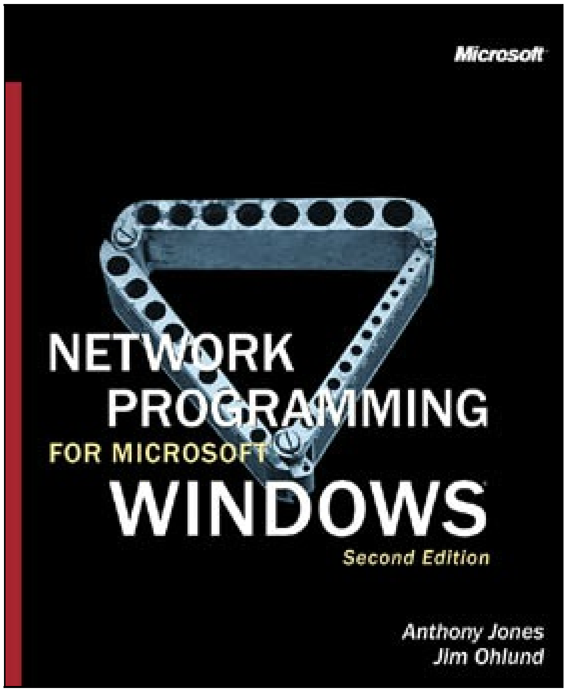

Microsoft
NETWORK PROGRAMMING
FOR MICROSOFT
WINDOWS
Second Edition
Anthony Jones
Jim Ohlund

PUBLISHED BY
Microsoft Press
A Division of Microsoft Corporation
One Microsoft Way
Redmond, Washington 98052-6399

Library of Congress Cataloging-in-Publication Data
Jones, Anthony, 1973-
Network Programming for Microsoft Windows / Anthony Jones, Jim Ohlund.--2nd ed.
p. cm.
Includes index.
ISBN 0-7356-1579-9
1. Internet programming. 2. Microsoft Windows (Computer file) I. Ohlund, Jim, 1966-
II. Title.

QA76.625 .J65 2002
005.2'768--dc21 2001058715

Printed and bound in the United States of America.

1 2 3 4 5 6 7 8 9 QWT 7 6 5 4 3 2

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 *www.microsoft.com/mspress*. Send comments to *mspinput@microsoft.com*.

The example companies, organizations, products, domain names, e-mail addresses, logos, people, places, and events depicted herein are fictitious. No association with any real company, organization, product, domain name, e-mail address, logo, person, place, or event is intended or should be inferred.

Acquisitions Editor: David Clark
Project Editor: Kurt Stephan

Body Part No. X08-68161

Dedication

For my loving wife, Genevieve, thanks for your patience and understanding.

-A.J.

For Samantha

-J.O.

Acknowledgments

In addition to all the people who contributed to the first edition, we would like to thank the following individuals for their generous help in writing this edition. Very special thanks go to Jory Prather for verifying the code samples as well as fixing them for consistency. Thanks to Dave Thaler, Brian Zill, and Rich Draves for clarifying our IPv6 questions, Mohammad Alam and Rajesh Peddibhotla for help with reliable multicasting, and Jeff Venable for his contributions on the Network Location Awareness functionality. Thanks to Vadim Eydelman for his Winsock expertise. And finally we would like to thank the .NET Application Frameworks team (Lance Olson, Mauro Ottaviani, and Ron Alberda) for their help with our questions about .NET Sockets.

Introduction

Welcome to *Network Programming for Microsoft Windows, Second Edition!* The second edition covers the same topics as the first edition and even more as well. This book primarily focuses on the Winsock network programming technology. In particular, we've added a chapter on writing high-performance, scalable Winsock applications and a chapter devoted to Winsock programming in the C# programming language using the exciting new .NET Application Frameworks library. In addition, we've completely updated the chapter on the Windows Service Provider Interface (SPI), and we cover additional protocols (such as IPv6 and reliable multicasting) and reveal functionality that is new to Windows XP.

This book covers a wide variety of networking functions available in Windows 95, Windows 98, Windows Me, Windows NT 4.0, Windows 2000, Windows XP, and Windows CE. The majority of the text covers intermediate and advanced networking topics, but we retooled the Winsock section so that it is more accessible to programmers of all levels.

How to Use This Book

This book covers six technical areas:

- Winsock application programming interface (API)
- .NET Sockets (from C#)
- Visual Basic Winsock Control
- Client Remote Access Server (RAS)
- IP Helper API
- Legacy Networking—NetBIOS and the Windows redirector

The NetBIOS and Windows redirector technologies have not changed since the first edition, and because of space considerations, these chapters are included only with the eBook located on the companion CD-ROM as Chapters 17-22.

In this edition, the majority of the book is dedicated to covering the Winsock API. Chapter 1 starts with an introduction to Winsock and is specifically geared for the beginning Winsock programmer. This chapter covers all the basics and introduces Transmission Control Protocol (TCP) and User Datagram Protocol (UDP) through simple samples, as well as providing a roadmap to advanced Winsock topics covered in other chapters. For the sake of simplicity, Chapter 1 covers the IPv4 protocol.

Chapter 2 discusses the Winsock architecture such as the Winsock catalog, as well as how Winsock fits into the overall network stack. In addition, we cover how to enumerate the Winsock catalog and find characteristics specific to each protocol installed on the system.

Chapter 3 is dedicated to the Internet Protocol (IP). In this chapter we cover both IPv4 and IPv6 and include addressing information and name resolution for each. The last part of this chapter illustrates how to write applications that work seamlessly over either protocol. The remaining protocols accessible from Winsock are covered in Chapter 4. In both chapters, we present simple samples illustrating the basic concepts of each protocol family.

Chapters 5 and 6 cover the input/output (I/O) models Winsock offers for the advanced Winsock programmer. Chapter 5 presents each model and the basics of how to use it, while Chapter 6 goes into detail on how to write high-performance, scalable Winsock applications. In this chapter, we discuss resource management and the merits of the different I/O models as well as performance numbers. Sample code is provided that illustrates each of the I/O models.

Chapter 7 is a reference that covers all the socket options and ioctls that can be accessed from Winsock. These include generic Winsock options as well as protocol-specific ioctls. For each option, we provide the expected input and output parameters necessary for successfully accessing the option. This chapter has been updated to include options specific to new protocols (such as IPv6 and reliable multicasting).

Chapter 8 covers Winsock registration and name resolution and introduces the different name spaces in which queries can be performed, such as Domain Name System (DNS), Service Advertising Protocol (SAP), and the Active Directory directory service. The chapter also discusses the new Network Location Awareness (NLA) namespace, which can provide valuable information about the network you are currently connected to.

Multicasting is the topic of Chapter 9. This chapter covers IPv4 and IPv6 multicasting as well as the reliable multicasting transport new to Windows XP. This chapter also discusses ATM point-to-multipoint communications. Chapter 10 is devoted to Quality of Service (QOS), which is a technology that allows for guaranteeing a portion of the network bandwidth to an application. Chapter 11 moves on to raw IP sockets and discusses how to build your own protocol headers which can be used to communicate directly over IP networks—this includes both IPv4 and IPv6.

Chapter 12 covers the Winsock Service Provider Interface (SPI). This interface is a means by which a programmer can install a layer between Winsock and lower-level service providers such as Transmission Control Protocol/Internet Protocol (TCP/IP) for the purpose of manipulating socket and protocol behavior or name registration and resolution. This is an advanced feature that allows software developers to extend Winsock functionality. The SPI chapter has been completely rewritten and provides fully functioning, robust layered service provider (LSP) sample code.

Chapter 13 covers the .NET Application Framework's Network Socket object. In this chapter, we show how to access the new Socket class from the C# language. Chapter 14 discusses the Microsoft Visual Basic Winsock control. We decided to include this chapter after seeing how many developers rely on Visual Basic and this control. The control is limited in its ability to utilize the advanced features of Winsock, but it is fantastic for Visual Basic developers who require simple, easy-to-use network communication.

Chapter 15 covers the Remote Access Server (RAS) client API. We decided to include a chapter on RAS because of the popularity of the Internet, dial-up communication, and Virtual Private Networking

(VPN) communication. The ability for a programmer to add dial-up capability to a network application is quite useful since it makes the program easier for the user. That is, an end user does not need to know how to set up and establish a dial-up connection to use your network application.

Chapter 16 covers the IP Helper API, which provides useful information about the network configuration on the current computer. This includes a new function that enumerates IPv6 specific information.

Finally, chapters on the legacy technologies (NetBIOS, mailslots, named pipes, and Windows Redirector) from the first edition, as well as NetBIOS command and Winsock error code references, are included in eBook form (as Chapters 17-22) on the companion CD-ROM.

We hope that you will find this book to be a valuable learning and reference tool. We still believe it is the most comprehensive book about Windows network programming available.

How to Use the Companion CD-ROM

In each chapter, we present code examples that demonstrate how to use many of the net-working APIs we describe. These examples are available on the accompanying CD-ROM. To install them, place the CD into your drive and Autorun will launch a starting menu. If the starting menu does not launch automatically, it can be accessed by running StartCD.exe in the disc's root directory. The sample code can be installed by selecting the Install Example Code option on the starting menu, or you can access each example from the CD (under Samples\ChapterXX).

> The CD-ROM requires a computer running a 32-bit Microsoft Windows platform.

About the eBook

The companion CD-ROM also includes an electronic version of the book that you can view using Microsoft Internet Explorer 5.01 or later.

To Use the eBook

1. Insert the companion CD-ROM into your CD-ROM drive.
2. On the starting menu that appears, click eBook and follow the instructions, or select Run from the Start menu and type **D:\eBook\Autorun.exe** (where D is the name of your CD-ROM disk drive). This will install an icon for the eBook on your desktop.
3. Click OK to exit the Installation Wizard.
4. You must have the companion CD-ROM inserted in your CD-ROM drive to run the eBook.

Microsoft Press Support Information

Every effort has been made to ensure the accuracy of the contents of this book and the companion CD-ROM. Microsoft Press provides corrections for this book at http://www.microsoft.com/mspress/support/.

Many of the function definitions and tables in this book were adapted or reprinted here with the generous participation and permission of the Microsoft Platform SDK documentation group. Some material is based on preliminary documentation and is subject to change. For the latest Platform SDK information as well as updates and bug fixes, please visit the MSDN Web site at http://www.microsoft.com/msdownload/platformsdk/sdkupdate/.

If you have comments, questions, or ideas regarding this book or its companion CD, please send them to Microsoft Press via e-mail to:

MSPInput@Microsoft.com

or via postal mail to:

Microsoft Press
Attn: *Network Programming for Microsoft Windows, Second Edition* Editor
One Microsoft Way
Redmond, WA 98052-6399

Please note that product support is not offered through the above address.

Chapter 1

Introduction to Winsock

This chapter is dedicated to learning the basic techniques for writing successful Winsock applications. Winsock is a standard application programming interface (API) that allows two or more applications (or processes) to communicate either on the same machine or across a network and is primarily designed to foster data communication over a network. It is important to understand that Winsock is a network programming interface and not a protocol. Winsock provides the programming interface for applications to communicate using popular network protocols such as Transmission Control Protocol/Internet Protocol (TCP/IP) and Internetwork Packet Exchange (IPX). The Winsock interface inherits a great deal from the BSD Sockets implementation on UNIX platforms. In Windows environments, the interface has evolved into a truly protocol-independent interface, especially with the release of Winsock 2.

In this chapter, we'll look at the fundamentals of setting up communication from one machine on a network to another, along with how to send and receive data. 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 fundamental Winsock calls, the examples presented use straight blocking Winsock calls. Chapter 5 presents non-blocking and other advanced I/O methods available in Winsock, including code examples.

In addition, 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.

Before you begin developing an application using Winsock, you need to understand what files and libraries are required to build your application.

Winsock Headers and Libraries

As mentioned previously, Winsock is available in two major versions—Winsock 1 and Winsock 2—on all Windows platforms except Windows CE (Windows CE has only Winsock 1). When developing new applications you should target the Winsock 2 specification by including WINSOCK2.H in your application. For compatibility with older Winsock applications and when developing on Windows CE platforms, WINSOCK.H is available. There is also an additional header file: MSWSOCK.H, which targets Microsoft-specific programming extensions that are normally used for developing high performance Winsock applications, which will be described in Chapter 6.

When compiling your application with WINSOCK2.H, you should link with WS2_32.LIB library. When using WINSOCK.H (as on Windows CE) you should use WSOCK32.LIB. If you use extension APIs from MSWSOCK.H, you must also link with MSWSOCK.DLL. Once you have included the necessary header files and link environment, you are ready to begin coding your application, which requires initializing Winsock.

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 returns a *SOCKET_ERROR;* 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, * 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, and 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* (see Chapter 2). The maximum number of concurrent sockets isn't some magic number—it depends more on the physical resources available. Finally, the *lpVendorInfo* field is reserved for vendor-specific information regarding the implementation of Winsock. This field is not used on any Windows platforms.

Table 1-1 lists the 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.

Table 1-1 *Supported Winsock*

Table 1-1 *Supported Winsock Versions*

Platform	Winsock Version
Windows 95	1.1 (2.2)
Windows 98	2.2
Windows Me	2.2
Windows NT 4.0	2.2
Windows 2000	2.2
Windows XP	2.2
Windows CE	1.1

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 Winsock stack's behavior 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 the most part, 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.

When your application is completely finished using the Winsock interface, you should call WSACleanup, which allows Winsock to free up any resources allocated by Winsock and cancel any pending Winsock calls that your application made. WSACleanup is defined as

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

Failure to call WSACleanup when your application exits is not harmful because the operating system will free up resources automatically; however, your application will not be following the Winsock specification. Also, you should call WSACleanup for each call that is made to WSAStartup.

Error Checking and Handling

We'll first cover error checking and handling, as they are vital to writing a successful Winsock application. It is actually common for Winsock functions to return an error; however, there are some cases in which 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*. On the flip side of WSAGetLastError, there is WSASetLastError, which allows you to manually set error codes that WSAGetLastError retrieves.

The following program demonstrates how to construct a skeleton Winsock application based on the discussion so far:

```
#include <winsock2.h>

void main(void)
{
   WSADATA wsaData;

   // Initialize Winsock version 2.2

   if ((Ret = WSAStartup(MAKEWORD(2,2), &wsaData)) != 0)
   {
      // NOTE: Since Winsock failed to load we cannot use
      // WSAGetLastError to determine the specific error for
      // why it failed. Instead we can rely on the return
      // status of WSAStartup.

      printf("WSAStartup failed with error %d\n", Ret);
      return;
   }

   // Setup Winsock communication code here

   // When your application is finished call WSACleanup
   if (WSACleanup() == SOCKET_ERROR)
   {
      printf("WSACleanup failed with error %d\n", WSAGetLastError());
   }
}
```

Now we are ready to describe how to set up communication using a network protocol.

Addressing a Protocol

For simplicity's sake, and to avoid repetition, the remaining discussion in this chapter is limited to describing how to make fundamental Winsock calls to set up communication using the Internet Protocol (IP). We chose IP because most Winsock applications developed today use it because it is widely available due to the popularity of the Internet. As we mentioned earlier, Winsock is a protocol-independent interface and specific details for using other protocols, such as IPX, are covered in Chapter 4. Also, our discussion of IP in this chapter is limited to briefly describing IP version 4 (IPv4). Chapter 3 fully describes all IP versions—IPv4 and IP version 6 (IPv6)—in greater detail.

Throughout the remainder of this chapter, we will demonstrate the basics of how to set up Winsock communication using the IPv4 protocol. 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 data delivery. Two higher-level protocols—Transmission Control Protocol (TCP) and User Datagram Protocol (UDP)—are used for connection-oriented and connectionless data communication over IP, which we will describe later. Both TCP and UDP use IP for data transmission and are normally referred to as TCP/IP and UDP/IP. To use IPv4 in Winsock, you need understand how to address IPv4.

Addressing IPv4

In IPv4, computers are assigned an address that is represented as a 32-bit quantity. 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.

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). There are more details about choosing a port in Chapter 2.

The *sin_addr* field of the *SOCKADDR_IN* structure is used for storing an IPv4 address as a four-byte quantity, which is an unsigned long integer data type. Depending on how this field is used, it can represent a local or a remote IP address. IP addresses are normally specified in Internet standard dotted notation as "a.b.c.d." Each letter represents a number (in decimal, octal, or hexadecimal format) 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.

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 to most significant. 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), 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 four-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 two-byte number in host-byte order. The *htons* function returns the number as a two-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 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 address IPv4 by creating a *SOCKADDR_IN* structure using the *inet_addr* and *htons* functions described previously.

```
SOCKADDR_IN InternetAddr;
INT nPortId = 5150;

InternetAddr.sin_family = AF_INET;

// Convert the proposed dotted Internet address 136.149.3.29
// to a four-byte integer, and assign it to sin_addr

InternetAddr.sin_addr.s_addr = inet_addr("136.149.3.29");

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

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

As you can probably tell, IP addresses aren't easy to remember. Most people would much rather refer to a machine (or host) by using an easy-to-remember, user-friendly host name instead of an IP address. Chapter 3 describes useful address and name resolution functions that can help you resolve a host name, such as www.somewebsite.com, to an IP address and a service name, such as FTP, to a port number using functions such as getaddrinfo, getnameinfo, gethostbyaddr, gethostbyname, gethostname, getprotobyname, getprotobynumber, get-servbyname, and getservbyport. There are also some asynchronous versions of some of these functions: WSAAsyncGetHostByAddr, WSAAsyncGetHostByName, WSAAsyncGetProtoByName, WSAAsyncGetProtoByNumber, WSAAsyncGetServBy- Name, and WSAAsyncGetServByPort.

Now that you have the basics of addressing a protocol such as IPv4, you can prepare to set up communication by creating a socket.

Creating a Socket

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 Windows, a socket is not the same thing as a file descriptor and therefore is a separate type: *SOCKET* in WINSOCK2.H. There are two functions that can be used to create a socket: socket and WSASocket. The next three chapters describe socket creation for each of the available protocols in great detail. For simplicity, we will briefly describe socket:

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

The first parameter, *af*, is the protocol's address family. Since we describe Winsock in this chapter using only the IPv4 protocol, you should set this field to AF_INET. The second parameter, *type*, is the protocol's socket type. When you are creating a socket to use TCP/IP, set this field to *SOCK_STREAM*, for UDP/IP use *SOCK_DGRAM*. The third parameter is *protocol* and is used to qualify a specific transport if there are multiple entries for the given address family and socket type. For TCP you should set this field to IPPROTO_TCP; for UDP use IPPROTO_UDP. Chapter 2 describes socket creation in greater detail for all protocols, including the WSASocket API.

Winsock features four useful functions to control various socket options and socket behaviors: setsockopt, getsockopt, ioctlsocket, and WSAIoctl. For simple Winsock programming, you will not need to use them specifically. Chapter 7 describes each of these functions and all the available options. Once you have successfully created a socket, you are ready to set up communication on the socket to prepare it for sending and receiving data. In Winsock there are two basic communication techniques: connection-oriented and connectionless communication.

Connection-Oriented Communication

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

In IP, connection-oriented communication is accomplished through the TCP/IP protocol. 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.

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 create a socket with either the *socket* or *WSASocket* call and bind the 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 *WSAAccept* call. In the next few sections, we will discuss each API call that is required for binding, listening, and accepting a client connection. Figure 1-1 illustrates the basic calls a server and a client must perform in order to establish a communication channel.

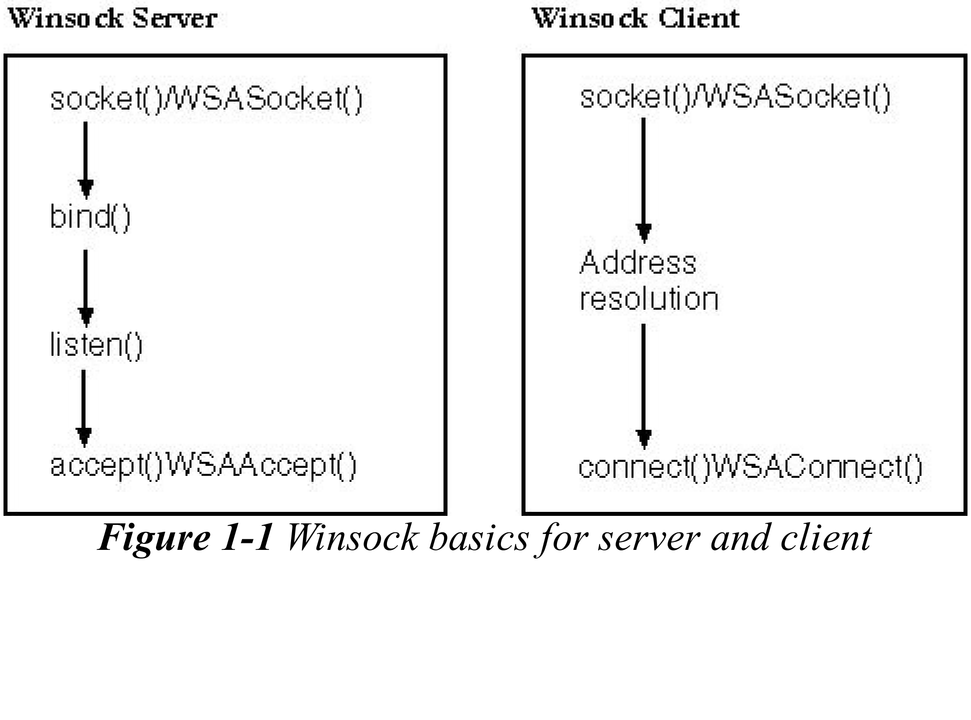


Figure 1-1 *Winsock basics for server and client*

Binding

Once the socket of a particular protocol is created, you must bind it 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;
SOCKADDR_IN           tcpaddr;
int                   port = 5150;

s = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);

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

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

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 by using a special address, INADDR_ANY, and occupies port number 5150. We could have specified an explicit IP address available on the system, but INADDR_ANY allows us to bind to all available interfaces on the system so that any incoming client connection on any interface (but the correct port) will be accepted by our listening socket. The call to *bind* formally establishes this association of the socket with the local IP interface and port.

On error, *bind* returns *SOCKET_ERROR*. The most common error encountered with *bind* is *WSAEADDRINUSE*. With 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.

Listening

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 two. 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 request is removed from the queue so that others can make a request. The *backlog* parameter is silently limited to a value that the underlying protocol provider determines. Illegal values are replaced with their nearest legal values. In addition, 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.

Accepting Connections

Now you're ready to accept client connections. This is accomplished with the *accept*, *WSAAccept,* or *AcceptEx* function. (AcceptEx, an extended version of *accept*, is described in detail in Chapter 6.) 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 IPv4 address information of the client making the connection request, and the *addrlen* parameter indicates the size of the structure. In addition, *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 open to accept other client connections and is still in listening mode.

If an error occurs, *INVALID_SOCKET* is returned. The most common error encountered is *WSAEWOULDBLOCK* if the listening socket is in asynchronous or non-blocking mode and there is no connection to be accepted. Block, non-blocking, and other socket modes are covered in Chapter 5. Winsock 2 introduced the function *WSAAccept*, which has the capability to conditionally accept a connection based on the return value of a condition function. Chapter 10 will describe *WSAAccept* in greater detail.

At this point, we have described all the necessary elements to construct a simple Winsock TCP/IP server application. The following program fragment demonstrates how to write a simple server that can accept one TCP/IP connection. We did not perform any error checking on the calls to make reading the code less confusing. You will find a complete version of this application in a file named TCPSERVER on the companion CD.

