packets on the wire, in which case the receiver needs to perform a receive call multiple times. In order to better illustrate streaming, run the client and the server with the -o option. This causes the client to only send data and the receiver to only read data. Execute the server like this:

```
server –p:5150 –o
```

and the client like this:

```
client –p:5150 –s:IP –n:10 –o
```

What you'll most likely see is that the client calls *send* 10 times, but the server reads all 10 messages in one or two *recv* calls.

```
// Module Name: Client.c
//
// Description:
//    This sample is the echo client. It connects to the T
//    sends data, and reads data back from the server.
//
// Compile:
//    cl -o Client Client.c ws2_32.lib
//
// Command Line Options:
//    client [-p:x] [-s:IP] [-n:x] [-o]
//           -p:x      Remote port to send to
//           -s:IP     Server's IP address or host name
//           -n:x      Number of times to send message
//           -o        Send messages only; don't receive
//
#include <winsock2.h>
#include <stdio.h>
#include <stdlib.h>

#define DEFAULT_COUNT       20
#define DEFAULT_PORT        5150
#define DEFAULT_BUFFER      2048
#define DEFAULT_MESSAGE     "This is a test of the emergen
broadcasting system"

char  szServer[128],          // Server to connect to
      szMessage[1024];        // Message to send to sever
int   iPort     = DEFAULT_PORT;  // Port on server to conn
```

```
DWORD dwCount   = DEFAULT_COUNT; // Number of times to sen
BOOL  bSendOnly = FALSE;         // Send data only; don't

//
// Function: usage:
//
// Description:
//    Print usage information and exit
//
void usage()
{
    printf("usage: client [-p:x] [-s:IP] [-n:x] [-o]\n\n")
    printf("       -p:x      Remote port to send to\n");
    printf("       -s:IP     Server's IP address or host n
    printf("       -n:x      Number of times to send messa
    printf("       -o        Send messages only; don't rec
    ExitProcess(1);
}

//
// Function: ValidateArgs
//
// Description:
//    Parse the command line arguments, and set some globa
//    to indicate what actions to perform
//
 void ValidateArgs(int argc, char **argv)
{
    int               i;

    for(i = 1; i < argc; i++)
    {
        if ((argv[i][0] == '-') || (argv[i][0] == '/'))
        {
            switch (tolower(argv[i][1]))
            {
                case 'p':        // Remote port
                    if (strlen(argv[i]) > 3)
                        iPort = atoi(&argv[i][3]);
                    break;
                case 's':       // Server
                    if (strlen(argv[i]) > 3)
                        strcpy(szServer, &argv[i][3]);
                    break;
                case 'n':       // Number of times to send
                    if (strlen(argv[i]) > 3)
```

```
                        dwCount = atol(&argv[i][3]);
                    break;
                case 'o':       // Only send message; don'
                    bSendOnly = TRUE;
                    break;
                default:
                    usage();
                    break;
            }
        }
    }
}

//
// Function: main
//
// Description:
//    Main thread of execution. Initialize Winsock, parse
//    command line arguments, create a socket, connect to
//    server, and then send and receive data
//
int main(int argc, char **argv)
{
    WSADATA       wsd;
    SOCKET        sClient;
    char          szBuffer[DEFAULT_BUFFER];
    int           ret,
                  i;
    struct sockaddr_in server;
    struct hostent    *host = NULL;

    // Parse the command line, and load Winsock
    //
    ValidateArgs(argc, argv);
    if (WSAStartup(MAKEWORD(2,2), &wsd) != 0)
    {
        printf("Failed to load Winsock library!\n");
        return 1;
    }
    strcpy(szMessage, DEFAULT_MESSAGE);
    //
    // Create the socket, and attempt to connect to the se
    //
    sClient = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
    if (sClient == INVALID_SOCKET)
    {
```

```
    printf("socket() failed: %d\n", WSAGetLastError())
    return 1;
}
server.sin_family = AF_INET;
server.sin_port = htons(iPort);
server.sin_addr.s_addr = inet_addr(szServer);
//
// If the supplied server address wasn't in the form
// "aaa.bbb.ccc.ddd," it's a host name, so try to reso
//
if (server.sin_addr.s_addr == INADDR_NONE)
{
    host = gethostbyname(szServer);
    if (host == NULL)
    {
        printf("Unable to resolve server: %s\n", szSer
        return 1;
    }
    CopyMemory(&server.sin_addr, host->h_addr_list[0],
        host->h_length);
}
if (connect(sClient, (struct sockaddr *)&server,
    sizeof(server)) == SOCKET_ERROR)
{
    printf("connect() failed: %d\n", WSAGetLastError()
    return 1;
}
// Send and receive data
//
 for(i = 0; i < dwCount; i++)
{
    ret = send(sClient, szMessage, strlen(szMessage),
    if (ret == 0)
        break;
    else if (ret == SOCKET_ERROR)
    {
        printf("send() failed: %d\n", WSAGetLastError(
        break;
    }
    printf("Send %d bytes\n", ret);
    if (!bSendOnly)
    {
        ret = recv(sClient, szBuffer, DEFAULT_BUFFER,
        if (ret == 0)        // Graceful close
            break;
        else if (ret == SOCKET_ERROR)
```

```
            {
                printf("recv() failed: %d\n", WSAGetLastEr
                break;
            }
            szBuffer[ret] = '\0';
            printf("RECV [%d bytes]: '%s'\n", ret, szBuffe
        }
    }
    closesocket(sClient);

    WSACleanup();
    return 0;
}
```

**Figure 7-4. Echo client code**

# Connectionless Protocols

Connectionless protocols behave differently than connection-oriented protocols, so the method for sending and receiving data is substantially different. First we'll discuss the receiver (or server, if you prefer) because the connectionless receiver requires little change when compared with the session-oriented servers. Following that we'll look at the sender.