```
#include <winsock2.h>

void main(void)
{
   WSADATA              wsaData;
   SOCKET               ListeningSocket;
   SOCKET               NewConnection;
   SOCKADDR_IN          ServerAddr;
   SOCKADDR_IN          ClientAddr;
   int                  Port = 5150;

   // Initialize Winsock version 2.2

   WSAStartup(MAKEWORD(2,2), &wsaData);

      // Create a new socket to listen for client connections.

      ListeningSocket = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);

      // Set up a SOCKADDR_IN structure that will tell bind that we
      // want to listen for connections on all interfaces using port
      // 5150. Notice how we convert the Port variable from host byte
      // order to network byte order.

      ServerAddr.sin_family = AF_INET;
      ServerAddr.sin_port = htons(Port);
      ServerAddr.sin_addr.s_addr = htonl(INADDR_ANY);

      // Associate the address information with the socket using bind.

      bind(ListeningSocket, (SOCKADDR *)&ServerAddr,
      sizeof(ServerAddr));

   // Listen for client connections. We used a backlog of 5, which
   // is normal for many applications.

      listen(ListeningSocket, 5);

   // Accept a new connection when one arrives.

      NewConnection = accept(ListeningSocket, (SOCKADDR *)
                          &ClientAddr,&ClientAddrLen));

   // At this point you can do two things with these sockets. Wait
   // for more connections by calling accept again on ListeningSocket
   // and start sending or receiving data on NewConnection. We will
   // describe how to send and receive data later in the chapter.

   // When you are finished sending and receiving data on the
   // NewConnection socket and are finished accepting new connections
   // on ListeningSocket, you should close the sockets using the
   // closesocket API. We will describe socket closure later in the
   // chapter.

      closesocket(NewConnection);
      closesocket(ListeningSocket);

   // When your application is finished handling the connections,
   // call WSACleanup.

      WSACleanup();
}
```

Now that you understand how to construct a server that can receive a client connection, we will describe how to construct a client.

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.
2. Set up a SOCKADDR address structure with the name of server you are going to connect to (dependent on underlying protocol). For TCP/IP, this is the server's IP address and port number its application is listening on.
3. Initiate the connection with *connect* or *WSAConnect*.

You already know how to create the socket and construct a SOCKADDR structure, so the only remaining step is establishing a connection.

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 affect change in the underlying protocol. In addition, 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-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, it is known as an active socket closure; otherwise, the socket closure is passive. Figure 1-2 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 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.

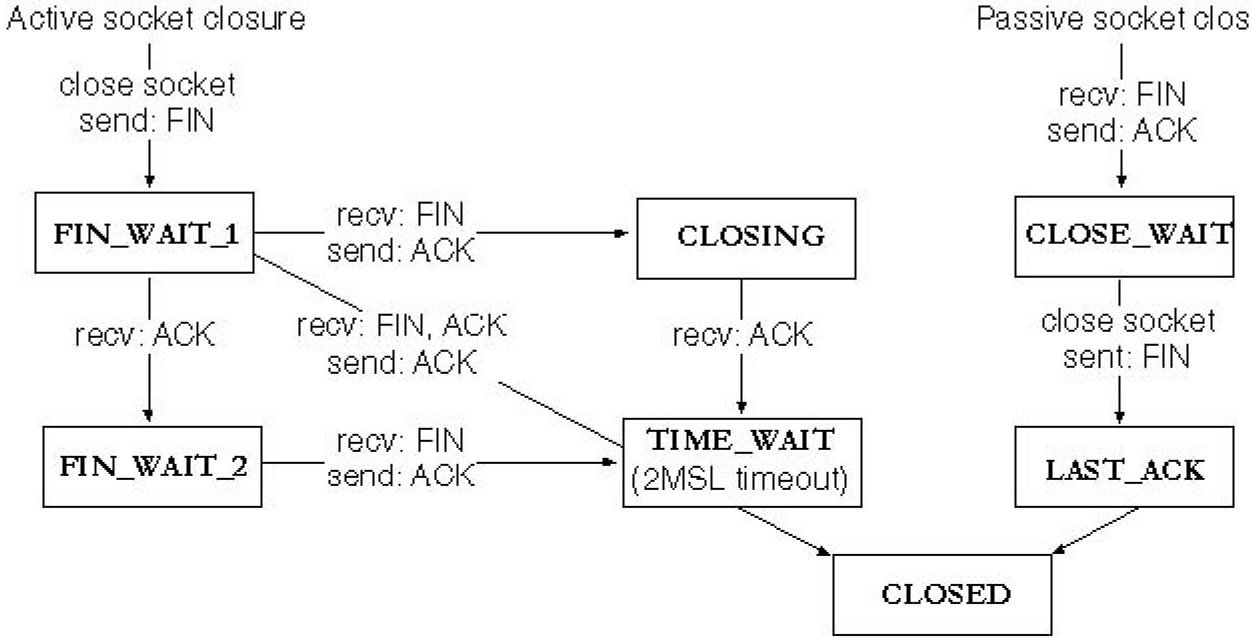


***Figure 1-2** TCP socket closure states*

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 7 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

Connecting a socket is accomplished by calling *connect*, *WSAConnect, or ConnectEx.* 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.

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 *WSAECONNREFUSED* error. 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).

The following program fragment demonstrates how to write a simple client that can connect to the server application described earlier. You will find a complete version of this application in a file called TCPCLIENT on the companion CD.

```
#include <winsock2.h>

void main(void)
{
   WSADATA              wsaData;
   SOCKET               s;
   SOCKADDR_IN          ServerAddr;
   int                  Port = 5150;

   // Initialize Winsock version 2.2

   WSAStartup(MAKEWORD(2,2), &wsaData);

   // Create a new socket to make a client connection.

      s = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);

   // Set up a SOCKADDR_IN structure that will be used to connect
   // to a listening server on port 5150. For demonstration
   // purposes, let's assume our server's IP address is 136.149.3.29.
   // Obviously, you will want to prompt the user for an IP address
   // and fill in this field with the user's data.

      ServerAddr.sin_family = AF_INET;
      ServerAddr.sin_port = htons(Port);
      ServerAddr.sin_addr.s_addr = inet_addr("136.149.3.29");

   // Make a connection to the server with socket s.

      connect(s, (SOCKADDR *) &ServerAddr, sizeof(ServerAddr));

   // At this point you can start sending or receiving data on
   // the socket s. We will describe sending and receiving data
   // later in the chapter.

   // When you are finished sending and receiving data on socket s,
   // you should close the socket using the closesocket API. We will
   // describe socket closure later in the chapter.

      closesocket(s);

   // When your application is finished handling the connection, call
   // WSACleanup.

      WSACleanup();
}
```

Now that you can set up communication for a connection-oriented server and client, you are ready to begin handling data transmission.

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 which is just simple byte-oriented data. In reality, it can be a buffer with any raw data in it—whether it's binary or string data doesn't matter.

In addition, 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 two 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 5, 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 *OR* 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 protocol 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* will be returned. A common error is *WSAECO-NNABORTED,* 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 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 that Winsock supports and is discussed in detail in Chapter 5.

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. There is one final send function you should be aware of: WSASendDisconnect.

WSASendDisconnect

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

```
int WSASendDisconnect (
    SOCKET s,
    LPWSABUF lpOutboundDisconnectData
);
```

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). To check whether pending data contains urgent data, you must call the *ioctlsocket* function with the *SIOCATMARK* option. Chapter 7 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 inlining 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 7, controls the behavior of OOB data.

Telnet and Rlogin use urgent data for several reasons. However, unless you plan to write 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 Windows. 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 (which is described later) with an *SD_SEND* argument, but it also sends the data contained in its lpOut*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, and *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*. In addition, 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 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.

There are some considerations when using *recv* on a message- or datagram-based socket such as UDP, which we will describe later. If 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 such as TCP 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, and *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 *OR* combination of those values. The *MSG_PARTIAL* flag has several different meanings depending on where it is used or encountered. For message-oriented protocols that support partial messaging (like AppleTalk), 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. In addition, not all protocols support partial messages. The protocol entry for each protocol contains a flag indicating whether it supports this feature. See Chapter 2 for more information. The *lpOverlapped* and *lpCompletionRoutine* parameters are used in overlapped I/O operations, discussed in Chapter 5. There is one other specialized receive function you should be aware of: *WSARecvDisconnect*.

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 of *WSASendDisconnect* are the connected socket handle and a valid *WSABUF* structure with the data to be received. The data received can be only disconnect data that is sent by a *WSASendDisconnect* on the other side; it cannot be used to receive normal data. In addition, once the data is received, this function disables reception from the remote party, which is equivalent to calling the *shutdown* function (which is described later) with *SD_RECEIVE*.

Stream Protocols

Because most connection-oriented communication, such as TCP, is streaming protocols, we'll briefly describe them here. A streaming protocol is one that the sender and receiver may break up or coalesce data into smaller or larger groups. 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 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 be buffer space to hold only 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 same principle 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;

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 four 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 four 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 it 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 effects —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. And 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 (because shutdown has no meaning for connectionless protocols). 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.

Note that not all connection-oriented protocols support graceful closure, which is what the shutdown

API performs. For these protocols (such as ATM), only closesocket needs to be called to terminate the session.

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 synchronous 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*. Socket I/O models are discussed in greater depth in Chapter 5. In addition, 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 7 for a complete explanation.

Connectionless Communication

Connectionless communication behaves differently than connection-oriented communication, 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 connection-oriented servers. After that we'll look at the sender.

In IP, connectionless communication is accomplished through UDP/IP. UDP doesn't guarantee reliable data transmission and is capable of sending data to multiple destinations and receiving it from multiple sources. For example, if a client sends data to a server, the data is transmitted immediately regardless of whether the server is ready to receive it. If the server receives data from the client, it doesn't acknowledge the receipt. Data is transmitted using datagrams, which are discrete message packets.

Receiver

The steps in the process of receiving data on a connectionless socket 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 bitwise *OR* 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 store the address of the sending machine in the *lpFrom* parameter (a pointer to a *SOCKADDR* structure). 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 Chapter 5).

Another method of receiving (and sending) data on a connectionless socket is to establish a connection. This might seem 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 *recv* and *WSARecv* can be used instead of *recvfrom* or *WSARecvFrom* because the data's origin is known. The capability to connect a datagram socket is handy if you intend to communicate with only one endpoint at a time in your application.

The following code sample demonstrates how to construct a simple UDP receiver application. You will find a complete version of this application in a file named UDPRECEIVER on the companion CD.

```
#include <winsock2.h>

void main(void)
{
   WSADATA                wsaData;
   SOCKET                 ReceivingSocket;
   SOCKADDR_IN            ReceiverAddr;
   int                    Port = 5150;
   char                   ReceiveBuf[1024];
   int                    BufLength = 1024;
   SOCKADDR_IN            SenderAddr;
   int                    SenderAddrSize = sizeof(SenderAddr);

   // Initialize Winsock version 2.2

   WSAStartup(MAKEWORD(2,2), &wsaData);

      // Create a new socket to receive datagrams on.
      ReceivingSocket = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);

      // Set up a SOCKADDR_IN structure that will tell bind that we
      // want to receive datagrams from all interfaces using port
      // 5150.

      ReceiverAddr.sin_family = AF_INET;
      ReceiverAddr.sin_port = htons(Port);
      ReceiverAddr.sin_addr.s_addr = htonl(INADDR_ANY);

      // Associate the address information with the socket using bind.

      bind(ReceivingSocket, (SOCKADDR *)&SenderAddr, sizeof(SenderAddr));

   // At this point you can receive datagrams on your bound socket.
   recvfrom(ReceivingSocket, ReceiveBuf, BufLength, 0,
            (SOCKADDR *)&SenderAddr, &SenderAddrSize);

   // When your application is finished receiving datagrams close
   // the socket.

   closesocket(ReceivingSocket);

   // When your application is finished call WSACleanup.

   WSACleanup();
}
```

Now that you understand how to construct a receiver that can receive a datagram, we will describe how to construct a sender.

Sender

There are two options to send data on a connectionless socket. 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 5).

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 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 connect with the destination address of *INADDR_ANY*.

The following code sample demonstrates how to construct a simple UDP sender application. You will find a complete version of this application on the companion CD in a file named UDPSENDER.

```
#include <winsock2.h>

void main(void)
{
   WSADATA              wsaData;
   SOCKET               SendingSocket;
   SOCKADDR_IN          ReceiverAddr;
   int                  Port = 5150;
   char                 SendBuf[1024];
   int                  BufLength = 1024;

   // Initialize Winsock version 2.2

   WSAStartup(MAKEWORD(2,2), &wsaData);

   // Create a new socket to receive datagrams on.

   SendingSocket = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);

   // Set up a SOCKADDR_IN structure that will identify who we
   // will send datagrams to. For demonstration purposes, let's
   // assume our receiver's IP address is 136.149.3.29 and waits
   // for datagrams on port 5150.
   ReceiverAddr.sin_family = AF_INET;
   ReceiverAddr.sin_port = htons(Port);
   ReceiverAddr.sin_addr.s_addr = inet_addr("136.149.3.29");

   // Send a datagram to the receiver.

   sendto(SendingSocket, SendBuf, BufLength, 0,
          (SOCKADDR *)&ReceiverAddr, sizeof(RecieverAddr));

   // When your application is finished sending datagrams close
   // the socket.

   closesocket(SendingSocket);

   // When your application is finished call WSACleanup.

   WSACleanup();
}
```

Message-Based Protocols

Just as most connection-oriented communication is also streaming, connectionless communication is 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 non-blocking 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 Chapter 5. The main point to remember is that with message-based protocols, the write can occur as an autonomous action only.

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 WSARecv, WSARecvEx, or WSARecvFrom functions set the 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 because when you make a call to *recv/recvfrom*, 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 if the entire message has been read. Subsequent calls to *recv/recvfrom* 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 if 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 about 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 use either *sendto*/*WSASendTo* or perform a connect first, the network stack automatically picks the best local IP address based on the routing table. So 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.

Miscellaneous APIs

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; 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 necessary only 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).

Both socket descriptors can be used independently for I/O. Winsock provides no access control, however, 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. The underlying socket, however, will remain open until *closesocket* is called on the last remaining descriptor.

In addition, be aware of some issues with notification on shared sockets when using *WSAAsyncSelect* and *WSAEventSelect*. These two functions are discussed in Chapter 5. 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 your application design so that it uses threads instead of processes.

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 with Windows CE are *WSAStartup*, *WSACleanup*, *WSAGetLastError, WSASetLastError*, and *WSAIoctl*. We have already discussed the first three of these functions; the last will be covered in Chapter 7.

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, Windows 98, and Windows NT platforms. 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. In addition, there is the dilemma of UNICODE vs. non-UNICODE Winsock functions. It is up to the programmer to decide if the strings passed to the sending and receiving Winsock functions are UNICODE or ANSI strings. Winsock doesn't care what you pass as long as 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 Windows system 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 so 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*.

Conclusion

In this chapter, we presented the core Winsock functions that are required for connection-oriented and connectionless communication using the TCP and UDP protocols specifically. For connection-oriented communication, we demonstrated 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 communication, we also described how to send and receive data. Since this chapter was designed to introduce the core Winsock APIs, we did not address network programming performance considerations. Chapter 6 will address performance issues and introduce the Microsoft Winsock extensions *TransmitFile*, *TransmitPackets*, *AcceptEx*, *GetAcceptExSockaddrs*, *ConnectEx*, *DisconnectEx*, and *WSARecvMsg,* which can help you write high performance, scalable Winsock applications.

Our discussions so far have demonstrated using Winsock with the IPv4 protocol. In the next three chapters, we will present the design of the Winsock architecture and show how to use other available protocols.

Chapter 2

Winsock Design

Now that we've introduced the Winsock basics, we'll delve into the system architecture and how Winsock fits into the overall system design. Afterward, we'll discuss protocol characteristics and how applications can enumerate the installed protocols. Then we'll discuss the details of socket creation via the *socket* and *WSASocket* functions and how they interact with the Winsock catalog.

System Architecture

Before getting into the system architecture, it is important to mention that applications should not rely on the specific details mentioned in this chapter (names of drivers, DLLs, etc.) as they may change in future releases. Instead, applications should be concerned with the Winsock specification—but it is also important to have a general understanding of the overall system design. The following section pertains mainly to the Windows NT–based operating systems. At the end of this section, we will note some of the differences between Windows NT and Windows 95, Windows 98, and Windows Me.

The majority of the Winsock API is implemented in WS2_32.DLL and is declared in WINSOCK2.H. The only exception is for the Microsoft–specific Winsock extensions (such as TransmitFile, AcceptEx, etc.), which are located in MSWSOCK.DLL. These extensions are not a part of the formal Winsock specification but have been added by Microsoft. Also, because these are Microsoft-specific extensions and are not part of the formal Winsock specification, some of the extension APIs are available only on certain versions of Windows. Chapter 6 covers these APIs in detail.

When an application calls into the Winsock API, it calls into WS2_32.DLL. The Winsock DLL performs some parameter validation and then determines which protocol service provider the call should be routed to. There may be multiple providers installed in the Winsock catalog and WS2_32.DLL determines which provider should handle the call.

There are two types of providers: base and layered. A base provider sits on top of a transport protocol, such as Microsoft TCP/IP and UDP/IP providers or the Resource Reservation Protocol (RSVP) provider, which implements QOS (Chapter 10). The Microsoft base provider consists of MSAFD.DLL and MSWSOCK.DLL, but actually exposes one or more providers for the individual protocols of TCP/IP, IPX/SPX, NetBIOS, AppleTalk, etc. Later in this chapter we'll describe how to programmatically enumerate the providers available on the system.

A layered provider sits below WS2_32.DLL and above a base provider and can intercept and manipulate the Winsock calls. That is, if an application creates a socket from the layered provider, the layered provider will intercept all Winsock calls using that socket. The layered provider may block, modify, or pass the call unmodified to the underlying provider. Also, there may be numerous layered providers installed, one on top of another. Layered service providers are covered in detail in Chapter 12.

Once a Winsock call makes it to the base provider, the base provider will in turn make calls to the Winsock kernel mode component. Unlike some other operating systems, the Windows NT transport protocols do not have a Winsock-like interface that applications can use to directly talk to them. Instead, they implement a much more general API called the *Transport Driver Interface* (*TDI*). The generality of this API allows the Windows NT subsystems to free themselves from being tied to a particular version-of-the-decade network-programming interface. The *sockets* emulation is provided by the Winsock kernel mode driver (currently implemented in AFD.SYS). This driver is responsible for the connection and buffer management related to providing a *sockets*-like interface to an application. AFD, in turn, talks *TDI* to the transport protocol driver (see Figure 2-1).

For Windows 95, Windows 98, and Windows Me, the overall architecture is similar to Windows NT except for the kernel mode components. Since Windows 95, Windows 98, and Windows Me implement drivers within VXDs, there is no AFD.SYS or TCPIP.SYS. Instead, the drivers are AFVXD.VXD, WSOCK2.VXD, WSOCK.VXD, and so on. Of course, as we stated earlier, applications should not concern themselves with the actual filenames of drivers or system components. The Winsock API is a specification that is available across all versions of the Windows operating system.

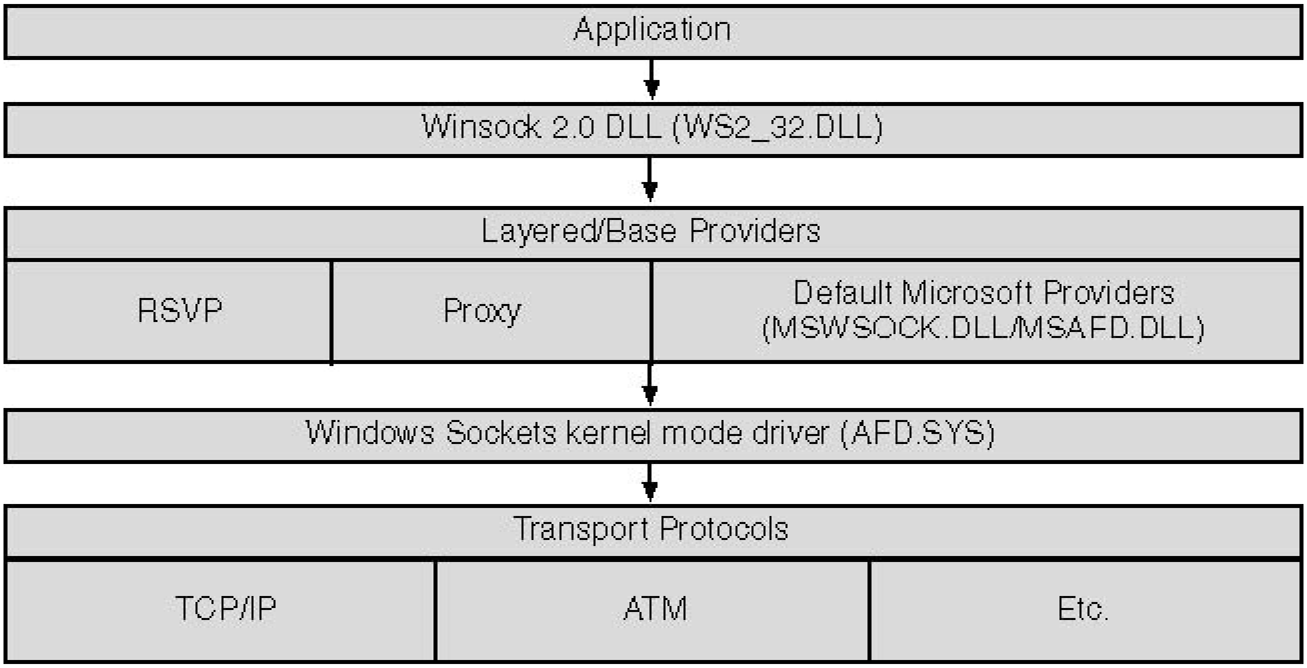

Application
Winsock 2.0 DLL (WS2_32.DLL)
Layered/Base Providers
RSVP
Proxy
Default Microsoft Providers
(MSWSOCK.DLL/MSAFD.DLL)
Windows Sockets kernel mode driver (AFD.SYS)
Transport Protocols
TCP/IP
ATM
Etc.

Figure 2-1 *Winsock system architecture*

Protocol Characteristics

Now that we've given some information about the internals of Winsock, we'll discuss the protocols that may be accessed. As we mentioned earlier, a Winsock provider implements a protocol that exhibits certain characteristics. A multitude of different transport protocols are available on Windows, such as TCP, UDP, IPX, and SPX. Each protocol behaves differently. Some require a connection to be established before sending or receiving data. Others don't guarantee the reliability or integrity of the data. In this section we'll look at the characteristics that apply to protocols.

Message-Oriented

A protocol is said to be message-oriented if for each discrete write command, it transmits 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 2-2, 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 *recv* calls with a 256-byte buffer. Each call in succession returns 128, 64, and 32 bytes. The first call to *recv* 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.

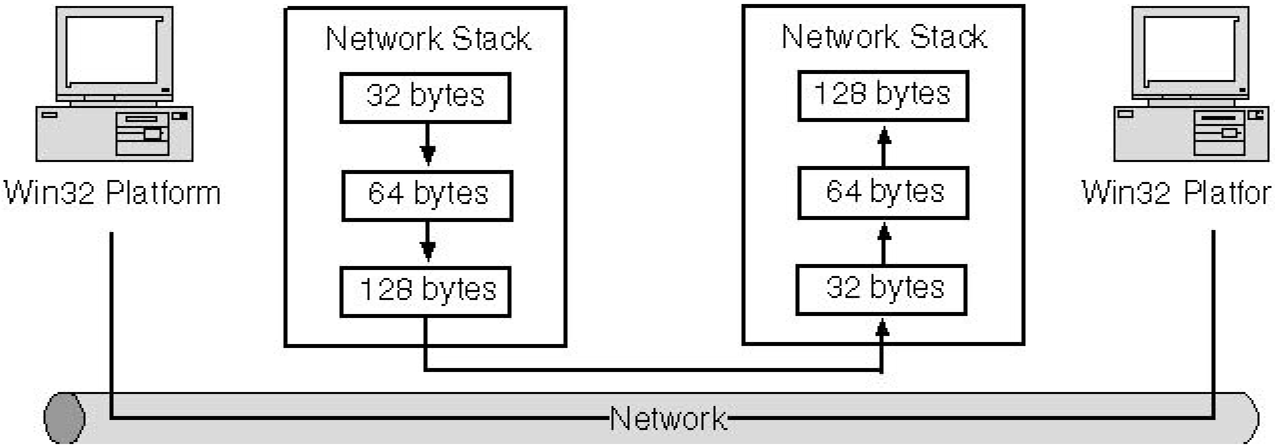


Figure 2-2 *Datagram services*

Stream-Oriented

A protocol that does not preserve message boundaries is often referred to as a stream-based protocol. Be aware that the term *stream based* is often loosely used to imply additional characteristics. Stream service is defined as transmitting 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 to 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 that the client call supplied. In Figure 2-3, 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 the peer doesn't have to send a packet with only the acknowledgement. Sending many small packets is inefficient and adds substantial overhead for error checking and acknowledgments.

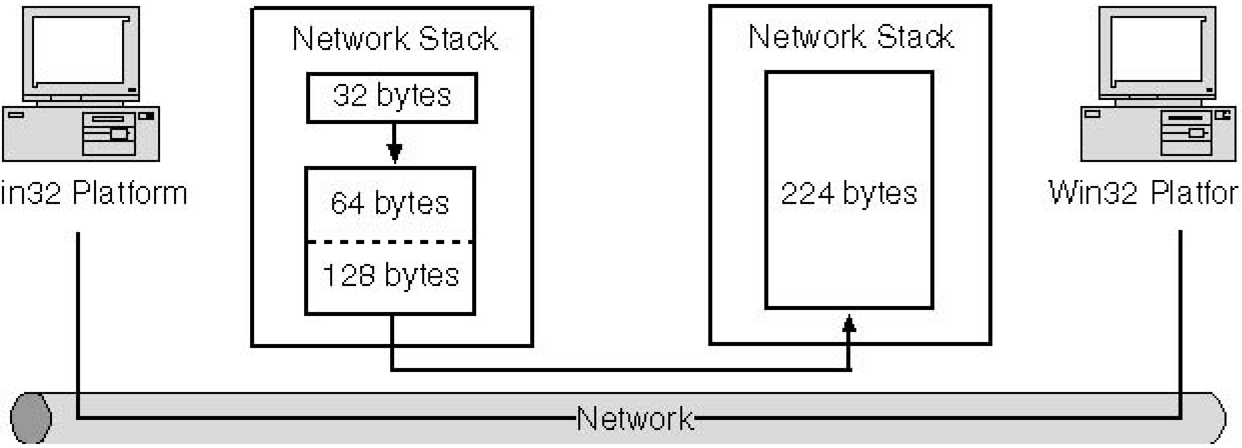


Figure 2-3 *Stream services*

On the receiving end, the network stack pools together all incoming data for the given process. Take a look at Figure 2-2. If the receiver performs a *recv* 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.

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 the receiving application reads data chunks of any size. Combining the sender in Figure 2-2 with the receiver in Figure 2-3 illustrates how pseudo streams work. The sender must send each individual packet separately, but the receiver is free to coalesce them in whatever sizes are available. For the most part, treat pseudo streaming as you would a stream-oriented protocol.

Connection-Oriented and Connectionless

A protocol 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 substantial overhead. In addition, most connection-oriented protocols guarantee delivery, which increases 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 if the recipient is expecting to receive a letter or if severe storms are preventing the post office from delivering the message.

Note that for some connectionless protocols, such as UDP, a Winsock application may call *connect* with the destination's IP address but this does not imply that any physical connection is established. It is simply a convenient way to associate a destination address with the socket so that the *send* and *WSASend* APIs may be used instead of *sendto* and *WSASendTo*.

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. 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 of 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 that it was sent. Obviously, a protocol that does not preserve order makes no such guarantees.

In most cases, 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. Note that by ensuring 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 and connection-oriented protocols do. Why would anyone use datagrams with all these faults? In general, connectionless protocols are much 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 data-grams 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 with connection-oriented protocols only. 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 closed completely.

Broadcast Data

To broadcast data is to send data from one workstation so that all other workstations on the LAN 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 if 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 if 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 between networks.

Multicast Data

Multicasting is the capability of one process to send data that one or more recipients will receive. 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 that group address only 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 9 covers multicast programming from Winsock and other critical multicasting issues.

Quality of Service (QOS)

QOS is an application's capability to request certain network bandwidth requirements to be dedicated for exclusive use. One good use for QOS is real-time video streaming. 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 was a period during which data was not being received fast enough, the playback routine had a certain number of buffered frames that it could 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 10.

Partial Messages

Partial messages apply to message-oriented protocols only. Let's say an application wants to receive a message but the local computer has received only part of the data. This can occur if the sending computer is transmitting large messages and the local machine does 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 large enough to require being broken 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 for application developers 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 network from machine B. A router linking the two networks separates the two machines. A routable protocol realizes that machine B is not on the same network as machine A; therefore, the protocol directs the data to the router, which decides how to best forward it 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 Windows platforms that is not capable of being routed.

Other Characteristics

Each protocol that Windows supports has 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 provides a facility to enumerate each available protocol provider and query its characteristics. The next section of this chapter explains this function and presents a code sample.

Winsock Catalog

The Winsock catalog is a database that contains the different protocols available on the system. Winsock 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*, 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_INFO {
    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;
    TCHAR               szProtocol[WSAPROTOCOL_LEN + 1];
} WSAPROTOCOL_INFO, FAR * LPWSAPROTOCOL_INFO;
```

The easiest way to call *WSAEnumProtocols* is to make the first call with *lpProtocolBuffer* equal to *NULL* and set *lpdwBufferLength* 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 called ENUMCAT.C on the companion CD enumerates all installed protocols and prints out each protocol's characteristics.

The most commonly used field of the *WSAPROTOCOL_INFO* structure is *dwServiceFlags1*, which is a bit field for the various protocol attributes. Table 2-1 lists the various bit flags that can be set in the field and briefly describes the meaning of each property.

Table 2-1 *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. Multipoint communication is covered in Chapter 9.
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. QOS is covered in Chapter 10.
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*.
XP1_INTERRUPT	Reserved flag.

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 an 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 a given protocol.

Winsock Catalog and Win64

On 64-bit Windows, it is possible to run 32-bit applications under the WOW64 (Windows on Windows) subsystem. Because both 32-bit and 64-bit applications may need to access the Winsock catalog, the system maintains two separate catalogs. When a 64-bit Winsock application runs and calls *WSAEnumProtocols*, the 64-bit catalog is used. Likewise, when a 32-bit Winsock application calls *WSAEnumProtocols*, the 32-bit catalog is used. This will become more important when dealing with the Winsock Service Provider Interface, which is discussed in Chapter 12.

Creating Sockets

In Chapter 1 we saw simple examples of how to create a socket using the *socket* function. This function takes three parameters: address family, socket type, and protocol. When an application creates a socket, the Winsock catalog is consulted and an attempt is made to match the supplied parameters with those contained in each *WSAPROTOCOL_INFO* structure. If the parameters match, then a socket is created from that provider. Note that in some instances the protocol parameter can be 0. If the *dwProviderFlags* field of the *WSAPROTOCOL_INFO* structure is *PFL_MATCHES_PROTOCOL_ZERO* and if the requested address family and socket type match its entries, then that provider is used to create a socket. For example, consider the following call:

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

Winsock will enumerate the catalog and first match the address family followed by the socket type. Since the protocol value is 0 and the MSAFD TCP provider contains the *PFL_MATCHES_PROTOCOL_ZERO* flag, this call will create a TCP socket from the MSAFD TCP provider. The system will not attempt to match the request to any further providers.

In some instances, multiple providers may share the same address family, socket type, and protocol. This is the case with the RSVP provider. The RSVP provider offers QOS over TCP/IP and UDP/IP. Because multiple providers share the same address family, socket type, and protocol, there is no way to use the *socket* API to create a socket from the RSVP provider. To do so, you must use the Winsock 2 function *WSASocket*, which is defined as

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

The first three parameters are the same as those of *socket* but the fourth parameter takes a *WSAPROTOCOL_INFO* structure. If *lpProtocolInfo* references a provider entry and each of the first three parameters is the predefined value *FROM_PROTOCOL_INFO*, then a socket is created from the given provider. An application can create an RSVP socket using this method.

The fourth parameter deals with socket groups that are discussed in the Winsock specification but are not implemented in Windows. The last parameter is optional flags that may be passed. For now, the only flag of importance is *WSA_FLAG_OVERLAPPED*. If you plan on using overlapped IO (as described in Chapter 5), then this flag needs to be present when creating the socket. Note that when using the *socket* API, the overlapped flag is always implied. The other socket flags pertain to multicasting and are covered in Chapter 9.

Conclusion

In this chapter, you have seen how Winsock fits into the overall system architecture and how various protocols plug into the system. In addition, you looked at the characteristics that protocols exhibit as well as how to programmatically enumerate the Winsock catalog to obtain this information. Finally, you have seen how to create a socket from an explicit provider using the *WSASocket* API. In the next chapter, we'll examine the IP protocol, including IPv4 and IPv6, in more detail.

Chapter 3

Internet Protocol

This chapter describes the Internet Protocol (IP). As we discussed in Chapter 1, to establish communication through Winsock you must understand how to address a workstation for a particular protocol (as we demonstrated with IPv4 in Chapter 1). This chapter covers IPv4 and IPv6. Chapter 4 will cover the most common protocols available on Windows platforms.

IPv4 is commonly known as the network protocol that the Internet uses. IP is widely available on most computer operating systems and can be used on most LANs, such as a small network in your office, and on WANs, such as the Internet. With the explosion in the number of computers on the Internet, the limitations of IPv4 are becoming apparent, and as a result, the next generation IP was developed, which is known as IPv6.

In this chapter, we will discuss the background, addressing scheme, name resolution, and Winsock specifics for both IPv4 and IPv6. Then, we'll discuss how to write applications that seamlessly operate over either version of IP.

IPv4

IPv4 was developed by the U.S. Department of Defense's Advanced Research Project Agency (ARPA), which built an experimental packet switching network in the 1960s. The initial network protocols were cumbersome, which led to the development of a better protocol in the mid 1970s. This research eventually led to IPv4 as well as TCP.

Addressing

In IPv4, computers are assigned an address that is represented as a 32-bit number, formally known as an IPv4 address. IPv4 addresses are typically represented in a dotted decimal format in which each octet (8 bits) of the address is converted to a decimal number and separated by a period (“dots”).

IPv4 addresses are divided into classes that describe the portion of the address assigned to the network and the portion assigned to endpoints. Table 3-1 lists the different classes.

Table 3-1 *IPv4 Address Classes*

Class	Network Portion	First Number	Number of Endpoints
A	8 bits	0–127	16,777,216
B	16 bits	128–191	65,536
C	24 bits	192–223	256
D	N/A	224–239	N/A
E	N/A	240–255	N/A

When specifying an IP address, the number of bits indicating the network portion can be appended to the dotted decimal address after a back slash (/). For example, the address 172.31.28.120/16 indicates that the first 16 bits make up the network portion of the address. This is equivalent to a subnet mask of 255.255.0.0.

The last two entries in Table 3-1 are special classes of IPv4 addresses. Class D addresses are reserved for IPv4 multicasting and class E addresses are experimental. Also, there are several blocks of addresses that have been reserved for private use and cannot be used by a system on the Internet. They are the following:

- 10.0.0.0–10.255.255.255 (10.0.0.0/8)
- 172.16.0.0–172.31.255.255 (172.16.0.0/12)
- 192.168.0.0–192.168.255.255 (192.168.0.0/16)

Finally, there is the loopback address (127.0.0.1), which is a special address that refers to the local computer.

To list the IPv4 addresses assigned to the local interfaces, the IPCONFIG.EXE command can be used to list each network adapter and the IPv4 address(es) assigned to it. If an application needs to programmatically obtain a list of its IPv4 addresses, it can call *WSAIoctl* with the *SIO_ADDRESS_LIST_QUERY* command, which is covered in Chapter 7. In addition, the IP Helper APIs provide this function and are described in Chapter 16.

We've discussed the breakdown of the IPv4 address space, and from within these different address classes there are three types of IPv4 addresses: unicast, multicast, and broadcast. Each address type will be covered in the next sections.

Unicast

Unicast addresses are those addresses that are assigned to an individual computer interface. Only one interface may be assigned that address. If another computer is configured with the same address on the network, then that is an error that will result in data being delivered incorrectly. Classes A, B, and C comprise the unicast address space for IPv4.

Typically, an interface on a host is assigned an IPv4 (unicast) address either statically or by a configuration protocol like Dynamic Host Configuration Protocol (DHCP). If a DHCP server cannot be reached, the system automatically assigns an address in the range of 169.254.0.0/16 using Automatic Private IP Addressing (APIPA).

To prevent having to memorize numeric IP addresses, an IPv4 address can be associated to the host computer name by using the Domain Name System (DNS). Later, we will discuss how to resolve the host name to its IPv4 address (and its IPv6 address as well).

Multicast

Multicast addresses are not assigned to a specific interface. Instead, multiple computers may “join” a multicast group listening on a particular multicast address. Everyone joined to that group will receive any data destined to that address. Multicast addresses are class D addresses. One of the greatest benefits to multicasting is the capability to deliver multicast data to only those machines that are interested in that data. IP multicasting is discussed in detail in Chapter 9.

Broadcast

IPv4 supports broadcasting data. This means that data sent to the limited broadcast address, 255.255.255.255, will be received and processed by every machine on the local network. This is generally considered a bad practice because even those computers that are not interested in the broadcast data must process the packet.

If applications require broadcasting, it is better to use subnet directed broadcasts. This is still broadcasting data, but as the name implies it is directed to machines on a specific subnet only. For example, a datagram sent to 172.31.28.255 will be received by every machine on only that same subnet.

IPv4 Management Protocols

The IPv4 protocol relies on several other protocols to function. The three support protocols we are most interested in is the Address Resolution Protocol (ARP), the Internet Control Message Protocol (ICMP), and the Internet Group Management Protocol (IGMP).

ARP is used to resolve the 32-bit IPv4 address into a physical or hardware address so the IPv4 packet can be wrapped in the appropriate media frame (such as an Ethernet frame). A host must resolve the next-hop IPv4 address to its corresponding hardware address before sending data on the wire. If the destination address is on the local network, the ARP request is made for the destination's physical address. If one or more routers separate the source from the destination, an ARP request is made for the default gateway and the packet is forwarded to it. The IP Helper API contains some ARP routines and is described in Chapter 16.

ICMP is designed to send status and error messages between IPv4 hosts. The types of messages include echo requests and replies, destination unreachable, and time exceeded. ICMP is also used to discover nearby routers. Chapter 11 will go into more detail on ICMP and will illustrate how to send your own ICMP messages.

IGMP is used to manage multicast group membership. When applications on a host join multicast group, the host sends out IGMP membership reports, which inform routers on the network segment which multicast groups data is to be received on. Routers need this information to forward multicast packets destined to these multicast groups to network segments only when there are receivers interested in it. IGMP will be discussed in more detail in Chapter 9.

Addressing IPv4 from Winsock

In Winsock, applications specify IPv4 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 you are using the IP address family. The *sin_port* field defines which TCP or UDP communication port will be used to identify a server service. Note that the port number does not actually apply to the IPv4 protocol but is a property of the transport layer protocol(s) encapsulated within an IPv4 header, such as TCP or UDP.

Applications should be particularly careful in choosing a port because some of the available port numbers are reserved for well-known services, such as FTP and HTTP. The ports that well-known services use are controlled and assigned by the Internet Assigned Numbers Authority (IANA) and are listed on its Web page at http://www.iana.org/assignments/port-numbers. Essentially, the port numbers are divided into the following three ranges: well-known, registered, and dynamic and/or private ports.

- 0–1023 are controlled by IANA and are reserved for well-known services.
- 1024–49151 are registered ports listed by 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 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*. Also, it is valid for clients to send or connect without explicitly binding to a local address and port. In this case, the system will implicitly bind the socket to a local port from the range of 1024 to 5000. This is the same behavior that occurs if an application explicitly binds the socket but specifies a local port of zero.

The *sin_addr* field of the *SOCKADDR_IN* structure is used for storing an IPv4 address as a four-byte, network-byte-ordered quantity, which is an unsigned long integer data type. Depending on how this field is used, it can represent a local or a 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.

All fields of this and every other socket address structure need to be in network byte order. However, if applications use the name resolution and assignment APIs discussed later in this chapter, the necessary conversion is automatically performed. It is only when an application explicitly assigns or retrieves values from the structure members that the byte order conversion is required. Byte ordering was described in Chapter 1.

IPv6

With the explosion in the number of computers on the Internet, the limitations of IPv4 are becoming apparent. First and foremost, the number of available IPv4 addresses is being exhausted. This has led to the use of network address translators (NATs), which map multiple private addresses to a single public IP addresses. NATs are useful for client-server applications but can be problematic when connecting two organizations that use the private address space. Also, NATs must sometimes be aware of the underlying protocols to perform the appropriate address translation.

Second, IPv4 addressing is not entirely hierarchical, which means that the Internet backbone routers must maintain vast routing tables to deliver IPv4 packets correctly to any location on the Internet.

Another incentive for developing IPv6 is to provide simpler configuration. With IPv4, addresses must be assigned statically or via a configuration protocol such as DHCP. Ideally, hosts would not have to rely upon the administration of a DHCP infrastructure. Instead, they will be able to auto configure themselves based on the network segment on which they are located.

A developer-release version of IPv6 is provided with Windows XP. For Windows 2000, a technology preview IPv6 protocol is available for download from http://www.microsoft.com/ipv6. For Windows NT 4.0, a Microsoft Research IPv6 protocol may also be obtained from http://www.microsoft.com/ipv6.

In this section, we will cover the different types of IPv6 addresses, the support protocols that IPv6 uses, and how IPv6 addresses are handled from Winsock. Although we will discuss addressing and name resolution, we will not cover all aspects of IPv6, such as routing or setting up an IPv6 network. For more information, consult the Windows XP online help or the book ***Understanding IPv6***, by Joseph Davies (Microsoft Press, 2002).

Addressing

The most noticeable difference between IPv4 and IPv6 addresses is that an IPv6 address is 128 bits, which is four times larger than an IPv4 address. One reason for such a large address space is to subdivide the available addresses into a hierarchy of routing domains that reflect the Internet's topology. Table 3-2 lists a portion of how the address space is allocated and lists the address prefix for each portion. The address prefix denotes the high order bits of an IPv6 address. IPv6 addressing is described in RFC 2373.

Table 3-2 *IPv6 Address Allocation*

Allocation	Address Prefix	Fraction of Address Space
Reserved	0000 0000	1/256
Reserved for NSAP allocation	0000 001	1/128
Aggregatable global unicast addresses	001	1/8
Link-local unicast addresses	1111 1110 10	1/1024

Site-local unicast addresses	1111 1110 11	1/1024
Multicast addresses	1111 1111	1/256

An IPv6 address is typically expressed in 16-bit chunks displayed as hexadecimal numbers separated by colons. The following is an example of an IPv6 address:

```
21DA:00D3:0000:2F3B:02AA:00FF:FE28:9C5A
```

Leading zeroes within each 16-bit block may be removed, as seen here:

```
21DA:D3:0:2F3B:2AA:FF:FE28:9C5A
```

Many IPv6 addresses contain long sequences of zeroes, which may be compressed by substituting two colons for the block of zeros. For example, the following address:

```
FE80:0:0:0:12:0:34:56
```

can be compressed to:

```
FE80::12:0:34:56
```

Note that only a single contiguous sequence of 16-bit zero blocks may be compressed.