## Receiver

For a process to receive data on a connectionless socket, the steps are simple. First create the socket with either *socket* or *WSASocket*. Next bind the socket to the interface on which you wish to receive data. This is done with the *bind* function (exactly like the session-oriented example). The difference with connectionless sockets is that you do not call *listen* or *accept*. Instead, you simply wait to receive the incoming data. Because there is no connection, the receiving socket can receive datagrams originating from any machine on the network. The simplest of the receive functions is *recvfrom*, which is defined as

```
int recvfrom(
    SOCKET s,
    char FAR* buf,
    int len,
    int flags,
```

```
    struct sockaddr FAR* from,
    int FAR* fromlen
);
```

The first four parameters are the same as *recv*, including the possible values for *flags—MSG_OOB* and *MSG_PEEK*. The same warnings for using the *MSG_PEEK* flag also apply to connectionless sockets. The *from* parameter is a *SOCKADDR* structure for the given protocol of the listening socket, with *fromlen* pointing to the size of the address structure. When the API call returns with data, the *SOCKADDR* structure is filled with the address of the workstation that sent the data.

The Winsock 2 version of the *recvfrom* function is *WSARecvFrom*. The prototype for this function is

```
int WSARecvFrom(
    SOCKET s,
    LPWSABUF lpBuffers,
    DWORD dwBufferCount,
    LPDWORD lpNumberOfBytesRecvd,
    LPDWORD lpFlags,
    struct sockaddr FAR * lpFrom,
    LPINT lpFromlen,
    LPWSAOVERLAPPED lpOverlapped,
    LPWSAOVERLAPPED_COMPLETION_ROUTINE lpCompletionROUTINE
);
```

The difference is the use of *WSABUF* structures for receiving the data. You can supply one or more *WSABUF* buffers to *WSARecvFrom* with *dwBufferCount* indicating this. By supplying multiple buffers, scatter-gather I/O is possible. The total number of bytes read is returned in *lpNumberOfBytesRecvd*. When you call *WSARecvFrom*, the *lpFlags* parameter can be 0 for no options, *MSG_OOB*, *MSG_PEEK*, or *MSG_PARTIAL*. These flags can be ORed together. If *MSG_PARTIAL* is specified when the function is called, the provider knows to return data even if only a partial message has been received. Upon return, the flag *MSG_PARTIAL* is set if only a partial message was received. Upon return, *WSARecvFrom* will set the *lpFrom* parameter (a pointer to a *SOCKADDR* structure) to the address of the sending machine. Again, *lpFromLen* points to the size of the *SOCKADDR* structure, except that in this function it is a pointer to a *DWORD*. The last two parameters, *lpOverlapped* and *lpCompletionROUTINE*, are used for overlapped I/O (which we'll

discuss in the next chapter).

Another method of receiving (and sending) data on a connectionless socket is to establish a connection. This might sound strange, but it's not quite what it sounds like. Once a connectionless socket is created, you can call *connect* or *WSAConnect* with the *SOCKADDR* parameter set to the address of the remote machine to communicate with. No actual connection is made, however. The socket address passed into a connect function is associated with the socket so that *recv* and *WSARecv* can be used instead of *recvfrom* or *WSARecvFrom* because the data's origin is known. The ability to connect a datagram socket is handy if you intend to communicate with only one endpoint at a time in your application.

## Sender

To send data on a connectionless socket, there are two options. The first, and simplest, is to create a socket and call either *sendto* or *WSASendTo*. We'll cover *sendto* first, which is defined as

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

The parameters are the same as *recvfrom* except that *buf* is the buffer of data to send and *len* indicates how many bytes to send. Also, the *to* parameter is a pointer to a *SOCKADDR* structure with the destination address of the workstation to receive the data. The Winsock 2 function *WSASendTo* can also be used. This function is defined as

```
int WSASendTo(
    SOCKET s,
    LPWSABUF lpBuffers,
    DWORD dwBufferCount,
    LPDWORD lpNumberOfBytesSent,
    DWORD dwFlags,
    const struct sockaddr FAR * lpTo,
    int iToLen,
```

```
    LPWSAOVERLAPPED lpOverlapped,
    LPWSAOVERLAPPED_COMPLETION_ROUTINE lpCompletionROUTINE
);
```

Again, *WSASendTo* is similar to its ancestor. This function takes a pointer to one or more *WSABUF* structures with data to send to the recipient as the *lpBuffers* parameter, with *dwBufferCount* indicating how many structures are present. You can send multiple *WSABUF* structures to enable scatter-gather I/O. Before returning, *WSASendTo* sets the fourth parameter, *lpNumberOfBytesSent*, to the number of bytes actually sent to the receiver. The *lpTo* parameter is a *SOCKADDR* structure for the given protocol, with the recipient's address. The *iToLen* parameter is the length of the *SOCKADDR* structure. The last two parameters, *lpOverlapped* and *lpCompletionROUTINE*, are used for overlapped I/O (discussed in Chapter 8).

As with receiving data, a connectionless socket can be connected to an endpoint address and data can be sent with *send* and *WSASend*. Once this association is established, you cannot go back to using *sendto* or *WSASendTo* with an address other than the address passed to one of the connect functions. If you do attempt to send data to a different address, the call will fail with *WSAEISCONN*. The only way to disassociate the socket handle from that destination is to call *closesocket* on the handle and create a new socket.