Depending on the platform, you can use one of two methods to obtain a list of the IPv6 addresses assigned to a computer's interfaces. For the Microsoft Research and Windows 2000 Technology Preview stacks downloaded from the Web as well as Windows XP Home Edition and Windows XP Professional, the IPV6.EXE command is used. To enumerate the IPv6 interfaces, execute IPV6.EXE if at the command prompt. For all versions of Windows 2000 and Windows XP (including future versions of Windows releases), the NETSH.EXE command may also be used. The command syntax is: NETSH.EXE interface IPv6 show interface. To programmatically obtain the configuration of local interfaces, the *SIO_ADDRESS_LIST_QUERY* ioctl (Chapter 7) and the IP Helper API (Chapter 16) can be used.

There are three basic types of IPv6 addresses: unicast, anycast, and multicast. Note that IPv6 does not define a broadcast address (multicasting is used instead). In the following sections, we will discuss each address type.

Unicast

A unicast address identifies a single interface. With IPv6, however, an interface will most likely have more than one unicast address assigned to it. There are four types of unicast addresses that you will likely encounter:

- Link-local addresses
- Site-local addresses
- Global addresses

- Compatibility addresses

An interface will always have a link-local address assigned to it—each physical network interface is auto configured with one. A link-local address is used to communicate only with other nodes on the same link. Link-local address always begin with an fe80::/64 prefix. Also, because no routing information is kept for link-local addresses, the interface index is often displayed with the address. Every physical interface on the system is assigned an adapter index number (also known as a scope ID). When a link-local address is assigned to an interface, the link number is appended to the address. The following address is the link-local address assigned to the physical adapter whose interface index number is five.

```
fe80::250:8bff:fea0:92ed%5
```

In Winsock, if a connection is being established using link-local addresses, then the interface index must be present to indicate which link the remote host is reachable from. An IPv6 link-local address is synonymous with an IPv4 APIPA address discussed earlier in the chapter.

For example, consider host A, which has the link-local address `fe80::250:8bff:fea0:92ed%5` and host B, which has the link-local address `fe80::250:daff:fec3:9e34%4`. If host A issues a *connect* to host B, it would use the destination address of B with its own scope ID that can reach host B. The address to connect to would be `fe80::250:daff:fec3:9e34%5.`

Site-local addresses are IPv6 addresses that are reachable only on the local network environment, such as the corporate network at a particular site. These addresses are comparable to the IPv4 private address space because they cannot be reached from other sites or the Internet and routers on the private network do not forward this traffic beyond the local site. Site-local addresses use the prefix fec0::/48. Site-local addresses must be assigned from either an IPv6 router or via DHCPv6. Currently, Microsoft's implementation of IPv6 does not support DHCPv6. IPv6-enabled routers will send Router Advertisement (RA) messages, which advertise the network portion of the address (such as the first 64 bits of the address consisting of the 48-bit site-local prefix and a 16-bit subnet ID), which the host will then use to assign a site-local address to the interface on which the RA was received.

Global addresses are just that: globally reachable on IPv6 Internet. Global addresses begin with 001. The remaining 61 bits of the first 64 bits are used to establish a routing hierarchy, and the last 64 bits comprise the interface identifier that uniquely identifies a network interface on a subnet. Global addresses are also assigned via router advertisements or by using DHCPv6.

The last type of unicast addresses are compatibility addresses, which are designed to aid in the transition from IPv4 to IPv6. There are four kinds of compatibility addresses that Windows supports: Intrasite Automatic Tunnel Addressing Protocol (ISATAP), 6to4, 6over4, and IPv4 compatible. ISATAP addresses can be derived from any IPv6 unicast address, such as link-local, site-local, and global addresses. Most often you will see an ISATAP address derived from a link-local address. These addresses also contain an embedded IPv4 address. For example, the ISATAP address `fe80::5efe:172.17.7.2` is a link-local address and contains the IPv4 address of the host

(172.17.7.2). When data is sent from this interface, the IPv6 packet is encapsulated within an IPv4 header. The IPv4 destination address is obtained from the v4 address embedded within the IPv6 ISATAP destination address. The v4 address must be globally reachable for two endpoints to communicate via automatic tunneling. ISATAP addresses are currently an Internet Engineering Task Force (IETF) draft.

The second type of compatibility address is called 6to4 and is described in RFC 3056. 6to4 addresses use the global prefix 2002:WWXX:YYZZ::/48, in which WWXX:YYZZ is the hexadecimal-colon representation of *w.x.y.z*, a public IPv4 address. 6to4 allows IPv6/IPv4 hosts to communicate over an IPv4 routing infrastructure.

Windows XP provides a 6to4 service. This service allows hosts to communicate with other 6to4 hosts within the same site, 6to4 hosts connected to the Internet, 6to4 hosts in other sites across the IPv4 Internet, as well as with hosts on the IPv6 Internet using a 6to4 relay router. On Windows XP, the 6to4 service is configured to run automatically. If there is a public IPv4 address assigned to an interface, a 6to4 Tunneling Interface (interface index 3) is created and assigned the 6to4 address(es).

The third type of compatibility address is 6over4, which is a tunneling technique using IPv4 multicasting. It allows IPv4 and IPv6 nodes to communicate using IPv6 over an IPv4 infrastructure. This technique is described in RFC 2529.

The last type of compatibility address is the IPv4 compatible address. These addresses take the form of 0:0:0:0:0:0:*w.x.y.z* (or ::*w.x.y.z*) in which *w.x.y.z* is the dotted decimal representation of a public IPv4 address. When a IPv4 compatible address is used by an application as the destination, the IPv6 traffic is automatically encapsulated within an IPv4 header and sent to the destination over the IPv4 network.

Anycast

Anycast is an address that identifies multiple interfaces. The purpose of these addresses is to route packets destined to an anycast address to the nearest interface assigned that anycast address. A good scenario for anycast addresses is when there are several nodes on the network that provide a certain service. Each machine can be assigned the same anycast address and clients interested in contacting that service will be routed to the nearest member. This is different from multicast because this communication is one to one of many instead of one to many. Currently however, anycast addresses are assigned to routers only.

Multicast

Multicasting in IPv6 is similar to IPv4 multicasting. A process joins a multicast group on a particular interface and data destined to that multicast address is received. IPv6 multicast addresses begin with 1111 1111 (FF). IPv6 multicasting and IPv6 multicast addresses are covered in more detail in Chapter 9.

IPv6 Management Protocols

IPv6 requires only a single helper protocol: Internet Control Message Protocol for IPv6 (ICMPv6), which is defined in RFC 2463. ICMPv6 provides the same types of services that ICMP does, such as destination unreachable, echo and echo reply, but also provides a mechanism for Multicast Listener Discovery (MLD) and Neighbor Discovery (ND). MLD replaces IGMP and ND replaces ARP.

Addressing IPv6 from Winsock

To specify IPv6 addresses in Winsock applications, the following structure is used.

```
struct sockaddr_in6 {
                short            sin6_family;
                u_short          sin6_port;
                u_long           sin6_flowinfo;
                struct in6_addr sin6_addr;
                u_long           sin6_scope_id;
};
```

The first field simply identifies the address family, which is AF_INET6, and the second is the port number. All fields within this structure must be in network byte order. Note that all the information discussed about port numbers in the IPv4 section apply equally to IPv6 because the port number is a property of the encapsulated protocols, such as TCP and UDP, which are also available from IPv6. The third field, *sin6_flowinfo*, is used to mark the traffic for the connection but is not implemented in the Microsoft IPv6 stack. The fourth field is a 16-byte structure that contains the binary IPv6 address. The last member*, sin6_scope_id*, indicates the interface index (or scope ID) on which the address is located. Remember that for link-local addresses, the local scope ID on which the destination is located must be specified and the *sin6_scope_id* field is used for this. Site-local addresses may reference the site number as the scope ID. Global addresses do not contain a scope ID.

One last item to note is that the *SOCKADDR_IN6* structure is 28 bytes in length and the *SOCKADDR* and *SOCKADDR_IN* structures are only 16 bytes long.

Address and Name Resolution

In this section, we'll cover how to assign both literal string addresses and resolve names to the address specific structures for both IP protocols. First, we will cover the new name resolution APIs: *getaddrinfo* and *getnameinfo*. These APIs have replaced the IPv4 specific routines. Then we'll cover the generic Winsock APIs for converting between string literal addresses and socket address structure. These APIs are *WSAAddressToString* and *WSAStringToAddress*. Note that these functions perform only address conversion and assignment, not name resolution.

Next, the IPv4 specific legacy routines will be described. We include the legacy API descriptions in case legacy code needs to be maintained, but any new projects should use the newer API functions. By using the newer functions it will be trivial to write an application that can seamlessly operate over both IPv4 and IPv6, which is the topic of the last section in this chapter.

Finally, note that all the name resolution functions covered in this chapter deal only with resolving names and not registering a name with an address. This is accomplished by the Winsock Registration and Name Resolution (RNR) APIs discussed in Chapter 8.

Name Resolution Routines

Along with IPv6, several new name resolution functions were introduced that could handle both IPv4 and IPv6 addresses. The legacy functions like *gethostbyname* and *inet_addr* work with IPv4 addresses only. The replacement functions are named *getnameinfo* and *getaddrinfo*.

These new name resolution routines are defined in WS2TCPIP.H. Also, note that although these functions are new for Windows XP, they can be made available to work on all Winsock 2 enabled platforms. This is done by including the header file WSPIAPI.H before including WS2TCPIP.H. The compiled binary will then run on all Winsock 2–enabled platforms, such as Windows 95, Windows 98, Windows Me, Windows NT 4.0, and Windows 2000.

The *getaddrinfo* function provides protocol-independent name resolution. The function prototype is

```
int getaddrinfo(
                const char FAR *nodename,
                const char FAR *servname,
                const struct addrinfo FAR *hints,
                struct addrinfo FAR *FAR *res
);
```

The *nodename* parameter specifies the NULL-terminated host name or literal address. The *servname* is a NULL-terminated string containing the port number or a service name such as “ftp” or “telnet.” The third parameter, *hints*, is a structure that can pass one or more options that affect how the name resolution is performed. Finally, the *res* parameter returns a linked list of *addrinfo* structure containing the addresses the string name was resolved to. If the operation succeeds, zero is returned; otherwise the Winsock error code is returned.

The *addrinfo* structure is defined as

```
struct addrinfo {
                int             ai_flags;
                int             ai_family;
                int             ai_socktype;
                int             ai_protocol;
                size_t          ai_addrlen;
                char            *ai_canonname;
                struct sockaddr *ai_addr;
                struct addrinfo *ai_next;
};
```

When passing hints into the API, the structure should be zeroed out beforehand, and the first four fields are relevant:

- *ai_flags* indicates one of three values: *AI_PASSIVE*, *AI_CANONNAME*, or *AI_NUMERICHOST*. *AI_CANONNAME* indicates that *nodename* is a computer name like www.microsoft.com and *AI_NUMERICHOST* indicates that it is a literal string address such as “10.10.10.1”. *AI_PASSIVE* will be discussed later.
- *ai_family* can indicate AF_INET, AF_INET6, or AF_UNSPEC. If you wish to resolve to a specific address, type the supply AF_INET or AF_INET6. Otherwise, if AF_UNSPEC is given, then the addresses returned could be either IPv4 or IPv6 or both.
- *ai_socktype* specifies the desired socket type, such as SOCK_DGRAM, SOCK_STREAM. This field is used when *servname* contains the name of a service. That is, some services have different port numbers depending on whether UDP or TCP is used.
- *ai_protocol* specifies the desired protocol, such as IPPROTO_TCP. Again, this field is useful when *servname* indicates a service.

If no hints are passed into *getaddrinfo*, the function behaves as if a zeroed hints structure was provided with an *ai_family* of AF_UNSPEC.

If the function succeeds, then the resolved addresses are returned via *res*. If the name resolved to more than one address, then the result is a linked list chained by the *ai_next* field. Every address resolved from the name is indicated in *ai_addr* with the length of that socket address structure given in *ai_addrlen*. These two fields may be passed directly into *bind, connect, sendto*, etc.

The following code sample illustrates how to resolve a hostname along with the port number before making a TCP connection to the server.

```
SOCKET                          s;
struct addrinfo         hints,
                                *result;
int                             rc;

memset(&hints, 0, sizeof(hints));
hints.ai_flags = AI_CANONNAME;
hints.ai_family = AF_UNSPEC;
hints.ai_socktype = SOCK_STREAM;
hints.ai_protocol = IPPROTO_TCP;
```

```
rc = getaddrinfo("foobar", "5001", &hints, &result);
if (rc != 0) {
                // unable to resolve the name
}
s = socket(result->ai_family, result->ai_socktype,
result->ai_protocol);
if (s == INVALID_SOCKET) {
                // socket API failed
}
rc = connect(s, result->ai_addr, result->ai_addrlen);
if (rc == SOCKET_ERROR) {
                // connect API failed
}
freeaddrinfo(result);
```

In this example, the application is resolving the hostname "foobar" and wants to establish a TCP connection to a service on port 5001. You'll also notice that this code doesn't care if the name resolved to an IPv4 or an IPv6 address. It is possible that "foobar" has both IPv4 and IPv6 addresses registered, in which case *result* will contain additional *addrinfo* structures linked by the *ai_next* field. If an application wanted only IPv4 addresses registered to "foobar," the *hints.ai_family* should be set to AF_INET. Finally, note that the information returned via *res* is dynamically allocated and needs to be freed by calling the *freeaddrinfo* API once the application is finished using the information.

Another common action that applications perform is assigning a literal string address such as "172.17.7.1" or "fe80::1234" into a socket address structure of the appropriate type. The *getaddrinfo* function does this by setting the *AI_NUMERICHOST* flag within the hints. The following code illustrates this.

```
struct addrinfo         hints,
                                *result;
int                             rc;

memset(&hints, 0, sizeof(hints));
hints.ai_flags = AI_NUMERICHOST;
hints.ai_family = AI_UNSPEC;
hints.ai_socktype = SOCK_STREAM;
hints.ai_protocol = IPPROTO_TCP;
rc = getaddrinfo("172.17.7.1", "5001", &hints, &result);
if (rc != 0) {
                // invalid literal address
}
// Use the result
freeaddrinfo(result);
```

The literal address "172.17.7.1" will be converted to the necessary socket address structure and returned via *result*. Because AF_UNSPEC is passed, the API will determine the correct socket address structure (SOCKADDR_IN or SOCKADDR_IN6) required and convert the address accordingly. As before, the port field of the resulting socket address structure will be initialized to 5001.

Note that if no flags are passed as part of the hints and a literal string address is resolved, the returned structure *addrinfo* containing the converted address will have the *AI_NUMERICHOST* flags set. Likewise, if a hostname is resolved but no hints are passed, the returned structure *addrinfo* flag will contain *AI_CANONNAME*.

The last flag that can be used with *getaddrinfo* is *AI_PASSIVE*, which is used to obtain an address that can be passed to the *bind* function. For IPv4, this would be INADDR_ANY (0.0.0.0) and for IPv6 it would be IN6ADDR_ANY (::). To obtain the bind address, the hints should indicate which address family the passive address is to be obtained for (via *ai_family*), *nodename* should be NULL, and *servname* should be non-NULL—indicating the port number the application will bind to (which can be "0"). If AF_UNSPEC is passed in the hints, then two *addrinfo* structures will be returned, one with the IPv4 bind address and the other with the IPv6 bind address.

The AI_PASSIVE flag is useful after resolving a hostname via *getaddrinfo*. Once the resolved address is returned, the original result's *ai_family* can be used in another call to *getaddrinfo* to obtain the appropriate bind address for that address family. This prevents applications from having to touch the internal socket address structure's fields and also removes the need for two separate code paths for binding the socket depending on which address family the address was resolved to.

The other new name resolution API is *getnameinfo*, which performs the reverse of the *getaddrinfo* function. It takes a socket address structure already initialized and returns the host and service name corresponding to the address and port information. The *getnameinfo* function is prototyped as the following:

```
int getnameinfo(
                const struct sockaddr FAR *sa,
                socklen_t salen,
                char FAR *host,
                DWORD hostlen,
                char FAR *serv,
                DWORD servlen,
                int flags
);
```

The parameters are fairly self-explanatory. *sa* is the socket address structure on which the name information will be obtained for and *salen* is the size of that structure. *host* is the character buffer to receive the host's name. By default, the fully qualified domain name (FQDN) will be returned. *hostlen* simply indicates the size of the host buffer. *serv* is the character buffer to receive the service/port information and *servlen* is the length of that buffer. Finally, the flags parameter indicates how the socket address should be resolved. The possible flag values are the following:

- *NI_NOFQDN* indicates that only the relative distinguished name (RDN) returned. For example, with this flag set the host named "mist.microsoft.com" would return only "mist".
- *NI_NUMERICHOST* indicates to return the string representation of the address and not the hostname.
- *NI_NAMEREQD* indicates if the address cannot be resolved to a FQDN to return an error.
- *NI_NUMERICSERV* indicates to return the port information as a string instead of resolving to a well-known service name, such as "ftp." Note that if the *serv* buffer is supplied with this flag absent

and the port number cannot be resolved to a well-known service, *genameinfo* will fail with error WSANO_DATA (11004).

- *NI_DGRAM* is used to differentiate datagram services from stream services. This is necessary for those few services that define different port numbers for UDP and TCP.

There is a file named RESOLVER.CPP that performs name resolution on the companion CD. A hostname or address is passed to the application along with service or port information and is resolved via *getaddrinfo*. Then for every resolved address returned, *getnameinfo* is called to obtain either the machine name back or the string literal address.

Simple Address Conversion

When an application needs only to convert between string literal addresses and socket address structures, the *WSAStringToAddress* and *WSAAddressToString* APIs are available. *WSAStringToAddress* is not as "smart" as *getaddrinfo* because you must specify the address family that the string address belongs to. The API is the following:

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

The first parameter is the string to convert and the second indicates the address family the string belongs to (such as AF_INET, AF_INET6, or AF_IPX). The third parameter, *lpProtocolInfo*, is an optional pointer to the *WSAPROTOCOL_INFO* structure that defines the protocol provider to use when performing the conversion. If there are multiple providers implementing a protocol, this parameter can be used to specify an explicit provider. The fourth parameter is the appropriate socket address structure to which the string address will be converted and assigned into.

Note that this API will convert string addresses that contain port numbers. For example, the IPv4 string notation allows a colon followed by the port number at the end of the address. For example, "157.54.126.42:1200" indicates the IPv4 address using port 1200. In IPv6, the IPv6 address string must be enclosed in square brackets after which the colon and port notation may be used. For example, `[fe80::250:8bff:fea0:92ed%5]:80` indicates a link-local address with its scope ID followed by port 80. Note that only port numbers will be resolved and not service names (such as "ftp"). For both these examples, if these strings were converted with *WSAStringToAddress*, then the returned socket address structure will be initialized with the appropriate binary IP address, port number, and address family. For IPv6, the scope ID field will also be initialized if the string address contains "%*scope_ID*" after the address portion.

The *WSAAddressToString* provides a mapping from a socket address structure to a string representation of that address. The prototype is

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

This function takes a *SOCKADDR* structure and formats the binary address to a string indicated by the buffer *lpszAddressString*. Again, if there is more than one transport provider for a given protocol, a specific one may be selected by passing its *WSAPROTOCOL_INFO* structure as *lpProtocolInfo*. Note that the address family is a field of the *SOCKADDR* structure passed as *lpsaAddress*.

Legacy Name Resolution Routines

This section covers the legacy name resolution and is included only for the sake of code maintenance, because new applications should be using *getaddrinfo* and *getnameinfo*. The other feature you will notice is that the two new API calls replace eight legacy functions.

The function *inet_addr* converts a dotted IPv4 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 IPv4 address as a 32-bit unsigned long integer in network-byte order, which can be assigned into the *SOCKADDR_IN* field *sin_addr*. Network-byte order is described in Chapter 1.

The reverse of *inet_addr* is *inet_ntoa*, which takes an IPv4 network address and converts it to a string. This function is declared as

```
char FAR *inet_ntoa(
                Struct in_addr in
);
```

The following code sample demonstrates how to create a *SOCKADDR_IN* structure using the *inet_addr* and *htons* functions.

```
SOCKADDR_IN InternetAddr;
INT nPortId = 5150;

InternetAddr.sin_family = AF_INET;

// Convert the proposed dotted Internet address 136.149.3.29
// to a 4-byte integer, and assign it to sin_addr

InternetAddr.sin_addr.s_addr = inet_addr("136.149.3.29");

// The nPortId variable is stored in host-byte order. Convert
// nPortId to network-byte order, and assign it to sin_port.
```

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

The Winsock functions *gethostbyname*, *WSAAsyncGetHostByName*, *gethostbyaddr*, and *WSAAsyncGetHostByAddr* retrieve host information corresponding to a host name or host address from a host database. The first two functions translate a hostname to its network IPv4 addresses and the second two do the reverse—map an IPv4 network address back to a hostname. These 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 DNS, it is the 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, which will be four bytes for IPv4 addresses. 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 prototypes for these functions are

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

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

struct HOSTENT FAR * gethostbyaddr(
    const char FAR * addr,
    int len,
    int type
);

HANDLE WSAAsyncGetHostByAddr(
                HWND hWnd,
                unsigned int wMsg,
                const char FAR *addr,
                int len,
                int type,
                char FAR *buf,
                int buflen
);
```

For the first two functions, the *name* parameter represents a friendly name of the host you are looking for, and the latter two functions take an IPv4 network address in the *addr* parameter. The length of the address is specified as *len*. Also, *type* indicates the address family of the network address passed, which would be AF_INET. All four functions return the results via a *HOSTENT* structure. For the two synchronous functions, the *HOSTENT* is a system-allocated buffer that the application should not rely on being static. The two asynchronous functions will copy the *HOSTENT* structure to the buffer indicated by the *buf* parameter. This buffer size should be equal to *MAXGETHOSTSTRUCT*.

Finally, these and the rest of the asynchronous name and service resolution functions return a HANDLE identifying the operation issued. Upon completion, a window message indicated by *wMsg* is posted to the window given by *hWnd*. If at some point the application wishes to cancel the asynchronous request, the *WSACancelAsyncRequest* function is used. This function is declared as

```
int WSACancelAsyncRequest(
                HANDLE hAsyncTaskHandle
);
```

Keep in mind that the synchronous API calls will block until the query completes or times out, which could take several seconds.

The next type of legacy resolution functions provide the capability to retrieve port numbers for well-known services and the reverse. The API functions *getservbyname* and *WSAAsyncGetServByName* take the name of a well-known service like “FTP” and return the port number that the service uses. The functions *getservbyport* and *WSAAsyncGetServByPort* perform the reverse operation by taking the port number and returning the service name that uses that port. These functions simply retrieve static information from a file named services. In Windows 95, Windows 98, and Windows Me, the services file is located under %WINDOWS%; in Windows NT, it is located under %WINDOWS%\System32\Drivers\Etc.

These four functions return the service information in a *SERVENT* structure that is defined as

```
struct servent {
                char FAR *          s_name;
                char FAR * FAR *s_aliases;
                short               s_port;
                char FAR *          s_proto
};
```

The field *s_name* is the name of the service and *s_aliases* is a NULL terminated array of string pointers, each containing another name for the service. *s_port* is the port number used by the service and *s_proto* is the protocol used by the service, such as the strings “tcp” and “udp.”

These functions are defined as follows:

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

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

struct servent FAR *getservbyport(
                int port,
                const char FAR *proto
);

HANDLE WSAAsyncGetServByPort(
                HWND hWnd,
                unsigned int wMsg,
                int port,
                const char FAR *proto,
                char FAR *buf,
                int buflen
);
```

The *name* parameter represents the name of the service you are looking for. The *proto* parameter optionally points to a string that indicates the protocol that the service in *name* is registered under, such as “tcp” or “udp”. The second two functions simply take the port number to match to a service name. The synchronous API functions return a *SERVENT* structure, which is a system allocated buffer, and the asynchronous ones take an application supplied buffer, which should also be of the size *MAXGETHOSTSTRUCT*.

The last set of legacy name resolution API functions convert between a protocol string name, such as “tcp”, and its protocol number (“tcp” would resolve to IPPROTO_TCP.). These functions are *getprotobyname*, *WSAAsyncGetProtoByName*, *getprotobynumber*, and *WSAAsyncGetProtoByNumber*. The first two convert from the string protocol to the protocol number and the latter two do the opposite—map the protocol number back to its string name. These functions return a *PROTOENT* structure defined as

```
struct protoent {
                char FAR *         p_name;
                char FAR * FAR *p_aliases;
                short              p_proto;
};
```

The first field, *p_name*, is the string name of the protocol, and *p_aliases* is a NULL terminated array of string pointers that contain other names the protocol is known by. Finally, *p_proto* is the protocol number (such as IPPROTO_UDP or IPPROTO_TCP).

These function prototypes are

```
struct protoent FAR *getprotbyname(
                const char FAR *name
);

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

struct protoent FAR *getprotobynumber(
                int number
);

HANDLE WSAAsyncGetProtoByNumber(
                HWND hWnd,
                unsigned int wMsg,
                int number,
                char FAR *buf,
                int buflen
);
```

These functions behave the same way the legacy name resolution functions described earlier do in terms of synchronous and asynchronous functions.

Writing IP Version–Independent Programs

In this section, we'll cover how to develop applications that work seamlessly over IPv4 and IPv6. This method requires using the new name resolution APIs *getaddrinfo* and *getnameinfo* and requires a bit of rearranging Winsock calls from what you are probably used to.

Before we get into the specifics, let's cover some of the basic practices that you should follow. First, applications should not allocate the socket address structures specific to each protocol (such as *SOCKADDR_IN* and *SOCKADDR_IN6* for IPv4 and IPv6, respectively) because they can be different sizes. Instead, a new socket address structure *SOCKADDR_STORAGE* has been introduced that is as large as the largest possible protocol specific address structure and includes padding for 64-bit alignment issues. The following code uses a *SOCKADDR_STORAGE* structure to store the destination IPv6 address.

```
SOCKADDR_STORAGE                                         saDestination;
SOCKET                                   s;
int                                      addrlen,
                                         rc;

s = socket(AF_INET6, SOCK_STREAM, IPPROTO_TCP);
if (s == INVALID_SOCKET) {
                // socket failed
}
addrlen = sizeof(saDestination);
rc = WSAStringToAddress(
                        "3ffe:2900:d005:f28d:250:8bff:fea0:92ed",
                        AF_INET6,
                        NULL,
                        (SOCKADDR *)&saDestination,
                        &addrlen
                        );
if (rc == SOCKET_ERROR) {
                // conversion failed
}
rc = connect(s, (SOCKADDR *)&saDestination, sizeof(saDestination));
if (rc == SOCKET_ERROR) {
                // connect failed
}
```

Second, functions that take an address as a parameter should pass the entire socket address structure and not the protocol specific types like `struct in_addr` or `struct in6_addr`. This is important for IPv6, which might require the scope ID information to successfully connect. The *SOCKADDR_STORAGE* structure containing the address should be passed instead.

Third, avoid hardcode addresses regardless of whether they are IPv4 or IPv6. The Winsock header files define constants for all the address that are hard coded such as the loopback address and the wildcard address used for binding.

Now that some of the basic issues are out of the way, let's move to discussing how an application should be structured to be IP independent. We will divide our discussion into two sections: the client and the server.

Client

For both TCP and UDP clients, the application typically possesses the server (or recipient's) IP address or hostname. Whether it resolves to an IPv4 address or IPv6 address doesn't matter. The client should follow these three steps:

1. Resolve the address using the *getaddrinfo* function. The hints should contain AF_UNSPEC as well as the socket type and protocol depending on whether the client uses TCP or UDP to communicate.
2. Create the socket using the *ai_family*, *ai_socktype*, *and ai_protocol* fields from the *addrinfo* structure returned in step 1.
3. Call *connect* or *sendto* with the *ai_addr* member of the *addrinfo* structure.

The following code sample illustrates these principles.

```
SOCKET                           s;
struct addrinfo hints,
                        *res=NULL
char                    *szRemoteAddress=NULL,
                        *szRemotePort=NULL;
int                     rc;

// Parse the command line to obtain the remote server's
// hostname or address along with the port number, which are contained
// in szRemoteAddress and szRemotePort.
memset(&hints, 0, sizeof(hints));
hints.ai_family = AF_UNSPEC;
hints.ai_socktype = SOCK_STREAM;
hints.ai_protocol = IPPROTO_TCP;
// first resolve assuming string is a string literal address
rc = getaddrinfo(
                        szRemoteAddress,
                        szRemotePort,
                &hints,
                &res
         );
if (rc == WSANO_DATA) {
                // Unable to resolve name - bail out
   }
s = socket(res->ai_family, res->ai_socktype, res->ai_protocol);
if (s == INVALID_SOCKET) {
                // socket failed
}
rc = connect(s, res->ai_addr, res->ai_addrlen);
if (rc == SOCKET_ERROR) {
                // connect failed
}
freeaddrinfo(res);
```

First, you will notice that there are no explicit references to AF_INET or AF_INET6. Also, there's no need to manipulate the underlying SOCKADDR_IN or SOCKADDR_IN6 addresses. The *getaddrinfo* call fully initializes the returned socket address structure with all the required information—address family, binary address, etc.—that is necessary for connecting or sending datagrams.

If the client application needs to explicitly bind the socket to a local port after socket creation but before *connect* or *sendto*, then another *getaddrinfo* call can be made. This call would specify the address family, socket type, and protocol returned from the first call along with the *AI_PASSIVE* flag and desired local port, which will return another socket address structure initialized to the necessary bind address (such as 0.0.0.0 for IPv4 and :: for IPv6).

Server

The server side is a bit more involved than the client side. This is because the Windows IPv6 stack is a dual stack. That is, there is a separate stack for IPv4 and IPv6, so if a server wishes to accept both IPv4 and IPv6 connections, it must create a socket for each one. The two steps for creating an IP independent server are the following:

1. Call *getaddrinfo* with hints containing *AI_PASSIVE*, *AF_UNSPEC*, and the desired socket type and protocol along with the desired local port to listen or receive data on. This will return two *addrinfo* structures: one containing the listening address for IPv4 and the other containing the listening address for IPv6.
2. For every *addrinfo* structure returned, create a socket with the *ai_family*, *ai_socktype*, and *ai_protocol* fields followed by calling *bind* with the *ai_addr* and *ai_addrlen* members.

The following code illustrates this principle.

```
SOCKET                          slisten[16];
char                            *szPort="5150";
struct addrinfo                           hints,
                                * res=NULL,
                                * ptr=NULL;
int                               count=0,
                                  rc;

memset(&hints, 0, sizeof(hints));
hints.ai_family = AF_UNSPEC;
hints.ai_socktype = SOCK_STREAM;
hints.ai_protocol = IPPROTO_TCP;
hints.ai_flags = AI_PASSIVE;
rc = getaddrinfo(NULL, szPort, &hints, &res);
if (rc != 0) {
                // failed for some reason
}
ptr = res;
while (ptr)
{
                slisten[count] = socket(ptr->ai_family,
                ptr->ai_socktype, ptr->ai_protocol);
                if (slisten[count] == INVALID_SOCKET) {
                        // socket failed
                }
                rc = bind(slisten[count], ptr->ai_addr, ptr->ai_addrlen);
                if (rc == SOCKET_ERROR) {
                        // bind failed
                }
                rc = listen(slisten[count], 7);
                if (rc == SOCKET_ERROR) {
                        // listen failed
                }
                count++;
                ptr = ptr->ai_next;
}
```

Once the sockets are created and bound, the application simply needs to wait for incoming connections on each. Chapter 5 covers the various I/O models available in Winsock and provides fully functioning client and server samples that are written using the principles covered in this section.

Conclusion

In this chapter, we've discussed both IPv4 and IPv6. We covered addressing, name resolution, as well as the necessary Winsock data structures for each address family. The new name resolution functions were described followed by the legacy name resolution functions. Finally, we covered how to write applications that work seamlessly over IPv4 and IPv6. In Chapter 4, we will cover the remaining protocols accessible from Winsock, including IPX/SPX, AppleTalk, IrDA, and ATM.

Chapter 4

Other Supported Protocols

To establish communication through Winsock, you must understand how to address a workstation using a particular protocol. In Chapter 3, we described how to do so using the IPv4 and IPv6 address families specifically. This chapter explains other protocols that Winsock supports—IrDA, IPX/SPX, NetBIOS, AppleTalk, and ATM—and how each protocol resolves an address specific to that family to an actual machine on the network. We'll cover only the basic knowledge necessary to form an address structure for each protocol family. Chapter 8 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 8 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. In addition, we will cover the protocol-specific options for name resolution. For each protocol discussed, we present basic client and server examples on the companion CD.

Infrared Sockets

Infrared sockets, or *IrSock*, are a 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, Windows Me, Windows 2000, and Windows XP. 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 that the use of 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 pair 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, *PSOCKADDR_IRDA, FAR *LPSOCKADDR_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 which 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 connect successfully, but user-friendly names require name resolution. For the most part, you shouldn't use the direct LSAP-SEL "address" because its address space for IrDA services is limited. The Windows 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 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 a 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 also 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 an LSAP-SEL address directly 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*, in which *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, Windows Me, Windows 2000, and Windows XP implementation. Windows CE supported *IrSock* before the other platforms and provided minimal information about infrared devices. Later, Windows 98, Windows Me, Windows 2000, and Windows XP provided support for *IrSock*, but they added additional "hint" information that the enumeration request returned. (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 using the *IRLMP_ENUM_DEVICES* command for *getsockopt*. A *DEVICELIST* structure is passed as the *optval* parameter. There are two structures, one for Windows 98, Windows Me, Windows 2000, and Windows XP 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_DEVICELIST;

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

The only difference between the non–Windows CE platforms structure and the Windows CE structure is that the non–Windows CE 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 we 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 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 five general steps for an *IrSock* server are as follows:

1. Create a socket of address family *AF_IRDA* and of 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 because you must enumerate infrared devices. The following four 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 with 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 if a given service is running on a particular device. The first method is to 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, Windows Me, Windows 2000, and Windows XP, 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
{
    u_char     irdaDeviceID[4];
    char       irdaClassName[61];
    char       irdaAttribName[61];
    u_short    irdaAttribType;
    union
    {
        int    irdaAttribInt;
        struct
        {
            int      Len;
            u_char   OctetSeq[1];
            u_char   Reserved[3];
        } irdaAttribOctetSeq;
        struct
        {
            int      Len;
            u_char   CharSet;
            u_char   UsrStr[1];
            u_char   Reserved[2];
        } irdaAttribUsrStr;
    } irdaAttribute;
} WCE_IAS_QUERY, *PWCE_IAS_QUERY;
```

As you can see, the two structure definitions are similar except for the lengths of certain character arrays.

Performing a query for the LSAP-SEL number of a particular service is simple: set the *irdaClassName* field to the property string for *LSAP-SELs,* which is “IrDA:IrLMP:LsapSel”, and set the *irdaAttributeName* field to the service name you want to query for. In addition, you have to set the *irdaDeviceID* with a valid device within range.

Creating a Socket

Creating an infrared socket is simple. Few options are required because *IrSock* supports only connection-oriented streams. The following code illustrates how to create an infrared socket using either the *socket* or *WSASocket* call. You must use *socket* for Windows CE because of its Winsock

1.1 limitation.

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

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

If you want to be specific, you can pass *IRDA_PROTO_SOCK_STREAM* as the protocol parameter of either function. However, the protocol parameter isn't required because the transport catalog has only one entry of address family *AF_IRDA*. Specifying *AF_IRDA* causes that transport entry to be used by default.

Socket Options

Many *SO_* socket options aren't meaningful to IrDA. Only *SO_LINGER and SO_DONTLINGER* are 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 7, which summarizes all socket options and their parameters.

> You will find a basic client and server IrSock application named IRCLIENT.CPP and IRSERVER.CPP, respectively, on the companion CD.

IPX/SPX

The IPX protocol is known as the protocol most often 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 Me, and Windows NT platforms but not on Windows CE.

Addressing

In an IPX network, network segments are bridged using an IPX router. Every network segment is assigned a unique four-byte network number. As more network segments are bridged, 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 six-byte node number, which is usually the network adapter's physical address. 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 four-byte number representing a network number of a network segment on an IPX network. The *sa_nodenum* field is a six-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 at the beginning of this section, 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 how sockets in TCP/IP behave. 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. If a machine has multiple network interfaces (a multihomed machine), it isn't necessary to bind to a specific interface. Windows 95, Windows 98, Windows Me, and Windows NT platforms provide a virtual internal network in which every 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 binds successfully 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_IPX))
    == SOCKET_ERROR)
{
    printf("bind failed with error %d\n",
        WSAGetLastError());
    return;
}

if (getsockname(sdServer, (PSOCKADDR) &IPXAddr, &addrlen)
    == 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 Me, and Windows NT platforms 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 Me, or Windows NT platforms, 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 because you must supply multi-byte 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 you will see in Chapter 8, Winsock 2 provides a protocol-independent method for name registration using the *WSASetService* API function. Through the SAP protocol, IPX server applications 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.

On the companion CD, you will find an IPX client and server application named SOCKSPX.CPP that demonstrates how to transmit datagrams over IPX or transmit reliable data communication over SPX.

NetBIOS

The NetBIOS address family is more of an unusual protocol family accessible from Winsock. NetBIOS itself is a network programming interface (instead of a network protocol) that can communicate over many network protocols, such as TCP/IP. Winsock interfaces with the NetBIOS programming interface through the NetBIOS address family. Addressing NetBIOS from Winsock requires that you know NetBIOS names and LANA numbers. You can familiarize yourself with many NetBIOS concepts with the Chapter 17 NetBIOS API discussion included on the companion CD. We won't discuss the entire core NetBIOS interface concepts here, instead, we'll continue with the specifics of accessing NetBIOS from Winsock.

> The NetBIOS address family is exposed by Winsock only on Windows NT platforms. It not available on Windows 95, Windows 98, or Windows Me platforms or on Windows CE.

Addressing

The basis for addressing a machine under NetBIOS is a NetBIOS name. 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, called *_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 you don't have to resolve a name into a physical address before performing any operations like you have to with TCP and IrDA. This becomes clear when you consider that NetBIOS is implemented over multiple protocols and each protocol has its own addressing scheme.

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. 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, -lana,
              NULL, 0, WSA_FLAG_OVERLAPPED);
```

The *type* parameter of *WSASocket* is either *SOCK_DGRAM* or *SOCK_SEQPACKET*, 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 is not used. Finally, the *dwFlags* parameter is set to *WSA_FLAG_ OVERLAPPED*; you should specify *WSA_FLAG_OVERLAPPED* on all calls to *WSASocket*. If you plan on using overlapped IO (as described in the next chapter), then this flag needs to be present when creating the socket.

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 as described in Chapter 17 to get the valid LANAs. Chapter 2 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
}
for (i = 0; i < dwNum; i++)
{
    // Look for those entries in the AF_NETBIOS address family
    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_OVERLAPPED);
        }
    }
}
```

In the pseudocode shown, 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 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.

On the companion CD you will find a NetBIOS client and server application named WSNBCLNT.CPP and WSNBSVR.CPP, respectively.

AppleTalk

AppleTalk support in Winsock has been around for a while, although few people are aware of it. You probably will not choose AppleTalk 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: name, type, and zone. Each name can be up to 32 characters long. 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 if 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 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—which include *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"
#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_ERROR)
{
    // Unable to open an endpoint on the AppleTalk network
}
if (setsockopt(s, SOL_APPLETALK, SO_REGISTER_NAME,
               (char *)&atname, sizeof(WSH_NBP_NAME)) == SOCKET_ERROR)
{
    // 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 Data Stream 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 although this call to *bind* allows you to perform simple actions on the network, by itself it doesn't 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 name is probably already in use. The Winsock error returned is *WSAEADDRINUSE* (10048). 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 Winsock calls use. 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. The following sample 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_ENTITY);
                    bFindName = TRUE;
                    break;
                case 't':        // Specify a type name
                    if (strlen(argv[i]) > 3)
                        strncpy(szType, &argv[i][3], MAX_ENTITY);
                    bFindName = TRUE;
                    break;
                 case 'o':        // Specify an object name
                    if (strlen(argv[i]) > 3)
                        strncpy(szObject, &argv[i][3], MAX_ENTITY);
                    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, szObject);
        atlookup->LookupTuple.NbpName.ObjectNameLen =
            strlen(szObject);
        strcpy(atlookup->LookupTuple.NbpName.TypeName, szType);
        atlookup->LookupTuple.NbpName.TypeNameLen = strlen(szType);
        strcpy(atlookup->LookupTuple.NbpName.ZoneName, szZone);
        atlookup->LookupTuple.NbpName.ZoneNameLen = strlen(szZone);
    }
    // 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 the
// 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, szZone);
    if (getsockopt(s, SOL_APPLETALK, SO_LOOKUP_NAME,
                (char *)atlookup, &dwSize) == INVALID_SOCKET)
    {
        printf("getsockopt(SO_LOOKUP_NAME) failed: %d\n",
            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_NAME
    // 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 buffer
    // for this option. Windows NT 4 SP3 blue screens if it
    // is too small.
    //
    if (getsockopt(s, SOL_APPLETALK, SO_LOOKUP_ZONES,
                (char *)atlookup, &dwSize) == INVALID_SOCKET)
    {
        printf("getsockopt(SO_LOOKUP_NAME) failed: %d\n",
            WSAGetLastError());
            return 1;
        }
        printf("Lookup returned: %d zones\n", zonelookup->NoZones);
        //
        // The character buffer contains a list of null-separated
        //  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
        //
        if (getsockopt(s, SOL_APPLETALK, SO_LOOKUP_MYZONE,
            (char *)cLookupBuffer, &dwSize) == INVALID_SOCKET)
        {
            printf("getsockopt(SO_LOOKUP_NAME) failed: %d\n",
                WSAGetLastError());
            return 1;
        }
        printf("My Zone: '%s'\n", cLookupBuffer);
    }
    else
        usage();
    WSACleanup();
    return 0;
}
```

When you are using most of the AppleTalk socket options—such as *SO_LOOKUP_MYZONE*, *SO_LOOKUP_ZONES*, and *SO_LOOKUP_NAME*—you need to provide a large character buffer to the *getsockopt* call. If you call an option that requires you to provide a structure, that structure needs to be at the start of the supplied character buffer. If the call to *getsockopt* is successful, the function places the returned data in the character buffer after the end of the supplied structure. Take a look at the *SO_LOOKUP_NAME* section in the above code sample. The variable, *cLookupBuffer*, is a simple character array used in the call to *getsockopt*. First, cast it as a *PWSH_LOOKUP_NAME* and fill in the name information you want to find. Pass the buffer into *getsockopt*, and upon return, increment the character pointer *pTupleBuffer* so that it points to the character after the end of the *WSH_LOOKUP_NAME* structure. Next, cast that pointer to a variable of *PWSH_NBP_TUPLE* because the data returned from a lookup name call is an array of *WSH_NBP_TUPLE* structures. Once you have the proper starting location and type of the tuples, you can walk through the array. Chapter 7 contains more in-depth information about the various socket options specific to the AppleTalk address family.

Creating a Socket

AppleTalk is available in Winsock 1.1 and later, so you can use either socket-creation routine. Again, you have two options of specifying the underlying AppleTalk protocols. First, you can supply the corresponding define from *atalkwsh.h* for the protocol you want, or you can enumerate the protocols using *WSAEnumProtocols* and passing the *WSAPROTOCOL_INFO* structure. Table 4-1 lists the required parameters for each AppleTalk protocol type when you create a socket directly using *socket* or *WSASocket*.

Table 4-1 *AppleTalk Protocols and Parameters*

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

On the companion CD, you will find an AppleTalk application named ATALK.CPP whic can operate as a sender or a receiver application over the PAP and ADSP protocols.

ATM

The Asynchronous Transfer Mode (ATM) protocol is one of the newest protocols available that is supported by Winsock 2 on Windows 98, Windows Me, Windows 2000, and Windows XP. 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 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 4-1—typically comprises endpoints (or computers) that are interconnected by switches that bridge an ATM network together.