## Message-Based Protocols

Just as most connection-oriented protocols are also streaming, connectionless protocols are almost always message-based. Thus, there are some considerations when you're sending and receiving data. First, because message-based protocols preserve data boundaries, data submitted to a send function blocks until completed. For asynchronous or nonblocking I/O modes, if a send cannot be completely satisfied, the send function returns with the error *WSAEWOULDBLOCK*. This means that the underlying system was not able to process that data and you should attempt the send call again at a later time. This scenario will be discussed in greater detail in the next chapter. The main thing to remember is that with message-based protocols, the write can occur only as an autonomous action.

On the flip side, a call to a receive function must supply a

sufficiently large buffer. If the supplied buffer is not large enough, the receive call fails with the error *WSAEMSGSIZE*. If this occurs, the buffer is filled to its capacity, but the remaining data is discarded. The truncated data cannot be retrieved. The only exception is for protocols that do support partial messages, such as the AppleTalk PAP protocol. Prior to returning, the *WSARecvEx* function sets its in-out *flag* parameter to *MSG_PARTIAL* when it receives only part of a message.

For datagrams based on protocols supporting partial messages, consider using one of the *WSARecv* functions. When you make a call to *recv*, there is no notification that the data read is only a partial message. It is up to the programmer to implement a method for the receiver to determine whether the entire message has been read. Subsequent calls to *recv* return other pieces of the datagram. Because of this limitation, it can be convenient to use the *WSARecvEx* function, which allows the setting and reading of the *MSG_PARTIAL* flag to indicate whether the entire message was read. The Winsock 2 functions *WSARecv* and *WSARecvFrom* also support this flag. See the descriptions for *WSARecv*, *WSARecvEx*, and *WSARecvFrom* for additional information about this flag.

Finally, let's take a look at one of the more frequently asked questions concerning sending UDP/IP messages on machines with multiple network interfaces: what happens when a UDP socket is bound explicitly to a local IP interface and datagrams are sent? With UDP sockets, you don't really bind to the network interface; you create an association whereby the IP interface that is bound becomes the source IP address of UDP datagrams sent. The routing table actually determines which physical interface the datagram is transmitted on. If you do not call *bind* but instead either use *sendto*/*WSASendTo* or perform a connect first, the network stack automatically picks the best local IP address based on the routing table. What this means is that if you explicitly bind first, the source IP address could be incorrect. That is, the source IP might not be the IP address of the interface on which the datagram was actually sent.

## Releasing Socket Resources

Because there is no connection with connectionless protocols, there is no formal shutdown or graceful closing of the connection. When the sender or the receiver is finished sending or receiving data, it

simply calls the *closesocket* function on the socket handle. This releases any associated resources allocated to the socket.

## Putting It All Together

Now that we have covered the necessary steps for sending and receiving data on a connectionless socket, let's look at some actual code that performs this procedure. Figure 7-5 shows the first example, a connectionless receiver. The code illustrates how to receive a datagram on either the default interface or a specified local interface.

```
// Module Name: Receiver.c
//
// Description:
//    This sample receives UDP datagrams by binding to the
//    interface and port number and then blocking on a rec
//    call
//
// Compile:
//    cl -o Receiver Receiver.c ws2_32.lib
//
// Command Line Options:
//    sender [-p:int] [-i:IP][-n:x] [-b:x]
//           -p:int   Local port
//           -i:IP    Local IP address to listen on
//           -n:x     Number of times to send message
//           -b:x     Size of buffer to send
//
#include <winsock2.h>
#include <stdio.h>
#include <stdlib.h>

#define DEFAULT_PORT            5150
#define DEFAULT_COUNT           25
#define DEFAULT_BUFFER_LENGTH   4096

int   iPort    = DEFAULT_PORT;          // Port to receive
DWORD dwCount  = DEFAULT_COUNT,         // Number of messa
      dwLength = DEFAULT_BUFFER_LENGTH; // Length of recei
BOOL  bInterface = FALSE;               // Use an interfac
                                        // default
char  szInterface[32];            // Interface to read dat

//
```

```
// Function: usage:
//
// Description:
//    Print usage information and exit
//
void usage()
{
    printf("usage: sender [-p:int] [-i:IP][-n:x] [-b:x]\n\
    printf("       -p:int   Local port\n");
    printf("       -i:IP    Local IP address to listen on\
    printf("       -n:x     Number of times to send messag
    printf("       -b:x     Size of buffer to send\n\n");
    ExitProcess(1);
}

//
// Function: ValidateArgs
//
// Description:
//    Parse the command line arguments, and set some globa
//    indicate what actions to perform
//
void ValidateArgs(int argc, char **argv)
{
    int               i;

    for(i = 1; i < argc; i++)
     {
        if ((argv[i][0] == '-') || (argv[i][0] == '/'))
        {
            switch (tolower(argv[i][1]))
            {
                case 'p':   // Local port
                    if (strlen(argv[i]) > 3)
                        iPort = atoi(&argv[i][3]);
                    break;
                case 'n':   // Number of times to receive
                    if (strlen(argv[i]) > 3)
                        dwCount = atol(&argv[i][3]);
                    break;
                case 'b':   // Buffer size
                    if (strlen(argv[i]) > 3)
                        dwLength = atol(&argv[i][3]);
                    break;
                case 'i':    // Interface to receive datag
                    if (strlen(argv[i]) > 3)
```

```
                    {
                        bInterface = TRUE;
                    strcpy(szInterface, &argv[i][3]);
                    }
                    break;
                default:
                    usage();
                    break;
            }
        }
    }
}

//
// Function: main
//
// Description:
//    Main thread of execution. Initialize Winsock, parse
//    line arguments, create a socket, bind it to a local
//    and port, and then read datagrams.
//
int main(int argc, char **argv)
{
    WSADATA        wsd;
    SOCKET         s;
    char          *recvbuf = NULL;
    int            ret,
                   i;
    DWORD          dwSenderSize;
    SOCKADDR_IN    sender,
                   local;

    // Parse arguments and load Winsock
    //
    ValidateArgs(argc, argv);

    if (WSAStartup(MAKEWORD(2,2), &wsd) != 0)
    {
        printf("WSAStartup failed!\n");
        return 1;
    }
    // Create the socket, and bind it to a local interface
    //
    s = socket(AF_INET, SOCK_DGRAM, 0);
    if (s == INVALID_SOCKET)
    {
```

```
    printf("socket() failed; %d\n", WSAGetLastError())
    return 1;
}
local.sin_family = AF_INET;
local.sin_port = htons((short)iPort);
if (bInterface)
    local.sin_addr.s_addr = inet_addr(szInterface);
else
    local.sin_addr.s_addr = htonl(INADDR_ANY);
if (bind(s, (SOCKADDR *)&local, sizeof(local)) == SOCK
{
    printf("bind() failed: %d\n", WSAGetLastError());
    return 1;
}
// Allocate the receive buffer
//
recvbuf = GlobalAlloc(GMEM_FIXED, dwLength);
if (!recvbuf)
{
    printf("GlobalAlloc() failed: %d\n", GetLastError(
    return 1;
}
// Read the datagrams
//
for(i = 0; i < dwCount; i++)
{
    dwSenderSize = sizeof(sender);
    ret = recvfrom(s, recvbuf, dwLength, 0,
            (SOCKADDR *)&sender, &dwSenderSize);
    if (ret == SOCKET_ERROR)
     {
        printf("recvfrom() failed; %d\n", WSAGetLastEr
        break;
    }
    else if (ret == 0)
        break;
    else
    {
        recvbuf[ret] = '\0';
        printf("[%s] sent me: '%s'\n",
            inet_ntoa(sender.sin_addr), recvbuf);
    }
}
closesocket(s);

GlobalFree(recvbuf);
```

```
    WSACleanup();
    return 0;
}
```