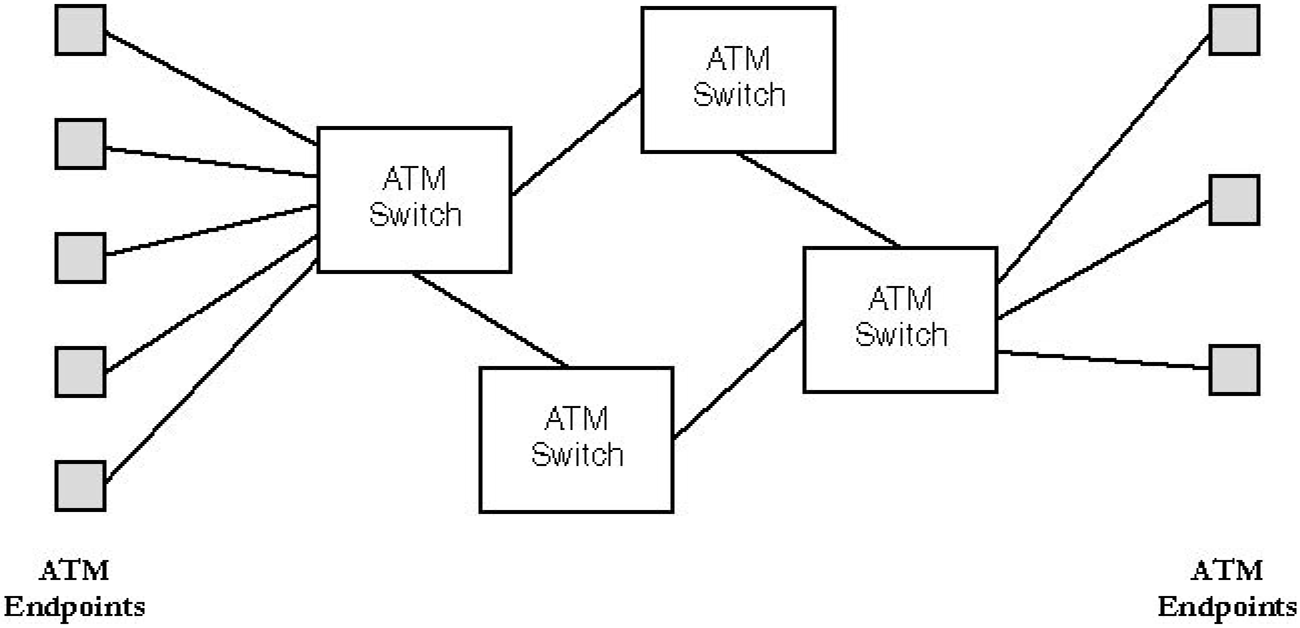


Figure 4-1 *ATM network*

There are a few points to be aware of when programming for the ATM protocol. First, ATM is a media type and not really a protocol. 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 in Chapter 3. Of course, ATM does offer some advantages over IP, such as a rooted multicast scheme (described in Chapter 9); however, the protocol that best suits you depends on your application's needs.

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. These are described here:

- 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 XP, Windows 2000, Windows Me, 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 an 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_ATM;
```

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. In addition, the *AddressType* field can be set to other values defined in Table 4-2 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 4-2 *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 either structure 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_SIZE);
```

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 tedious task. However, Winsock provides a protocol-independent API function, *WSAStringToAddress*, which can allow you to convert a 40-character ATM hexadecimal ASCII string to an *ATM_ADDRESS* structure. This API is described in Chapter 3. Another way to convert a hexadecimal ASCII string to hexadecimal (binary) format is to use the function *AtoH* defined in the following code. This function isn't part of Winsock. However, it is simple enough to develop, and you will see it used in the ATM sample (described later) included on the companion CD.

```
//
// 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;
}
```

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 2. 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 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_INFO,
            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 4-2. For an ATM socket, an SAP will be created once your application calls the Winsock *bind* API function (which we describe in Chapter 1), 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 which 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 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.

> On the companion CD, there is an ATM sample named WSOCKATM.CPP that serves as a client and server application and further demonstrates how to set up a socket that listen for connections on an SAP.

Finally, a Windows 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 Table 4-3 are available.

Table 4-3 *ATMADM.EXE Program Options*

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

Name Resolution

Currently, there are no name providers available for ATM under Winsock. Unfortunately, this requires applications to specify the 20-byte ATM address to establish socket communication over an ATM network. Chapter 8 discusses the Windows 2000 and Windows XP domain name space that can be generically used to register ATM addresses with user-friendly service names.

Conclusion

In this chapter, we described the remaining (non-IP) protocol address families that Winsock supports 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.

At this point in the book, we have completed our discussion of Winsock's basic communication techniques and have described all of the available address families that enable you to construct a simple Winsock application. Chapter 5 will start our discussion of advanced Winsock topics, and we will begin with advanced I/O methods that allow you to manage I/O in a Winsock application.

Chapter 5

Winsock I/O Methods

This chapter focuses on managing I/O in a Windows sockets application. Winsock features socket modes and socket I/O models to control how I/O is processed on a socket. A socket *mode* simply determines how Winsock functions behave when called with a socket. A socket *model*, on the other hand, describes how an application manages and processes I/O on a socket.

Winsock features two socket modes: *blocking* and *non-blocking*. The first part of this chapter describes these modes in detail and demonstrates how an application can use them to manage I/O. As you'll see later in the chapter, Winsock offers some interesting I/O models that help applications manage communication on one or more sockets at a time in an asynchronous fashion: *blocking*, *select*, *WSAAsyncSelect*, *WSAEventSelect*, *overlapped I/O*, and *completion port*. By the chapter's end, we'll review the pros and cons of the various socket modes and I/O models and help you decide which one best meets your application's needs.

All Windows platforms offer blocking and non-blocking socket operating modes. However, not every I/O model is available for every platform. As you can see in Table 5-1, only one of the I/O models is available under current versions of Windows CE. Windows 95, Windows 98, and Windows Me (depending on whether you have Winsock version 1 or 2) support most of the I/O models with the exception of I/O completion ports. Every I/O model is available on Windows NT and later versions.

Table 5-1 *Available Socket I/O Models*

Platform	Blocking	Non-blocking Select	*WSAAsync* Select	*WSAEvent* Select	Overlapped	Completion Port
Windows CE	Yes	Yes	No	No	No	No
Windows 95 (Winsock 1)	Yes	Yes	Yes	No	No	No
Windows 95 (Winsock 2)	Yes	Yes	Yes	Yes	Yes	No
Windows 98	Yes	Yes	Yes	Yes	Yes	No
Windows Me	Yes	Yes	Yes	Yes	Yes	No
Windows NT	Yes	Yes	Yes	Yes	Yes	Yes
Windows						

2000	Yes	Yes	Yes	Yes	Yes	Yes
Windows XP	Yes	Yes	Yes	Yes	Yes	Yes

Socket Modes

As we mentioned, Windows sockets perform I/O operations in two socket operating modes: blocking and non-blocking. In blocking mode, Winsock calls that perform I/O—such as *send* and *recv*—wait until the operation is complete before they return to the program. In non-blocking mode, the Winsock functions return immediately. Applications running on the Windows CE and Windows 95 (with Winsock 1) platforms, which support few of the I/O models, require you to take certain steps with blocking and non-blocking sockets to handle a variety of situations.

Blocking Mode

Blocking sockets cause concern because any Winsock API call on a blocking socket can do just that—block for some period of time. Most Winsock applications follow a producer-consumer model in which the application reads (or writes) a specified number of bytes and performs some computation on that data. The following code snippet illustrates this model:

```
SOCKET  sock;
char    buff[256];
int     done = 0,
        nBytes;

...

while(!done)
{
    nBytes = recv(sock, buff, 65);
    if (nBytes == SOCKET_ERROR)
     {
        printf("recv failed with error %d\n",
            WSAGetLastError());
        Return;
    }
    DoComputationOnData(buff);
}

...
```

The problem with this code is that the *recv* function might never return if no data is pending because the statement says to return only after reading some bytes from the system's input buffer. Some programmers might be tempted to peek for the necessary number of bytes in the system's buffer by using the *MSG_PEEK* flag in *recv* or by calling *ioctlsocket* with the *FIONREAD* option. Peeking for data without actually reading it is considered bad programming practice and should be avoided at all costs (reading the data actually removes it from the system's buffer). The overhead associated with peeking is great because one or more system calls are necessary just to check the number of bytes available. Then, of course, there is the overhead of making the *recv* call that removes the data from the system buffer. To avoid this, you need to prevent the application from totally freezing because of lack of data (either from network problems or from client problems) without continually peeking at the system network buffers. One method is to separate the application into a reading thread and a computation thread. Both threads share a common data buffer. Access to this buffer is protected with a synchronization object, such as an event or a mutex. The purpose of the reading thread is to continually read data from the network and place it in the shared buffer. When the reading thread has read the minimum amount of data necessary for the computation thread to do its work, it can signal an event that notifies the computation thread to begin. The computation thread then removes a chunk of data from the buffer and performs the necessary calculations.

The following section of code illustrates this approach by providing two functions: one responsible for reading network data (*ReadThread*) and one for performing the computations on the data (*ReadThread*).

```
#define MAX_BUFFER_SIZE    4096
// Initialize critical section (data) and create
// an auto-reset event (hEvent) before creating the
// two threads
CRITICAL_SECTION data;
HANDLE           hEvent;

SOCKET           sock;
TCHAR            buff[MAX_BUFFER_SIZE];
int              done=0;


// Create and connect sock
...
// Reader thread
void ReadThread(void)
{
    int nTotal = 0,
        nRead = 0,
        nLeft = 0,
        nBytes = 0;

    while (!done)
    {
        nTotal = 0;
        nLeft = NUM_BYTES_REQUIRED;

// However many bytes constitutes
// enough data for processing
// (i.e. non-zero)
        while (nTotal != NUM_BYTES_REQUIRED)
        {
            EnterCriticalSection(&data);
            nRead = recv(sock, &(buff[MAX_BUFFER_SIZE - nBytes]),
                nLeft, 0);
            if (nRead == -1)
            {
                printf("error\n");
                ExitThread();
            }
            nTotal += nRead;
            nLeft -= nRead;
            nBytes += nRead;
            LeaveCriticalSection(&data);
        }
        SetEvent(hEvent);
    }
}

// Computation thread
void ProcessThread(void)
{
    WaitForSingleObject(hEvent);

    EnterCriticalSection(&data);
    DoSomeComputationOnData(buff);

    // Remove the processed data from the input
    // buffer, and shift the remaining data to
    // the start of the array
    nBytes -= NUM_BYTES_REQUIRED;

    LeaveCriticalSection(&data);
}
```

One drawback of blocking sockets is that communicating via more than one connected socket at a time becomes difficult for the application. Using the foregoing scheme, the application could be modified to have a reading thread and a data processing thread per connected socket. This adds quite a bit of housekeeping overhead, but it is a feasible solution. The only drawback is that the solution does not scale well once you start dealing with a large number of sockets.

Non-blocking Mode

The alternative to blocking sockets is non-blocking sockets. Non-blocking sockets are a bit more challenging to use, but they are every bit as powerful as blocking sockets, with a few advantages. The following example illustrates how to create a socket and put it into non-blocking mode.

```
SOCKET        s;
unsigned long ul = 1;
int           nRet;

s = socket(AF_INET, SOCK_STREAM, 0);
nRet = ioctlsocket(s, FIONBIO, (unsigned long *) &ul);
if (nRet == SOCKET_ERROR)
{
    // Failed to put the socket into non-blocking mode
}
```

Once a socket is placed in non-blocking mode, Winsock API calls that deal with sending and receiving data or connection management return immediately. In most cases, these calls fail with the error *WSAEWOULDBLOCK*, which means that the requested operation did not have time to complete during the call. For example, a call to *recv* returns *WSAEWOULDBLOCK* if no data is pending in the system's input buffer. Often additional calls to the same function are required until it encounters a successful return code. Table 5-2 describes the meaning of *WSAEWOULDBLOCK* when returned by commonly used Winsock calls.

Table 5-2 *WSAEWOULDBLOCK Errors on Non-blocking Sockets*

Function Name	Description
WSAAccept and *accept*	The application has not received a connection request. Call again to check for a connection.
closesocket	In most cases, this means that *setsockopt* was called with the *SO_LINGER* option and a nonzero timeout was set.
WSAConnect and *connect*	The connection is initiated. Call again to check for completion.
WSARecv, recv, WSARecvFrom, and *recvfrom*	No data has been received. Check again later.
WSASend, send, WSASendTo, and *sendto*	No buffer space available for outgoing data. Try again later.

Because non-blocking calls frequently fail with the *WSAEWOULDBLOCK* error, you should check all return codes and be prepared for failure at any time. The trap many programmers fall into is that of continually calling a function until it returns a success. For example, placing a call to *recv* in a tight loop to read 200 bytes of data is no better than polling a blocking socket with the *MSG_PEEK* flag mentioned previously. Winsock's socket I/O models can help an application determine when a socket is available for reading and writing.

Each socket mode—blocking and non-blocking—has advantages and disadvantages. Blocking sockets are easier to use from a conceptual standpoint but become difficult to manage when dealing with multiple connected sockets or when data is sent and received in varying amounts and at arbitrary times. On the other hand, non-blocking sockets are more difficult because more code needs to be written to handle the possibility of receiving a *WSAEWOULDBLOCK* error on every Winsock call. Socket I/O models help applications manage communications on one or more sockets at a time in an asynchronous fashion.

Socket I/O Models

Essentially, six types of socket I/O models are available that allow Winsock applications to manage I/O: *blocking, select, WSAAsyncSelect, WSAEventSelect, overlapped,* and *completion port*. This section explains the features of each I/O model and outlines how to use it to develop an application that can manage one or more socket requests. On the companion CD, you will find sample applications for each I/O model demonstrating how to develop a simple TCP echo server using the principles described in each model.

Note that technically speaking, there could be a straight non-blocking I/O model—that is, an application that places all sockets into non-blocking mode with *ioctlsocket*. However, this soon becomes unmanageable because the application will spend most of its time cycling through socket handles and I/O operations until they succeed.

The *blocking* Model

Most Winsock programmers begin with the blocking model because it is the easiest and most straightforward model. The Winsock samples in Chapter 1 use this model. As we have mentioned, applications following this model typically use one or two threads per socket connection for handling I/O. Each thread will then issue blocking operations, such as *send* and *recv*.

The advantage to the blocking model is its simplicity. For very simple applications and rapid prototyping, this model is very useful. The disadvantage is that it does not scale up to many connections as the creation of more threads consumes valuable system resources.

The *select* Model

The *select* model is another I/O model widely available in Winsock. We call it the *select* model because it centers on using the *select* function to manage I/O. The design of this model originated on UNIX-based computers featuring Berkeley socket implementations. The *select* model was incorporated into Winsock 1.1 to allow applications that want to avoid blocking on socket calls the capability to manage multiple sockets in an organized manner. Because Winsock 1.1 is backward-compatible with Berkeley socket implementations, a Berkeley socket application that uses the *select* function should technically be able to run without modification.

The *select* function can be used to determine if there is data on a socket and if a socket can be written to. The reason for having this function is to prevent your application from blocking on an I/O bound call such as *send* or *recv* when a socket is in a blocking mode and to prevent the *WSAEWOULDBLOCK* error when a socket is in a non-blocking mode. The *select* function blocks for I/O operations until the conditions specified as parameters are met. The function prototype for *select* is as follows:

```
int select(
    int nfds,
    fd_set FAR * readfds,
    fd_set FAR * writefds,
    fd_set FAR * exceptfds,
    const struct timeval FAR * timeout
);
```

The first parameter, *nfds*, is ignored and is included only for compatibility with Berkeley socket applications. You'll notice that there are three *fd_set* parameters: one for checking readability (*readfds*), one for writeability (*writefds*), and one for out-of-band data (*exceptfds*). Essentially, the *fd_set* data type represents a collection of sockets. The *readfds* set identifies sockets that meet one of the following conditions:

- Data is available for reading.
- Connection has been closed, reset, or terminated.
- If *listen* has been called and a connection is pending, the *accept* function will succeed.

The *writefds* set identifies sockets in which one of the following is true:

- Data can be sent.
- If a non-blocking connect call is being processed, the connection has succeeded.

Finally, the *exceptfds* set identifies sockets in which one of the following is true:

- If a non-blocking connect call is being processed, the connection attempt failed.
- OOB data is available for reading.

For example, when you want to test a socket for readability, you must add it to the *readfds* set and wait for the *select* function to complete. When the *select* call completes, you have to determine if your socket is still part of the *readfds* set. If so, the socket is readable—you can begin to retrieve data from it. Any two of the three parameters (*readfds*, *writefds*, *exceptfds*) can be null values (at least one must not be null), and any non-null set must contain at least one socket handle; otherwise, the *select* function won't have anything to wait for. The final parameter, *timeout*, is a pointer to a *timeval* structure that determines how long the *select* function will wait for I/O to complete. If *timeout* is a null pointer, *select* will block indefinitely until at least one descriptor meets the specified criteria. The *timeval* structure is defined as

```
struct timeval
{
    long tv_sec;
    long tv_usec;
};
```

The *tv_sec* field indicates how long to wait in seconds; the *tv_usec* field indicates how long to wait in milliseconds. The timeout value {0, 0} indicates *select* will return immediately, allowing an application to poll on the *select* operation. This should be avoided for performance reasons. When *select* completes successfully, it returns the total number of socket handles that have I/O operations pending in the *fd_set* structures. If the *timeval* limit expires, it returns 0. If *select* fails for any reason, it returns *SOCKET_ERROR*.

Before you can begin to use *select* to monitor sockets, your application has to set up either one or all of the read, write, and exception *fd_set* structures by assigning socket handles to a set. When you assign a socket to one of the sets, you are asking *select* to let you know if the I/O activities just described have occurred on a socket. Winsock provides the following set of macros to manipulate and check the *fd_set* sets for I/O activity.

- FD_ZERO(*set) Initializes *set* to the empty set. A set should always be cleared before using.
- FD_CLR(s, *set) Removes socket *s* from *set*.
- FD_ISSET(s, *set) Checks to see if *s* is a member of *set* and returns *TRUE* if so.
- FD_SET(s, *set) Adds socket *s* to *set*.

For example, if you want to find out when it is safe to read data from a socket without blocking, simply assign your socket to the *fd_read* set using the *FD_SET* macro and then call *select*. To test whether your socket is still part of the *fd_read* set, use the *FD_ISSET* macro. The following five steps describe the basic flow of an application that uses *select* with one or more socket handles:

1. Initialize each *fd_set* of interest by using the *FD_ZERO* macro.
2. Assign socket handles to each of the *fd_set* sets of interest by using the *FD_SET* macro.
3. Call the *select* function and wait until I/O activity sets one or more of the socket handles in each *fd_set* set provided. When *select* completes, it returns the total number of socket handles that are set in all of the *fd_set* sets and updates each set accordingly.
4. Using the return value of *select*, your application can determine which application sockets have I/O pending by checking each *fd_set* set using the *FD_ISSET* macro.
5. After determining which sockets have I/O pending in each of the sets, process the I/O and go to step 1 to continue the *select* process.

When select returns, it modifies each of the *fd_set* structures by removing the socket handles that do not have pending I/O operations. This is why you should use the *FD_ISSET* macro as in step 4 to determine if a particular socket is part of a set. The following code sample outlines the basic steps needed to set up the select model for a single socket. Adding more sockets to this application simply involves maintaining a list or an array of additional sockets.

```
SOCKET  s;
fd_set  fdread;
int     ret;

// Create a socket, and accept a connection

// Manage I/O on the socket
while(TRUE)
{
    // Always clear the read set before calling
    // select()
    FD_ZERO(&fdread);

    // Add socket s to the read set
    FD_SET(s, &fdread);
    if ((ret = select(0, &fdread, NULL, NULL, NULL))
        == SOCKET_ERROR)
    {
        // Error condition
    }

    if (ret > 0)
    {
        // For this simple case, select() should return
        // the value 1. An application dealing with
        // more than one socket could get a value
        // greater than 1. At this point, your
        // application should check to see whether the
        // socket is part of a set.

        if (FD_ISSET(s, &fdread))
        {
            // A read event has occurred on socket s
        }
    }
}
```

The advantage of using *select* is the capability to multiplex connections and I/O on many sockets from a single thread. This prevents the explosion of threads associated with blocking sockets and multiple connections. The disadvantage is the maximum number of sockets that may be added to the *fd_set* structures. By default, the maximum is defined as *FD_SETSIZE*, which is defined in WINSOCK2.H as 64. To increase this limit, an application might define *FD_SETSIZE* to something large. This define must appear before including WINSOCK2.H. Also, the underlying provider imposes an arbitrary maximum *fd_set* size, which typically is 1024 but is not guaranteed to be. Finally, for a large *FD_SETSIZE*, consider the performance hit of setting 1000 sockets before calling *select* followed by checking whether each of those 1000 sockets is set after the call returns.

The *WSAAsyncSelect* Model

Winsock provides a useful asynchronous I/O model that allows an application to receive Windows message–based notification of network events on a socket. This is accomplished by calling the *WSAAsyncSelect* function after creating a socket. Before we continue, however, we need to make one subtle distinction. The *WSAAsyncSelect* and *WSAEventSelect* models provide asynchronous notification of the capability to read or write data. It does not provide asynchronous data transfer like the overlapped and completion port models.

This model originally existed in Winsock 1.1 implementations to help application programmers cope with the cooperative multitasking message-based environment of 16-bit Windows platforms, such as Windows for Workgroups. Applications can still benefit from this model, especially if they manage window messages in a standard Windows procedure, usually referred to as a *winproc*. This model is also used by the Microsoft Foundation Class (MFC) *CSocket* object.

Message Notification

To use the *WSAAsyncSelect* model, your application must first create a window using the *CreateWindow* function and supply a window procedure (*winproc*) support function for it. You can also use a dialog box with a dialog procedure instead of a window because dialog boxes ***are*** windows. For our purposes, we will demonstrate this model using a simple window with a supporting window procedure. Once you have set up the window infrastructure, you can begin creating sockets and turning on window message notification by calling the *WSAAsyncSelect* function, which is defined as

```
 int WSAAsyncSelect(
    SOCKET s,
    HWND hWnd,
    unsigned int wMsg,
    long lEvent
);
```

The *s* parameter represents the socket we are interested in. The *hWnd* parameter is a window handle identifying the window or the dialog box that receives a message when a network event occurs. The *wMsg* parameter identifies the message to be received when a network event occurs. This message is posted to the window that is identified by the *hWnd* window handle. Applications usually set this message to a value greater than the Windows *WM_USER* value to avoid confusing a network window message with a predefined standard window message. The last parameter, *lEvent*, represents a bitmask that specifies a combination of network events—listed in Table 5-3—that the application is interested in. Most applications are typically interested in the *FD_READ, FD_WRITE, FD_ACCEPT, FD_CONNECT,* and *FD_CLOSE* network event types. Of course, the use of the *FD_ACCEPT* or the *FD_CONNECT* type depends on whether your application is a client or a server. If your application is interested in more than one network event, simply set this field by performing a bitwise *OR* on the types and assigning them to *lEvent*. For example:

```
WSAAsyncSelect(s, hwnd, WM_SOCKET,
    FD_CONNECT | FD_READ | FD_WRITE | FD_CLOSE);
```

This allows our application to get connect, send, receive, and socket-closure network event notifications on socket *s*. It is impossible to register multiple events one at a time on the socket. Also note that once you turn on event notification on a socket, it remains on unless the socket is closed by a call to *closesocket* or the application changes the registered network event types by calling *WSAAsyncSelect* (again, on the socket). Setting the *lEvent* parameter to 0 effectively stops all network event notification on the socket.