**Figure 7-5. Connectionless receiver**

Receiving a datagram is easy. First you create a socket, and then you bind the socket to the local interface. If you bind to the default interface, you can find its IP address by using the *getsockname* function. This function simply returns the *SOCKADDR_IN* structure associated with the given socket passed to it, indicating the interface on which the socket is bound. After that, it's just a matter of making calls to *recvfrom* in order to read the incoming data. Note that we're using *recvfrom* because we are not concerned with partial messages; the UDP protocol does not support partial messages. In fact, when the TCP/IP stack receives pieces of a larger datagram message, it waits until all parts have been assembled. If the pieces are out of order or one or more pieces are missing, the stack discards the whole message.

Figure 7-6 provides the code for the next example, a connectionless sender. The sender example has quite a few more options than the receiver does. The necessary parameters are the IP address and port of the remote recipient. The -c option specifies whether to make a call to *connect* first; the default behavior is not to make that call. Again, the steps are simple. First create the socket. If the -c option is present, make a call to *connect* with the remote recipient's address and port number. This is followed by calls to *send*. If no connect is performed, simply start sending data to the recipient after socket creation with the *sendto* function.

```
// Module Name: Sender.c
//
// Description:
//    This sample sends UDP datagrams to the specified rec
//    The -c option first calls connect() to associate the
//    recipient's IP address with the socket handle so tha
//    send() function can be used as opposed to the sendto
//
// Compile:
//    cl -o Sender Sender.c ws2_32.lib
//
// Command line options:
//    sender [-p:int] [-r:IP] [-c] [-n:x] [-b:x] [-d:c]
```

```
//           -p:int   Remote port
//           -r:IP    Recipient's IP address or host name
//           -c       Connect to remote IP first
//           -n:x     Number of times to send message
//           -b:x     Size of buffer to send
//           -d:c     Character to fill buffer with
//
#include <winsock2.h>
#include <stdio.h>
#include <stdlib.h>

#define DEFAULT_PORT            5150
#define DEFAULT_COUNT           25
#define DEFAULT_CHAR            'a'
#define DEFAULT_BUFFER_LENGTH   64

BOOL  bConnect = FALSE;                 // Connect to reci
int   iPort    = DEFAULT_PORT;          // Port to send da
char  cChar    = DEFAULT_CHAR;          // Character to fi
DWORD dwCount  = DEFAULT_COUNT,         // Number of messa
      dwLength = DEFAULT_BUFFER_LENGTH; // Length of buffe
char  szRecipient[128];                 // Recipient's IP

//
// Function: usage
//
// Description:
//    Print usage information and exit
//
void usage()
{
    printf("usage: sender [-p:int] [-r:IP] "
           "[-c] [-n:x] [-b:x] [-d:c]\n\n");
    printf("       -p:int   Remote port\n");
    printf("       -r:IP    Recipient's IP address or host
    printf("       -c       Connect to remote IP first\n")
    printf("       -n:x     Number of times to send messag
    printf("       -b:x     Size of buffer to send\n");
    printf("       -d:c     Character to fill buffer with\
    ExitProcess(1);
}

 //
// Function: ValidateArgs
//
// Description:
```

```
//    Parse the command line arguments, and set some globa
//    indicate what actions to perform
//
void ValidateArgs(int argc, char **argv)
{
    int i;

    for(i = 1; i < argc; i++)
    {
        if ((argv[i][0] == '-') || (argv[i][0] == '/'))
        {
            switch (tolower(argv[i][1]))
            {
                case 'p':        // Remote port
                    if (strlen(argv[i]) > 3)
                        iPort = atoi(&argv[i][3]);
                    break;
                case 'r':        // Recipient's IP addr
                    if (strlen(argv[i]) > 3)
                        strcpy(szRecipient, &argv[i][3]);
                    break;
                case 'c':        // Connect to recipient's
                    bConnect = TRUE;
                    break;
                case 'n':        // Number of times to sen
                    if (strlen(argv[i]) > 3)
                        dwCount = atol(&argv[i][3]);
                    break;
                case 'b':        // Buffer size
                    if (strlen(argv[i]) > 3)
                        dwLength = atol(&argv[i][3]);
                    break;
                case 'd':        // Character to fill buff
                    cChar = argv[i][3];
                    break;
                default:
                    usage();
                    break;
            }
        }
    }
}

//
// Function: main
//
```

```
// Description:
//    Main thread of execution. Initialize Winsock, parse
//    line arguments, create a socket, connect to the remo
//    address if specified, and then send datagram message