When your application calls *WSAAsyncSelect* on a socket, the socket mode is automatically changed from blocking to the non-blocking mode that we described previously. As a result, if a Winsock I/O call such as *WSARecv* is called and has to wait for data, it will fail with error *WSAEWOULDBLOCK*. To avoid this error, applications should rely on the user-defined window message specified in the *wMsg* parameter of *WSAAsyncSelect* to indicate when network event types occur on the socket.

Table 5-3 *Network Event Types for the WSAAsyncSelect Function*

Event Type	Meaning
FD_READ	The application wants to receive notification of readiness

	for reading.
FD_WRITE	The application wants to receive notification of readiness for writing.
FD_OOB	The application wants to receive notification of the arrival of OOB data.
FD_ACCEPT	The application wants to receive notification of incoming connections.
FD_CONNECT	The application wants to receive notification of a completed connection or a multipoint *join* operation.
FD_CLOSE	The application wants to receive notification of socket closure.
FD_QOS	The application wants to receive notification of socket QOS changes.
FD_GROUP_QOS	The application wants to receive notification of socket group QOS changes (reserved for future use with socket groups).
FD_ROUTING_INTERFACE_CHANGE	The application wants to receive notification of routing interface changes for the specified destination(s).
FD_ADDRESS_LIST_CHANGE	The application wants to receive notification of local address list changes for the socket's protocol family.

After your application successfully calls *WSAAsyncSelect* on a socket, the application begins to receive network event notification as Windows messages in the window procedure associated with the *hWnd* parameter window handle. A window procedure is normally defined as

```
LRESULT CALLBACK WindowProc(
    HWND hWnd,
    UINT uMsg,
    WPARAM wParam,
    LPARAM lParam
);
```

The *hWnd* parameter is a handle to the window that invoked the window procedure. The *uMsg* parameter indicates which message needs to be processed. In your case, you will be looking for the message defined in the *WSAAsyncSelect* call. The *wParam* parameter identifies the socket on which a network event has occurred. This is important if you have more than one socket assigned to this window procedure. The *lParam* parameter contains two important pieces of information—the low word of *lParam* specifies the network event that has occurred, and the high word of *lParam* contains any error code.

When network event messages arrive at a window procedure, the application should first check the *lParam* high-word bits to determine whether a network error has occurred on the socket. There is a special macro, *WSAGETSELECTERROR*, that returns the value of the high-word bits error information. After the application has verified that no error occurred on the socket, the application should determine which network event type caused the Windows message to fire by reading the low-word bits of *lParam*. Another special macro, *WSAGETSELECTEVENT*, returns the value of the low-word portion of *lParam*.

The following example demonstrates how to manage window messages when using the *WSAAsyncSelect* I/O model. The code highlights the steps needed to develop a basic server application and removes the programming details of developing a fully featured Windows application.

```
#define WM_SOCKET WM_USER + 1
#include <winsock2.h>
#include <windows.h>

int WINAPI WinMain(HINSTANCE hInstance,
    HINSTANCE hPrevInstance, LPSTR lpCmdLine,
    int nCmdShow)
{
    WSADATA wsd;
    SOCKET Listen;
    SOCKADDR_IN InternetAddr;
    HWND Window;

    // Create a window and assign the ServerWinProc
    // below to it

    Window = CreateWindow();
    // Start Winsock and create a socket

    WSAStartup(MAKEWORD(2,2), &wsd);
    Listen = socket (AF_INET, SOCK_STREAM, IPPROTO_TCP);
    // Bind the socket to port 5150
    // and begin listening for connections

    InternetAddr.sin_family = AF_INET;
    InternetAddr.sin_addr.s_addr = htonl(INADDR_ANY);
    InternetAddr.sin_port = htons(5150);

    bind(Listen, (PSOCKADDR) &InternetAddr,
        sizeof(InternetAddr));

    // Set up window message notification on
    // the new socket using the WM_SOCKET define
    // above

    WSAAsyncSelect(Listen, Window, WM_SOCKET,
        FD_ACCEPT | FD_CLOSE);

    listen(Listen, 5);

    // Translate and dispatch window messages
    // until the application terminates
    while (1) {
     // ...
 }
}

BOOL CALLBACK ServerWinProc(HWND hDlg,UINT wMsg,
    WPARAM wParam, LPARAM lParam)
```

```
{
    SOCKET Accept;

    switch(wMsg)
    {
        case WM_PAINT:
            // Process window paint messages
            break;

        case WM_SOCKET:

            // Determine whether an error occurred on the
            // socket by using the WSAGETSELECTERROR() macro

            if (WSAGETSELECTERROR(lParam))
            {
                 // Display the error and close the socket
                closesocket( (SOCKET) wParam);
                break;
            }

            // Determine what event occurred on the
            // socket

            switch(WSAGETSELECTEVENT(lParam))
            {
                case FD_ACCEPT:

                    // Accept an incoming connection
                    Accept = accept(wParam, NULL, NULL);

                    // Prepare accepted socket for read,
                    // write, and close notification

                    WSAAsyncSelect(Accept, hDlg, WM_SOCKET,
                        FD_READ | FD_WRITE | FD_CLOSE);
                    break;

                case FD_READ:

                    // Receive data from the socket in
                    // wParam
                    break;

                case FD_WRITE:

                    // The socket in wParam is ready
                    // for sending data
                    break;

                case FD_CLOSE:

                    // The connection is now closed
                    closesocket( (SOCKET)wParam);
                    break;
            }
            break;
    }
    return TRUE;
}
```

One final detail worth noting is how applications should process *FD_WRITE* event notifications. *FD_WRITE* notifications are sent under only three conditions:

- After a socket is first connected with connect or *WSAConnect*
- After a socket is accepted with accept or *WSAAccept*
- When a send, *WSASend*, *sendto*, or *WSASendTo* operation fails with *WSAEWOULDBLOCK* and buffer space becomes available

Therefore, an application should assume that sends are always possible on a socket starting from the first *FD_WRITE* message and lasting until a *send*, *WSASend*, *sendto*, or *WSASendTo* returns the socket error *WSAEWOULDBLOCK*. After such failure, another *FD_WRITE* message notifies the application that sends are once again possible.

The *WSAAsyncSelect* model offers many advantages; foremost is the capability to handle many connections simultaneously without much overhead, unlike the select model's requirement of setting up the *fd_set* structures. The disadvantages are having to use a window if your application requires no windows (such as a service or console application). Also, having a single window procedure to service all the events on thousands of socket handles can become a performance bottleneck (meaning this model doesn't scale very well).

The *WSAEventSelect* Model

Winsock provides another useful asynchronous event notification I/O model that is similar to the *WSAAsyncSelect* model that allows an application to receive event-based notification of network events on one or more sockets. This model is similar to the *WSAAsyncSelect* model because your application receives and processes the same network events listed in Table 5-3 that the *WSAAsyncSelect* model uses. The major difference with this model is that network events are notified via an event object handle instead of a window procedure.

Event Notification

The event notification model requires your application to create an event object for each socket used by calling the *WSACreateEvent* function, which is defined as

```
WSAEVENT WSACreateEvent(void);
```

The *WSACreateEvent* function simply returns a manual reset event object handle. Once you have an event object handle, you have to associate it with a socket and register the network event types of interest, as shown in Table 5-3. This is accomplished by calling the *WSAEventSelect* function, which is defined as

```
int WSAEventSelect(
    SOCKET s,
    WSAEVENT hEventObject,
    long lNetworkEvents
);
```

The *s* parameter represents the socket of interest. The *hEventObject* parameter represents the event object—obtained with *WSACreateEvent*—to associate with the socket. The last parameter, *lNetworkEvents*, represents a bitmask that specifies a combination of network event types (listed in Table 5-3) that the application is interested in. For a detailed discussion of these event types, see the *WSAAsyncSelect* I/O model discussed previously.

The event created for *WSAEventSelect* has two operating states and two operating modes. The operating states are known as *signaled* and *non-signaled*. The operating modes are known as *manual reset* and *auto reset*. *WSACreateEvent* initially creates event handles in a non-signaled operating state with a manual reset operating mode. As network events trigger an event object associated with a socket, the operating state changes from non-signaled to signaled. Because the event object is created in a manual reset mode, your application is responsible for changing the operating state from signaled to non-signaled after processing an I/O request. This can be accomplished by calling the *WSAResetEvent* function, which is defined as

```
BOOL WSAResetEvent(WSAEVENT hEvent);
```

The function takes an event handle as its only parameter and returns *TRUE* or *FALSE* based on the success or failure of the call. When an application is finished with an event object, it should call the *WSACloseEvent* function to free the system resources used by an event handle. The *WSACloseEvent* function is defined as

```
BOOL WSACloseEvent(WSAEVENT hEvent);
```

This function also takes an event handle as its only parameter and returns *TRUE* if successful or *FALSE* if the call fails.

Once a socket is associated with an event object handle, the application can begin processing I/O by waiting for network events to trigger the operating state of the event object handle. The *WSAWaitForMultipleEvents* function is designed to wait on one or more event object handles and returns either when one or all of the specified handles are in the signaled state or when a specified timeout interval expires. *WSAWaitForMultipleEvents* is defined as

```
DWORD WSAWaitForMultipleEvents(
    DWORD cEvents,
    const WSAEVENT FAR * lphEvents,
    BOOL fWaitAll,
    DWORD dwTimeout,
    BOOL fAlertable
);
```

The *cEvents* and *lphEvents* parameters define an array of *WSAEVENT* objects in which *cEvents* is the number of event objects in the array and *lphEvents* is a pointer to the array. *WSAWaitForMultipleEvents* can support only a maximum of *WSA_MAXIMUM_WAIT_EVENTS* objects, which is defined as 64. Therefore, this I/O model is capable of supporting only a maximum of 64 sockets at a time for each thread that makes the *WSAWaitForMultipleEvents* call. If you need to have this model manage more than 64 sockets, you should create additional worker threads to wait on more event objects. The *fWaitAll* parameter specifies how *WSAWaitForMultipleEvents* waits for objects in the event array. If *TRUE*, the function returns when all event objects in the *lphEvents* array are signaled. If *FALSE*, the function returns when any one of the event objects is signaled. In the latter case, the return value indicates which event object caused the function to return. Typically, applications set this parameter to *FALSE* and service one socket event at a time. The *dwTimeout* parameter specifies how long (in milliseconds) *WSAWaitForMultipleEvents* will wait for a network event to occur. The function returns if the interval expires, even if conditions specified by the *fWaitAll* parameter are not satisfied. If the timeout value is 0, the function tests the state of the specified event objects and returns immediately, which effectively allows an application to poll on the event objects. If no events are ready for processing, *WSAWaitForMultipleEvents* returns *WSA_WAIT_TIMEOUT*. If *dwsTimeout* is set to *WSA_INFINITE*, the function returns only when a network event signals an event object. The final parameter, *fAlertable*, can be ignored when you're using the *WSAEventSelect* model and should be set to *FALSE*. It is intended for use in processing completion routines in the overlapped I/O model, which will be described later in this chapter.

Note that by servicing signaled events one at a time (by setting the *fWaitAll* parameter to *FALSE*), it is possible to starve sockets toward the end of the event array. Consider the following code:

```
WSAEVENT                                HandleArray[WSA_MAXIMUM_WAIT_EVENTS];
int                             WaitCount=0, ret, index;

// Assign event handles into HandleArray
while (1) {
                ret = WSAWaitForMultipleEvents(
                        WaitCount,
                        HandleArray,
                        FALSE,
                        WSA_INFINITE,
                        TRUE);
    if ((ret != WSA_WAIT_FAILED) && (ret != WSA_WAIT_TIMEOUT)) {
                index = ret - WSA_WAIT_OBJECT_0;
                        // Service event signaled on HandleArray[index]
                        WSAResetEvent(HandleArray[index]);
                }
}
```

If the socket connection associated in index 0 of the event array is continually receiving data such that after the event is reset additional data arrives causing the event to be signaled again, the rest of the events in the array are starved. This is clearly undesirable. Once an event within the loop is signaled and handled, all events in the array should be checked to see if they are signaled as well. This can be accomplished by using *WSAWaitForMultipleEvents* with each individual event handle after the first signaled event and specifying a *dwTimeOut* of zero.

When *WSAWaitForMultipleEvents* receives network event notification of an event object, it returns a value indicating the event object that caused the function to return. As a result, your application can determine which network event type is available on a particular socket by referencing the signaled event in the event array and matching it with the socket associated with the event. When you reference the events in the event array, you should reference them using the return value of

WSAWaitForMultipleEvents minus the predefined value *WSA_WAIT_EVENT_0*. For example:

```
Index = WSAWaitForMultipleEvents(...);
MyEvent = EventArray[Index - WSA_WAIT_EVENT_0];
```

Once you have the socket that caused the network event, you can determine which network events are available by calling the *WSAEnumNetworkEvents* function, which is defined as

```
int WSAEnumNetworkEvents(
    SOCKET s,
    WSAEVENT hEventObject,
    LPWSANETWORKEVENTS lpNetworkEvents
);
```

The *s* parameter represents the socket that caused the network event, and the *hEventObject* parameter is an optional parameter representing an event handle identifying an associated event object to be reset. Because our event object is in a signaled state, we can pass it in and it will be set to a non-signaled state. The *hEventObject* parameter is optional in case you wish to reset the event manually via the *WSAResetEvent* function. The final parameter, *lpNetworkEvents*, takes a pointer to a *WSANETWORKEVENTS* structure, which is used to retrieve network event types that occurred on the socket and any associated error codes. The *WSANETWORKEVENTS* structure is defined as

```
typedef struct _WSANETWORKEVENTS
{
    long lNetworkEvents;
    int  iErrorCode[FD_MAX_EVENTS];
} WSANETWORKEVENTS, FAR * LPWSANETWORKEVENTS;
```

The *lNetworkEvents* parameter is a value that indicates all the network event types (see Table 5-3) that have occurred on the socket.

More than one network event type can occur whenever an event is signaled. For example a busy server application might receive *FD_READ* and *FD_WRITE* notification at the same time.

The *iErrorCode* parameter is an array of error codes associated with the events in *lNetworkEvents*. For each network event type, there is a special event index similar to the event type names—except for an additional "_BIT" string appended to the event name. For example, for the *FD_READ* event type, the index identifier for the *iErrorCode* array is named *FD_READ_BIT*. The following code fragment demonstrates this for an *FD_READ* event:

```
// Process FD_READ notification
if (NetworkEvents.lNetworkEvents & FD_READ)
{
    if (NetworkEvents.iErrorCode[FD_READ_BIT] != 0)
    {
       printf("FD_READ failed with error %d\n",
            NetworkEvents.iErrorCode[FD_READ_BIT]);
    }
}
```

After you process the events in the *WSANETWORKEVENTS* structure, your application should continue waiting for more network events on all of the available sockets. The following example demonstrates how to develop a server and manage event objects when using the *WSAEventSelect* I/O model. The code highlights the steps needed to develop a basic server application capable of managing one or more sockets at a time.

```
SOCKET SocketArray [WSA_MAXIMUM_WAIT_EVENTS];
WSAEVENT EventArray [WSA_MAXIMUM_WAIT_EVENTS],
         NewEvent;
SOCKADDR_IN InternetAddr;
SOCKET Accept, Listen;
DWORD EventTotal = 0;
DWORD Index, i;
// Set up a TCP socket for listening on port 5150
Listen = socket (PF_INET, SOCK_STREAM, 0);

InternetAddr.sin_family = AF_INET;
InternetAddr.sin_addr.s_addr = htonl(INADDR_ANY);
InternetAddr.sin_port = htons(5150);

bind(Listen, (PSOCKADDR) &InternetAddr,
    sizeof(InternetAddr));

NewEvent = WSACreateEvent();

WSAEventSelect(Listen, NewEvent,
    FD_ACCEPT | FD_CLOSE);

listen(Listen, 5);

SocketArray[EventTotal] = Listen;
EventArray[EventTotal] = NewEvent;
EventTotal++;

while(TRUE)
{
    // Wait for network events on all sockets
    Index = WSAWaitForMultipleEvents(EventTotal,
        EventArray, FALSE, WSA_INFINITE, FALSE);
    Index = Index - WSA_WAIT_EVENT_0;

    // Iterate through all events to see if more than one is signaled
    for(i=Index; i < EventTotal ;i++
    {
     Index = WSAWaitForMultipleEvents(1, &EventArray[i], TRUE, 1000,
      FALSE);
     if ((Index == WSA_WAIT_FAILED) || (Index == WSA_WAIT_TIMEOUT))
         continue;
     else
     {
         Index = i;
         WSAEnumNetworkEvents(
             SocketArray[Index],
             EventArray[Index],
            &NetworkEvents);

         // Check for FD_ACCEPT messages
         if (NetworkEvents.lNetworkEvents & FD_ACCEPT)
         {
             if (NetworkEvents.iErrorCode[FD_ACCEPT_BIT] != 0)
             {
                 printf("FD_ACCEPT failed with error %d\n",
                     NetworkEvents.iErrorCode[FD_ACCEPT_BIT]);
                 break;
             }

             // Accept a new connection, and add it to the
             // socket and event lists
             Accept = accept(
                 SocketArray[Index],
                 NULL, NULL);

             // We cannot process more than
             // WSA_MAXIMUM_WAIT_EVENTS sockets, so close
             // the accepted socket
             if (EventTotal > WSA_MAXIMUM_WAIT_EVENTS)
             {
                 printf("Too many connections");
                 closesocket(Accept);
                 break;
             }

             NewEvent = WSACreateEvent();

             WSAEventSelect(Accept, NewEvent,
                 FD_READ | FD_WRITE | FD_CLOSE);

             EventArray[EventTotal] = NewEvent;
             SocketArray[EventTotal] = Accept;
             EventTotal++;
             printf("Socket %d connected\n", Accept);
         }

         // Process FD_READ notification
         if (NetworkEvents.lNetworkEvents & FD_READ)
         {
             if (NetworkEvents.iErrorCode[FD_READ_BIT] != 0)
             {
                 printf("FD_READ failed with error %d\n",
                     NetworkEvents.iErrorCode[FD_READ_BIT]);
                 break;
             }

             // Read data from the socket
             recv(SocketArray[Index - WSA_WAIT_EVENT_0],
                 buffer, sizeof(buffer), 0);
         }

         // Process FD_WRITE notification
         if (NetworkEvents.lNetworkEvents & FD_WRITE)
```

```
         {
             if (NetworkEvents.iErrorCode[FD_WRITE_BIT] != 0)
             {
                 printf("FD_WRITE failed with error %d\n",
                     NetworkEvents.iErrorCode[FD_WRITE_BIT]);
                 break;
             }

             send(SocketArray[Index - WSA_WAIT_EVENT_0],
                buffer, sizeof(buffer), 0);
            }

            if (NetworkEvents.lNetworkEvents & FD_CLOSE)
            {
                if (NetworkEvents.iErrorCode[FD_CLOSE_BIT] != 0)
                {
                    printf("FD_CLOSE failed with error %d\n",
                        NetworkEvents.iErrorCode[FD_CLOSE_BIT]);
                    break;
                }

                closesocket(SocketArray[Index]);

                // Remove socket and associated event from
                // the Socket and Event arrays and decrement
                // EventTotal
                CompressArrays(EventArray, SocketArray, &EventTotal);
            }
                }
    }
}
```

The *WSAEventSelect* model offers several advantages. It is conceptually simple and it does not require a windowed environment. The only drawback is its limitation of waiting on only 64 events at a time, which necessitates managing a thread pool when dealing with many sockets. Also, because many threads are required to handle a large number of socket connections, this model does not scale as well as the overlapped models discussed next.

The Overlapped Model

The overlapped I/O model in Winsock offers applications better system performance than any of the I/O models explained so far. The overlapped model's basic design allows your application to post one or more asynchronous I/O requests at a time using an overlapped data structure. At a later point, the application can service the submitted requests after they have completed. This model is available on all Windows platforms except Windows CE. The model's overall design is based on the Windows overlapped I/O mechanisms available for performing I/O operations on devices using the *ReadFile* and *WriteFile* functions.

Originally, the Winsock overlapped I/O model was available only to Winsock 1.1 applications running on Windows NT. Applications could take advantage of the model by calling *ReadFile* and *WriteFile* on a socket handle and specifying an overlapped structure. Since the release of Winsock 2, overlapped I/O has been incorporated into new Winsock functions, such as *WSASend* and *WSARecv*. As a result, the overlapped I/O model is now available on all Windows platforms that feature Winsock 2.

> With the release of Winsock 2, overlapped I/O can still be used with the functions *ReadFile* and *WriteFile* under Windows NT and Windows 2000. However, this functionality was not added to Windows 95, Windows 98, and Windows Me. For compatibility across platforms, you should always consider using the *WSARecv* and *WSASend* functions instead of the Windows *ReadFile* and *WriteFile* functions. This section will only describe how to use overlapped I/O through the Winsock 2 functions.

To use the overlapped I/O model on a socket, you must first create a socket that has the overlapped flag set. See Chapter 2 for more information on creating overlapped enabled sockets.

After you successfully create a socket and bind it to a local interface, overlapped I/O operations can commence by calling the Winsock functions listed below and specifying an optional *WSAOVERLAPPED* structure.