//    recipient.
//
int main(int argc, char **argv)
{
    WSADATA        wsd;
    SOCKET         s;
    char          *sendbuf = NULL;
    int            ret,
                   i;
    SOCKADDR_IN    recipient;

    // Parse the command line and load Winsock
    //
    ValidateArgs(argc, argv);

    if (WSAStartup(MAKEWORD(2, 2), &wsd) != 0)
    {
        printf("WSAStartup failed!\n");
        return 1;
    }
    // Create the socket
    //
    s = socket(AF_INET, SOCK_DGRAM, 0);
    if (s == INVALID_SOCKET)
    {
        printf("socket() failed; %d\n", WSAGetLastError())
        return 1;
    }
    // Resolve the recipient's IP address or host name
    //
    recipient.sin_family = AF_INET;
    recipient.sin_port = htons((short)iPort);
    if ((recipient.sin_addr.s_addr = inet_addr(szRecipient
        == INADDR_NONE)
    {
        struct hostent *host=NULL;

        host = gethostbyname(szRecipient);
        if (host)
            CopyMemory(&recipient.sin_addr, host->h_addr_l
                host->h_length);
        else
```

```
     {
        printf("gethostbyname() failed: %d\n", WSAGetL
        WSACleanup();
        return 1;
    }
}
// Allocate the send buffer
//
sendbuf = GlobalAlloc(GMEM_FIXED, dwLength);
if (!sendbuf)
{
    printf("GlobalAlloc() failed: %d\n", GetLastError(
    return 1;
}
memset(sendbuf, cChar, dwLength);
//
// If the connect option is set, "connect" to the reci
// and send the data with the send() function
//
if (bConnect)
{
    if (connect(s, (SOCKADDR *)&recipient,
            sizeof(recipient)) == SOCKET_ERROR)
    {
        printf("connect() failed: %d\n", WSAGetLastErr
        GlobalFree(sendbuf);
        WSACleanup();
        return 1;
    }
    for(i = 0; i < dwCount; i++)
    {
        ret = send(s, sendbuf, dwLength, 0);
        if (ret == SOCKET_ERROR)
        {
            printf("send() failed: %d\n", WSAGetLastEr
            break;
        }
        else if (ret == 0)
            break;
        // Send() succeeded!
    }
}
else
{
    // Otherwise, use the sendto() function
    //
```

```
        for(i = 0; i < dwCount; i++)
        {
            ret = sendto(s, sendbuf, dwLength, 0,
                    (SOCKADDR *)&recipient, sizeof(recipie
            if (ret == SOCKET_ERROR)
            {
                printf("sendto() failed; %d\n", WSAGetLast
                break;
            }
            else if (ret == 0)
                break;
            // sendto() succeeded!
        }
    }
    closesocket(s);

    GlobalFree(sendbuf);
    WSACleanup();
    return 0;
}
```

**Figure 7-6. Connectionless sender**

# Miscellaneous API Functions

In this section, we'll cover a few Winsock API functions that you might find useful when you put together your own network applications.

### *getpeername*

This function is used to obtain the peer's socket address information on a connected socket. The function is defined as

```
int getpeername(
    SOCKET s,
    struct sockaddr FAR* name,
    int FAR* namelen
);
```

The first parameter is the socket for the connection, while the last two parameters are a pointer to a *SOCKADDR* structure of the underlying protocol type and its length. For datagram sockets, this function returns the address passed to a connect call; however, it

will not return the address passed to a *sendto* or *WSASendTo* call.

### *getsockname*

This function is the opposite of *getpeername*. It returns the address information for the local interface of a given socket. The function is defined as follows:

```
int getsockname(
    SOCKET s,
    struct sockaddr FAR* name,
    int FAR* namelen
);
```

The parameters are the same as the *getpeername* parameters except that the address information returned for socket *s* is the local address information. In the case of TCP, the address is the same as the server socket listening on a specific port and IP interface.

### *WSADuplicateSocket*

The *WSADuplicateSocket* function is used to create a *WSAPROTOCOL_INFO* structure that can be passed to another process, thus enabling the other process to open a handle to the same underlying socket so that it too can perform operations on that resource. Note that this is only necessary between processes; threads in the same process can freely pass the socket descriptors. This function is defined as

```
int WSADuplicateSocket(
    SOCKET s,
    DWORD dwProcessId,
    LPWSAPROTOCOL_INFO lpProtocolInfo
);
```

The first parameter is the socket handle to duplicate. The second parameter, *dwProcessId*, is the process ID of the process that intends to use the duplicated socket. Third, the *lpProtocolInfo* parameter is a pointer to a *WSAPROTOCOL_INFO* structure that will contain the necessary information for the target process to open a duplicate handle. Some form of interprocess communication must occur so that the current process can pass the *WSAPROTOCOL_INFO* structure to the target process, which then uses this structure to create a handle to the socket (using the

*WSASocket* function).

The descriptors in both sockets can be used independently for I/O; however, Winsock provides no access control, so it is up to the programmer to enforce some kind of synchronization. All of the state information associated with a socket is held in common across all the descriptors because the socket descriptors are duplicated, not the actual socket. For example, any socket option set by the *setsockopt* function on one of the descriptors is subsequently visible using the *getsockopt* function from any or all descriptors. If a process calls *closesocket* on a duplicated socket, it causes the descriptor in that process to become deallocated; however, the underlying socket will remain open until *closesocket* is called on the last remaining descriptor.

Additionally, be aware of some issues with notification on shared sockets when using *WSAAsyncSelect* and *WSAEventSelect*. These two functions are used in asynchronous I/O (discussed in Chapter 8). Issuing either of these calls using any of the shared descriptors cancels any previous event registration for the socket, regardless of which descriptor was used to make that registration. Thus, for example, a shared socket cannot deliver *FD_READ* events to process A and *FD_WRITE* events to process B. If you require event notifications on both descriptors, you should rethink the design of your application to use threads as opposed to processes.

### *TransmitFile*

*TransmitFile* is a Microsoft-specific Winsock extension that allows for high-performance data transfers from a file. This is efficient is because the entire data transfer can occur in kernel mode. That is, if your application reads a chunk of data from the file and then uses *send* or *WSASend,* there are multiple send calls that involve user-mode-to-kernel-mode transitions. With *TransmitFile,* the entire read and send process is performed in kernel mode. The function is defined as

```
BOOL TransmitFile(
    SOCKET hSocket,
    HANDLE hFile,
    DWORD nNumberOfBytesToWrite,
    DWORD nNumberOfBytesPerSend,
    LPOVERLAPPED lpOverlapped,
    LPTRANSMIT_FILE_BUFFERS lpTransmitBuffers,
```

```
    DWORD dwFlags
);
```

The *hSocket* parameter identifies the connected socket on which to transfer the file. The *hFile* parameter is a handle to an opened file. (This is the file that will be sent.) The *nNumberOfBytesToWrite* indicates how many bytes to write from the file. Passing 0 indicates the entire file should be sent. The *nNumberOfBytesPerSend* parameter indicates the send size to use for write operations. For example, specifying 2048 causes *TransmitFile* to send the given file on the socket in 2-KB chunks. Passing 0 indicates using the default send size. The *lpOverlapped* parameter specifies an *OVERLAPPED* structure that is used in overlapped I/O. (See Chapter 8 for information on overlapped I/O.)

The next parameter, *lpTransmitBuffers,* is a *TRANSMIT_FILE_BUFFERS* structure that contains data to be sent before and after the file transfer. The structure is defined as

```
typedef struct _TRANSMIT_FILE_BUFFERS {
    PVOID Head;
    DWORD HeadLength;
    PVOID Tail;
    DWORD TailLength;
} TRANSMIT_FILE_BUFFERS;
```

The *Head* field is a pointer to the data to send before transmitting the file. *HeadLength* indicates the amount of data to send beforehand. The *Tail* field points to the data to send after the file is transmitted. *TailLength* is the number of bytes to send afterward.

The last parameter of *TransmitFile*, *dwFlags*, is used to specify flags to affect the behavior of *TransmitFile*. Table 7-2 contains the flags and their explanations.

**Table 7-2. TransmitFile flags**

| Flag | Description |
|---|---|
| *TF_DISCONNECT* | Initiates socket closure after data has been sent. |
| *TF_REUSE_SOCKET* | Allows the socket handle to be reused in *AcceptEx* as a client |

| | |
|---|---|
| | socket. |
| *TF_USE_DEFAULT_WORKER* | Indicates that the transfer should take place in the context of the system's default thread. This is useful for long file transfers. |
| *TF_USE_SYSTEM_THREAD* | Indicates that the transfer should take place in the context of the system thread. This is also useful for long file transfers. |
| *TF_USE_KERNEL_APC* | Indicates that kernel Asynchronous Procedure Calls (APC) should process the file transfer. This can offer a significant performance increase if reading the file into the cache requires only one read. |
| *TF_WRITE_BEHIND* | Indicates that *TransmitFile* should complete without having all the data acknowledged by the remote system. |

## Windows CE

All the information in the preceding sections applies equally to Windows CE. The only exception is that because Windows CE is based on the Winsock 1.1 specification, none of the Winsock 2–specific functions—such as *WSA* variants of the sending, receiving, connecting, and accepting functions—is available. The only *WSA* functions available on Windows CE are *WSAStartup*, *WSACleanup*, *WSAGetLastError*, and *WSAIoctl*. We have already discussed the first three of these functions; the last will be covered in Chapter 9.

Windows CE supports the TCP/IP protocol, which means you have access to both TCP and UDP. In addition to TCP/IP, infrared sockets are also supported. The IrDA protocol supports only stream-oriented

communication. For both protocols, you make all the usual Winsock 1.1 API calls for creating and transmitting data. The only exception has to do with a bug in UDP datagram sockets in Windows CE 2.0: every call to *send* or *sendto* causes a kernel memory leak. This bug was fixed in Windows CE 2.1, but because the kernel is distributed in ROM, no software updates can be distributed to fix the problem with Windows CE 2.0. The only solution is to avoid using datagrams in Windows CE 2.0.

Because Windows CE does not support console applications and uses UNICODE only, the examples presented in this chapter are targeted to Windows 95 and Windows 98, Windows NT, and Windows 2000. The purpose of our examples is to teach the core concepts of Winsock without having to trudge through code that doesn't relate to Winsock. Unless you're writing a service for Windows CE, a user interface is almost always required. This entails writing many additional functions for window handlers and other user-interface elements, which can obfuscate what we're trying to teach. Additionally, there is the dilemma of UNICODE vs. the non-UNICODE Winsock functions. It is up to the programmer to decide whether the strings passed to the sending and receiving Winsock functions are UNICODE or ANSI strings. Winsock doesn't care what you pass as long it's a valid buffer. (Of course, you might need to typecast the buffer to silence the compiler warnings.) Don't forget that if you cast a UNICODE string to *char *,* the length parameter for how many bytes to send should be adjusted accordingly! In Windows CE, if you want to display any data sent or received, you must take into account whether it is UNICODE so that it can be displayed, as all the other Win32 functions do require UNICODE strings. In sum, Windows CE requires a great deal more housekeeping to make a simple Winsock application.

If you do want to run these examples on Windows CE, only a few minor modifications are required for the Winsock code to compile. First the header file must be Winsock.h, as opposed to Winsock2.h. *WSAStartup* should load version 1.1 because that is the current version of Winsock in Windows CE. Also, Windows CE does not support console applications; therefore, you must use *WinMain* instead of *main*. Note that this does not mean you are required to incorporate a window into your application; it just means you can't use console text I/O functions such as *printf*.

## Other Address Families

All the Winsock API functions introduced in this chapter are protocol-independent. That is, the usage presented here can easily be applied to the other protocols supported by Win32 platforms. The following sections merely describe the sample client/server code for the other protocol families found on the companion CD-ROM.

## AppleTalk

A single AppleTalk sample is provided to illustrate basic client/server techniques. The sample supports both the AppleTalk PAP and ADSP protocols. The PAP protocol is a message-oriented, connectionless, unreliable protocol similar to UDP, but with two notable exceptions. First it supports partial messages, which means that a call to *WSARecvEx* will possibly return with only part of a datagram message. You must check for the *MSG_PARTIAL* flag on return to see whether additional calls are required to obtain the full message. The second exception is that you must set a socket option specific to the PAP protocol before every read. The option, *SO_PRIME_READ*, which is used with the *setsockopt* function, is discussed in Chapter 9. Take a look at the *Atalk.c* sample on the CD, which illustrates how to check for the *MSG_PARTIAL* flag and how to use the *SO_PRIME_READ* option.

The ADSP protocol is a connection-oriented, streaming, reliable protocol— much like TCP. The basic API calls for AppleTalk remain similar to the ones in the UDP and TCP examples presented in this chapter. The only differences will be specific to name resolution. Remember that for AppleTalk, you must bind to an empty address first before looking up or registering an AppleTalk name. This is discussed in more detail in the AppleTalk addressing section in Chapter 6.

The AppleTalk protocol has one limitation. Support for AppleTalk originated in Winsock 1.1, and when Winsock 2 was developed, it appears that AppleTalk was not fully "hooked" into the new functions. Using any of the *WSASend* or *WSARecv* functions might result in flaky results, such as negative byte count returns. This problem is actually described in the Knowledge Base article Q164565. The only exception is *WSARecvEx*, which is simply a *recv*

call except that the *flags* parameter is in/out and can be queried for the *MSG_PARTIAL* flag upon return.

## IrDA

The infrared protocol is a recent addition that is available on Windows CE, Windows 98, and Windows 2000. It offers only one protocol type, which is a connection-oriented, streaming, reliable protocol. Again, the only major difference in the code is the name resolution, which is significantly different from name resolution in IP. You should be aware of one other difference: because Windows CE supports only the Winsock 1.1 specification, you can use only Winsock 1.1 functions on infrared sockets on a Windows CE platform. On Windows 98 and Windows 2000, you can also use the functions specific to Winsock 2. The sample code uses only Winsock 1.1 functions. Of course, on Windows 98 and Windows 2000 you must load the Winsock 2.2 library or greater, as the support for the *AF_IRDA* address family is not available in earlier versions.

The sample code for infrared sockets is found in the following files: *Ircommon.h*, *Ircommon.c*, *Irclient.c*, and *Irserver.c*. The first two files simply define two common functions, one for sending data and the other for receiving data, which are used by both the client and the server. The client side is detailed in *Irclient.c*, which is straightforward. First all devices in range are enumerated. Then a connection attempt is made to each one with the given service name. The first device to accept the connection request is taken. Subsequently the client sends data and reads it back. On the server side of the equation is the file *Irserver.c*. The server simply creates an infrared socket, binds the specified service name to the socket, and waits for client connections. For each client, a thread is spawned to receive data and send it back to the client.

Note that these examples are written with Windows 98 and Windows 2000 in mind. Like the TCP/IP samples, these examples require only slight modifications to run on Windows CE. Regarding Windows CE, the two main points are that there is no support for console applications, and all functions (other than Winsock) use UNICODE strings.

## NetBIOS

We've presented several Winsock NetBIOS examples. As you learned in Chapter 1, NetBIOS is capable of using several different transports, which is still the case with Winsock. In Chapter 6, you learned how to enumerate the NetBIOS-capable transports and how to create sockets based on any one of these. Each protocol-to-adapter combination has two entries: one *SOCK_DGRAM* and one *SOCK_SEQPACKET* type. These correspond to connectionless datagram and stream sockets that are quite similar to UDP and TCP sockets. Besides name resolution, the NetBIOS Winsock interface is no different from what is presented earlier in this chapter. Remember that a well-written server should listen on all available LANAs and that the client should attempt to connect on all LANAs on its end.

The first examples on the CD are *Wsnbsvr.c* and *Wsnbclnt.c*. These examples use the *SOCK_SEQPACKET* socket type, which for all practical purposes appears to the programmer to be stream-oriented. The server creates a socket for each LANA, which is enumerated with the *WSAEnumProtocols* function, and binds it to the server's well-known name. Once a client connection is made, the server creates a thread to handle the connection. From there, the thread simply reads incoming data and echoes it back to the client. Similarly, the client attempts to connect on all LANAs. After the first connect succeeds, the other sockets are closed. The client then sends data to the server and the server reads it back.

The other example is *Wsnbdgs.c*, which is a datagram, or *SOCK_DGRAM*, example. This example includes the code for both sending and receiving datagram messages. This is a connectionless protocol, so the message sent to the server is sent on all available transports (since you really don't have any idea beforehand which transport or transports will be able to reach the server). Additionally, this example supports unique, group, and broadcast data (all discussed in Chapter 1).

## IPX/SPX

The IPX/SPX example, *Sockspx.c*, illustrates how to use the IPX protocol as well as stream and sequential packet SPXII. This single sample incorporates both the sender and the receiver for all three protocols. The particular protocol used is specified via the -p command line option. The sample is straightforward and easy to follow. The main function parses the command line arguments and

then calls either the *Server* or the *Client* function. For the connection-oriented SPXII protocol, this means that the server binds the socket to the internal network address and waits for client connections while the client attempts to connect to the server that is specified on the command line. Once the connection is established, data is sent and received in the normal fashion.

For the connectionless IPX protocol, the example is even simpler. The server simply binds to the internal network and waits for incoming data by calling the *recvfrom* function. The client sends data to the recipient specified on the command line via the *sendto* function.

Two sections of the example might need a bit of explanation. First is the function *FillIpxAddress*, which is responsible for encoding an ASCII IPX address specified on the command line into a *SOCKADDR_IPX* structure. As you saw in Chapter 6, IPX represents its addresses as hexadecimal strings, which means that each hexadecimal character in the address actually occupies 4 bits within the various address fields of the *SOCKADDR_IPX* structure. *FillIpxAddress* takes the IPX address and calls another function, *AtoH*, which actually performs the conversion.

The second function that needs explanation is *EnumerateAdapters,* which is executed if the -m flag is given on the command line. This function uses the socket option *IPX_MAX_ADAPTER_NUM* to find out how many local IPX addresses are available and then calls the *IPX_ADDRESS* socket option to obtain each address. These socket options and their parameters are discussed in Chapter 9. We use these options because our example uses straight IPX addresses and does not perform any name resolution. Chapter 10 examines the name registration and name resolution that are possible for IPX.

## ATM

The ATM protocol is accessible from Winsock on Windows 98 and Windows 2000. The ATM sample is contained in the files *Wsockatm.c*, *Support.c*, and *Support.h*. The latter two files simply contain support routines used by *Wsockatm.c*, such as local ATM address enumeration and ATM address encoding. ATM addresses are hexadecimal encoded, just like IPX addresses, and we use the same *AtoH* function. We also use the socket ioctl command *SIO_GET_NUMBER_OF_ATM_DEVICES* to get the number of local

ATM interfaces and then use the ioctl command *SIO_GET_ATM_ADDRESS* to retrieve the actual address. These ioctl commands are covered in Chapter 9.

Otherwise, both the client and server sides are implemented within *Wsockatm.c*. Because ATM supports only connection-oriented communication, the sample isn't very long and the majority of the code is given in the *main* function. The server will bind to an explicit local interface and wait for client connections, which are handled in the same thread as the listening socket. This means the server will only be able to service one client at a time. We designed the sample this way on purpose, to keep the code simple. On the other hand, the client calls *connect* with the server's ATM address. Once the connection is established, data is sent on the connection.

A few words of caution when using the ATM protocol: you will notice that after the *WSAAccept* call is made within the server, the address of the client is printed out. However, at the time the server receives the connection request, the client's address is not known. This is because the connection is not fully established when the *accept* function is triggered. This is also true on the client side. When the client makes a call to connect to the server, it will succeed even though the connection has not been fully established. This means that upon completion of a *connect* or an *accept* call, an attempt to send data immediately might silently fail until the connection is fully established. Unfortunately, there is no way for the application to determine at what point the connection becomes valid. Additionally, ATM supports only hard closes. That is, when the application calls *closesocket*, the connection is immediately terminated. For protocols that do not support graceful close, any data pending on the socket at either end is normally discarded at the point *closesocket* is called. This is perfectly acceptable behavior; however, the ATM provider is nice to developers. When data is pending on the socket and one party has closed its socket, Winsock still returns the data queued for receiving on the socket.

# Conclusion

In Chapter 6, you learned how to create a socket for a given protocol and how to resolve a host name for the protocol's address family. In this chapter, we took that knowledge and presented the basic Winsock functions that are required for those connection-

oriented and connectionless protocols. For connection-oriented protocols, you know how to accept a client connection and how to establish a client connection to a server. We covered the semantics for session-oriented data-send operations and data-receive operations. For connectionless protocols, you also learned how to send and receive data. Of course, we presented this information using only one I/O model: blocking sockets. In the next chapter, we will cover the other models available in Winsock that make it such a powerful API.

---