- *WSASend*
- *WSASendTo*
- *WSARecv*
- *WSARecvFrom*
- *WSAIoctl*
- *WSARecvMsg*
- *AcceptEx*
- *ConnectEx*
- *TransmitFile*
- *TransmitPackets*
- *DisconnectEx*
- *WSANSPIoctl*

To use overlapped I/O, each function takes a *WSAOVERLAPPED* structure as a parameter. When these functions are called with a *WSAOVERLAPPED* structure, they complete immediately—regardless of the socket's mode (described at the beginning of this chapter). They rely on the *WSAOVERLAPPED* structure to manage the completion of an I/O request. There are essentially two methods for managing the completion of an overlapped I/O request: your application can wait for ***event object notification*** or it can process completed requests through ***completion routines***. The first six functions in the list have another parameter in common: a

WSAOVERLAPPED_COMPLETION_ROUTINE. This parameter is an optional pointer to a completion routine function that gets called when an overlapped request completes. We will explore the event notification method next. Later in this chapter, you will learn how to use optional completion routines instead of events to process completed overlapped requests.

Event Notification

The event notification method of overlapped I/O requires associating Windows event objects with *WSAOVERLAPPED* structures. When I/O calls such as *WSASend* and *WSARecv* are made using a *WSAOVERLAPPED* structure, they return immediately. Typically, you will find that these I/O calls fail with the return value *SOCKET_ERROR* and that *WSAGetLastError* reports a *WSA_IO_PENDING* error status. This error status simply means that the I/O operation is in progress. At a later time, your application will need to determine when an overlapped I/O request completes by waiting on the event object associated with the *WSAOVERLAPPED* structure. The *WSAOVERLAPPED* structure provides the communication medium between the initiation of an overlapped I/O request and its subsequent completion, and is defined as

```
typedef struct WSAOVERLAPPED
{
    DWORD    Internal;
    DWORD    InternalHigh;
    DWORD    Offset;
    DWORD    OffsetHigh;
    WSAEVENT hEvent;
} WSAOVERLAPPED, FAR * LPWSAOVERLAPPED;
```

The *Internal, InternalHigh, Offset*, and *OffsetHigh* fields are all used internally by the system and an application should not manipulate or directly use them. The *hEvent* field, on the other hand, allows an application to associate an event object handle with this operation.

When an overlapped I/O request finally completes, your application is responsible for retrieving the overlapped results. In the event notification method, Winsock will change the event-signaling state of an event object that is associated with a *WSAOVERLAPPED* structure from non-signaled to signaled when an overlapped request finally completes. Because an event object is assigned to the *WSAOVERLAPPED* structure, you can easily determine when an overlapped I/O call completes by calling the *WSAWaitForMultipleEvents* function, which we also described in the *WSAEventSelect* I/O model. *WSAWaitForMultipleEvents* waits a specified amount of time for one or more event objects to become signaled. We can't stress this point enough: remember that *WSAWaitForMultipleEvents* is capable of waiting on only 64 event objects at a time. Once you determine which overlapped request has completed, you need to determine the success or failure of the overlapped call by calling *WSAGetOverlappedResult*, which is defined as

```
BOOL WSAGetOverlappedResult(
    SOCKET s,
    LPWSAOVERLAPPED lpOverlapped,
    LPDWORD lpcbTransfer,
    BOOL fWait,
    LPDWORD lpdwFlags
);
```

The *s* parameter identifies the socket that was specified when the overlapped operation was started. The *lpOverlapped* parameter is a pointer to the *WSAOVERLAPPED* structure that was specified when the overlapped operation was started. The *lpcbTransfer* parameter is a pointer to a *DWORD* variable that receives the number of bytes that were actually transferred by an overlapped send or receive operation. The *fWait* parameter determines whether the function should wait for a pending overlapped operation to complete. If *fWait* is *TRUE*, the function does not return until the operation has been completed. If *fWait* is *FALSE* and the operation is still pending, *WSAGetOverlappedResult* returns *FALSE* with the error *WSA_IO_INCOMPLETE*. Because in our case we waited on a signaled event for overlapped completion, this parameter has no effect. The final parameter, *lpdwFlags*, is a pointer to a *DWORD* that will receive resulting flags if the originating overlapped call was made with the *WSARecv* or the *WSARecvFrom* function.

If the *WSAGetOverlappedResult* function succeeds, the return value is *TRUE*. This means that your overlapped operation has completed successfully and that the value pointed to by *lpcbTransfer* has been updated. If the return value is *FALSE*, one of the following statements is true:

- The overlapped I/O operation is still pending (as we previously described).
- The overlapped operation completed, but with errors.
- The overlapped operation's completion status could not be determined because of errors in one or more of the parameters supplied to *WSAGetOverlappedResult*.

Upon failure, the value pointed to by *lpcbTransfer* will not be updated, and your application should call the *WSAGetLastError* function to determine the cause of the failure.

The following sample of code demonstrates how to structure a simple server application that is capable of managing overlapped I/O on one socket using the event notification described above.

```
#define DATA_BUFSIZE                                         4096
void main(void)
{
    WSABUF DataBuf;
    char buffer[DATA_BUFSIZE];
    DWORD EventTotal = 0,
          RecvBytes=0,
          Flags=0;
    WSAEVENT EventArray[WSA_MAXIMUM_WAIT_EVENTS];
    WSAOVERLAPPED AcceptOverlapped;
    SOCKET ListenSocket, AcceptSocket;

    // Step 1:
    //  Start Winsock and set up a listening socket
    ...

    // Step 2:
    //  Accept an inbound connection
    AcceptSocket = accept(ListenSocket, NULL, NULL);

    // Step 3:
```

```
   //  Set up an overlapped structure

   EventArray[EventTotal] = WSACreateEvent();

   ZeroMemory(&AcceptOverlapped,
       sizeof(WSAOVERLAPPED));
    AcceptOverlapped.hEvent = EventArray[EventTotal];

   DataBuf.len = DATA_BUFSIZE;
   DataBuf.buf = buffer;

   EventTotal++;

   // Step 4:
   //  Post a WSARecv request to begin receiving data
   //  on the socket

   if (WSARecv(AcceptSocket, &DataBuf, 1, &RecvBytes,
       &Flags, &AcceptOverlapped, NULL) == SOCKET_ERROR)
   {
    if (WSAGetLastError() != WSA_IO_PENDING)
    {
        // Error occurred
    }
}

   // Process overlapped receives on the socket

   while(TRUE)
   {
    DWORD    Index;
       // Step 5:
       //  Wait for the overlapped I/O call to complete
       Index = WSAWaitForMultipleEvents(EventTotal,
           EventArray, FALSE, WSA_INFINITE, FALSE);

       // Index should be 0 because we
       // have only one event handle in EventArray

       // Step 6:
       //  Reset the signaled event
       WSAResetEvent(
           EventArray[Index - WSA_WAIT_EVENT_0]);

       // Step 7:
       //  Determine the status of the overlapped
       //  request
       WSAGetOverlappedResult(AcceptSocket,
           &AcceptOverlapped, &BytesTransferred,
           FALSE, &Flags);

       // First check to see whether the peer has closed
       // the connection, and if so, close the
       // socket

       if (BytesTransferred == 0)
       {
           printf("Closing socket %d\n", AcceptSocket);
```

```
            closesocket(AcceptSocket);
            WSACloseEvent(
                EventArray[Index - WSA_WAIT_EVENT_0]);
            return;
        }

        // Do something with the received data
        // DataBuf contains the received data
        ...

        // Step 8:
        //  Post another WSARecv() request on the socket

        Flags = 0;
        ZeroMemory(&AcceptOverlapped,
            sizeof(WSAOVERLAPPED));

        AcceptOverlapped.hEvent = EventArray[Index -
            WSA_WAIT_EVENT_0];

        DataBuf.len = DATA_BUFSIZE;
        DataBuf.buf = buffer;

        if (WSARecv(AcceptSocket, &DataBuf, 1,
            &RecvBytes, &Flags, &AcceptOverlapped,
            NULL) == SOCKET_ERROR)
        {
            if (WSAGetLastError() != WSA_IO_PENDING)
            {
                // Unexpected error
            }
        }
    }
}
```

The application outlines the following programming steps:

1. Create a socket and begin listening for a connection on a specified port.
2. Accept an inbound connection.
3. Create a *WSAOVERLAPPED* structure for the accepted socket and assign an event object handle to the structure. Also assign the event object handle to an event array to be used later by the *WSAWaitForMultipleEvents* function.
4. Post an asynchronous *WSARecv* request on the socket by specifying the *WSAOVERLAPPED* structure as a parameter.
5. Call *WSAWaitForMultipleEvents* using the event array and wait for the event associated with the overlapped call to become signaled.
6. Determine the return status of the overlapped call by using *WSA-GetOverlappedResult*.
7. Reset the event object by using *WSAResetEvent* with the event array and process the completed overlapped request.

8. Post another overlapped *WSARecv* request on the socket.
9. Repeat steps 5–8.

This example can easily be expanded to handle more than one socket by moving the overlapped I/O processing portion of the code to a separate thread and allowing the main application thread to service additional connection requests.

> If a Winsock function is called in an overlapped fashion (either by specifying an event within the *WSAOVERLAPPED* structure or with a completion routine), the operation might complete immediately. For example, calling *WSARecv* when data has already been received and buffered causes *WSARecv* to return *NO_ERROR*. If any overlapped functio fails with *WSA_IO_PENDING* or immediately succeeds, the completion event will ***always*** be signaled and the completion routine will be scheduled to run (if specified). F overlapped I/O with a completion port, this means that completion notification will be posted to the completion port for servicing.

Completion Routines

Completion routines are the other method your application can use to manage completed overlapped I/O requests. Completion routines are simply functions that you optionally pass to an overlapped I/O request and that the system invokes when an overlapped I/O request completes. Their primary role is to service a completed I/O request using the caller's thread. In addition, applications can continue overlapped I/O processing through the completion routine.

To use completion routines for overlapped I/O requests, your application must specify a completion routine, along with a *WSAOVERLAPPED* structure, to an I/O bound Winsock function (described previously). A completion routine must have the following function prototype:

```
void CALLBACK CompletionROUTINE(
    DWORD dwError,
    DWORD cbTransferred,
    LPWSAOVERLAPPED lpOverlapped,
    DWORD dwFlags
);
```

When an overlapped I/O request completes using a completion routine, the parameters contain the following information:

- The parameter *dwError* specifies the completion status for the overlapped operation as indicated by *lpOverlapped*.
- The *cbTransferred* parameter specifies the number of bytes that were transferred during the overlapped operation.
- The *lpOverlapped* parameter is the same as the *WSAOVERLAPPED* structure passed into the

originating I/O call.

- The *dwFlags* parameter returns any flags that the operation may have completed with (such as from WSARecv).

There is a major difference between overlapped requests submitted with a completion routine and overlapped requests submitted with an event object. The *WSAOVERLAPPED* structure's event field, *hEvent*, is not used, which means you cannot associate an event object with the overlapped request. Once you make an overlapped I/O call with a completion routine, your calling thread must eventually service the completion routine once it has completed. This requires you to place your calling thread in an ***alertable wait state*** and process the completion routine later, after the I/O operation has completed. The *WSAWaitForMultipleEvents* function can be used to put your thread in an alertable wait state. The catch is that you must also have at least one event object available for the *WSAWaitForMultipleEvents* function. If your application handles only overlapped requests with completion routines, you are not likely to have any event objects around for processing. As an alternative, your application can use the Windows *SleepEx* function to set your thread in an alertable wait state. Of course, you can also create a dummy event object that is not associated with anything. If your calling thread is always busy and not in an alertable wait state, no posted completion routine will ever get called.

As you saw earlier, *WSAWaitForMultipleEvents* normally waits for event objects associated with *WSAOVERLAPPED* structures. This function is also designed to place your thread in an alertable wait state and to process completion routines for completed overlapped I/O requests if you set the parameter *fAlertable* to TRUE. When overlapped I/O requests complete with a completion routine, the return value is *WSA_IO_COMPLETION* instead of an event object index in the event array. The *SleepEx* function provides the same behavior as *WSAWaitForMultipleEvents* except that it does not need any event objects. The *SleepEx* function is defined as

```
DWORD SleepEx(
    DWORD dwMilliseconds,
    BOOL bAlertable
);
```

The *dwMilliseconds* parameter defines how long in milliseconds *SleepEx* will wait. If *dwMilliseconds* is set to *INFINITE*, *SleepEx* waits indefinitely. The *bAlertable* parameter determines how a completion routine will execute. If *bAlertable* is set to *FALSE* and an I/O completion callback occurs, the I/O completion function is not executed and the function does not return until the wait period specified in *dwMilliseconds* has elapsed. If it is set to *TRUE*, the completion routine executes and the *SleepEx* function returns *WAIT_IO_COMPLETION*.

The following code outlines how to structure a simple server application that is capable of managing one socket request using completion routines as described earlier.

```
#define DATA_BUFSIZE    4096

SOCKET AcceptSocket,
       ListenSocket;
WSABUF DataBuf;
WSAEVENT EventArray[MAXIMUM_WAIT_OBJECTS];
```

```
DWORD Flags,
      RecvBytes,
      Index;
char buffer[DATA_BUFSIZE];

void main(void)
{
    WSAOVERLAPPED Overlapped;

    // Step 1:
    //  Start Winsock, and set up a listening socket
    ...

    // Step 2:
    //  Accept a new connection
    AcceptSocket = accept(ListenSocket, NULL, NULL);

    // Step 3:
    //  Now that we have an accepted socket, start
    //  processing I/O using overlapped I/O with a
    //  completion routine. To get the overlapped I/O
    //  processing started, first submit an
    //  overlapped WSARecv() request.

    Flags = 0;

    ZeroMemory(&Overlapped, sizeof(WSAOVERLAPPED));

    DataBuf.len = DATA_BUFSIZE;
    DataBuf.buf = buffer;
     // Step 4:
    //  Post an asynchronous WSARecv() request
    //  on the socket by specifying the WSAOVERLAPPED
    //  structure as a parameter, and supply
    //  the WorkerRoutine function below as the
    //  completion routine

    if (WSARecv(AcceptSocket, &DataBuf, 1, &RecvBytes,
        &Flags, &Overlapped, WorkerRoutine)
        == SOCKET_ERROR)
    {
        if (WSAGetLastError() != WSA_IO_PENDING)
        {
            printf("WSARecv() failed with error %d\n",
                WSAGetLastError());
            return;
        }
    }

    // Because the WSAWaitForMultipleEvents() API
    // requires waiting on one or more event objects,
    // we will have to create a dummy event object.
    // As an alternative, we can use SleepEx()
    // instead.

    EventArray [0] = WSACreateEvent();

    while(TRUE)
    {
        // Step 5:
        Index = WSAWaitForMultipleEvents(1, EventArray,
            FALSE, WSA_INFINITE, TRUE);

        // Step 6:
        if (Index == WAIT_IO_COMPLETION)
        {
            // An overlapped request completion routine
            // just completed. Continue servicing
            // more completion routines.
            continue;
        }
        else
        {
            // A bad error occurred:óstop processing!
            // If we were also processing an event
            // object, this could be an index to
            // the event array.
            return;
        }
    }
}

void CALLBACK WorkerRoutine(DWORD Error,
                            DWORD BytesTransferred,
                            LPWSAOVERLAPPED Overlapped,
                            DWORD InFlags)
{
    DWORD SendBytes, RecvBytes;
    DWORD Flags;

    if (Error != 0 || BytesTransferred == 0)
    {
        // Either a bad error occurred on the socket
        // or the socket was closed by a peer
        closesocket(AcceptSocket);
        return;
    }

    // At this point, an overlapped WSARecv() request
    // completed successfully. Now we can retrieve the
    // received data that is contained in the variable
    // DataBuf. After processing the received data, we
    // need to post another overlapped WSARecv() or
    // WSASend() request. For simplicity, we will post
    // another WSARecv() request.

    Flags = 0;

    ZeroMemory(&Overlapped, sizeof(WSAOVERLAPPED));

    DataBuf.len = DATA_BUFSIZE;
    DataBuf.buf = buffer;

    if (WSARecv(AcceptSocket, &DataBuf, 1, &RecvBytes,
        &Flags, &Overlapped, WorkerRoutine)
        == SOCKET_ERROR)
    {
        if (WSAGetLastError() != WSA_IO_PENDING )
        {
            printf("WSARecv() failed with error %d\n",
                WSAGetLastError());
            return;
        }
    }
}
```

The application illustrates the following programming steps:

1. Create a socket and begin listening for a connection on a specified port.
2. Accept an inbound connection.
3. Create a *WSAOVERLAPPED* structure for the accepted socket.
4. Post an asynchronous *WSARecv* request on the socket by specifying the *WSAOVERLAPPED* structure as a parameter and supplying a completion routine.
5. Call *WSAWaitForMultipleEvents* with the *fAlertable* parameter set to *TRUE* and wait for an overlapped request to complete. When an overlapped request completes, the completion routine automatically executes and *WSAWaitForMultipleEvents* returns *WSA_IO_COMPLETION*. Inside the completion routine, then post another overlapped *WSARecv* request with a completion routine.
6. Verify that *WSAWaitForMultipleEvents* returns *WSA_IO_COMPLETION*.
7. Repeat steps 5 and 6.

The overlapped model provides high-performance socket I/O. It is different from all the previous models because an application posts buffers to send and receive data that the system uses directly. That is, if an application posts an overlapped receive with a 10 KB buffer and data arrives on the socket, it is copied directly into this posted buffer. In the previous models, data would arrive and be copied to the per-socket receive buffers at which point the application is notified of the capability to read. After the application calls a receive function, the data is copied from the per-socket buffer to the application's buffer. Chapter 6 will discuss strategies for developing high-performance, scalable Winsock applications. Chapter 6 will also discuss the *WSARecvMsg*, *AcceptEx*, *ConnectEx*, *TransmitFile*, *TransmitPackets*, and *DisconnectEx* API functions in more detail.

The disadvantage of using overlapped I/O with events is, again, the limitation of being able to wait on a maximum of 64 events at a time. Completion routines are a good alternative but care must be taken to ensure that the thread that posted the operation goes into an alertable wait state in order for the completion routine to complete. Also, care should be taken to make sure that the completion routines do not perform excessive computations so that these routines may fire as fast as possible under a heavy load.

The Completion Port Model

For newcomers, the completion port model seems overwhelmingly complicated because extra work is required to add sockets to a completion port when compared to the initialization steps for the other I/O models. However, as you will see, these steps are not that complicated once you understand them. Also, the completion port model offers the best system performance possible when an application has to manage many sockets at once. Unfortunately, it's available only on Windows NT, Windows 2000, and Windows XP; however, the completion port model offers the best scalability of all the models discussed so far. This model is well suited to handling hundreds or thousands of sockets.

Essentially, the completion port model requires you to create a Windows completion port object that will manage overlapped I/O requests using a specified number of threads to service the completed overlapped I/O requests. Note that a completion port is actually a Windows I/O construct that is capable of accepting more than just socket handles. However, this section will describe only how to take advantage of the completion port model by using socket handles. To begin using this model, you are required to create an I/O completion port object that will be used to manage multiple I/O requests for any number of socket handles. This is accomplished by calling the *CreateIoCompletionPort* function, which is defined as

```
HANDLE CreateIoCompletionPort(
    HANDLE FileHandle,
    HANDLE ExistingCompletionPort,
    DWORD CompletionKey,
    DWORD NumberOfConcurrentThreads
);
```

Before examining the parameters in detail, be aware that this function is actually used for two distinct purposes:

- To create a completion port object
- To associate a handle with a completion port

When you initially create a completion port object, the only parameter of interest is *NumberOfConcurrentThreads*; the first three parameters are not significant. The *NumberOfConcurrentThreads* parameter is special because it defines the number of threads that are allowed to execute concurrently on a completion port. Ideally, you want only one thread per processor to service the completion port to avoid thread context switching. The value 0 for this parameter tells the system to allow as many threads as there are processors in the system. The following code creates an I/O completion port.

```
 CompletionPort = CreateIoCompletionPort(INVALID_HANDLE_VALUE,
    NULL, 0, 0);
```

This will return a handle that is used to identify the completion port when a socket handle is assigned to it.

Worker Threads and Completion Ports

After a completion port is successfully created, you can begin to associate socket handles with the object. Before associating sockets, though, you have to create one or more worker threads to service the completion port when socket I/O requests are posted to the completion port object. At this point, you might wonder how many threads should be created to service the completion port. This is actually one of the more complicated aspects of the completion port model because the number needed to service I/O requests depends on the overall design of your application. It's important to note the distinction between number of concurrent threads to specify when calling *CreateIoCompletionPort* versus the number of worker threads to create; they do not represent the same thing. We recommended previously that you should have the *CreateIoCompletionPort* function specify one thread per processor to avoid thread context switching. The *NumberOfConcurrentThreads* parameter of *CreateIoCompletionPort* explicitly tells the system to allow only *n* threads to operate at a time on the completion port. If you create more than *n* worker threads on the completion port, only *n* threads will be allowed to operate at a time. (Actually, the system might exceed this value for a short amount of time, but the system will quickly bring it down to the value you specify in *CreateIoCompletionPort*.) You might be wondering why you would create more worker threads than the number specified by the *CreateIoCompletionPort* call. As we mentioned previously, this depends on the overall design of your application. If one of your worker threads calls a function—such as *Sleep* or *WaitForSingleObject*—and becomes suspended, another thread will be allowed to operate in its place. In other words, you always want to have as many threads available for execution as the number of threads you allow to execute in the *CreateIoCompletionPort* call. Thus, if you expect your worker thread to ever become blocked, it is reasonable to create more worker threads than the value specified in *CreateIoCompletionPort's NumberOfConcurrentThreads* parameter.

Once you have enough worker threads to service I/O requests on the completion port, you can begin to associate socket handles with the completion port. This requires calling the *CreateIoCompletionPort* function on an existing completion port and supplying the first three parameters—*FileHandle*, *ExistingCompletionPort*, and *CompletionKey*—with socket information. The *FileHandle* parameter represents a socket handle to associate with the completion port. The *ExistingCompletionPort* parameter identifies the completion port to which the socket handle is to be associated with. The *CompletionKey* parameter identifies *per-handle data* that you can associate with a particular socket handle. Applications are free to store any type of information associated with a socket by using this key. We call it per-handle data because it represents data associated with a socket handle. It is useful to store the socket handle using the key as a pointer to a data structure containing the socket handle and other socket-specific information. As we will see later in this chapter, the thread routines that service the completion port can retrieve socket-handle–specific information using this key.

Let's begin to construct a basic application framework from what we've described so far. The following example demonstrates how to start developing an echo server application using the completion port model. In this code, we take the following preparation steps:

1. Create a completion port. The fourth parameter is left as 0, specifying that only one worker thread per processor will be allowed to execute at a time on the completion port.
2. Determine how many processors exist on the system.
3. Create worker threads to service completed I/O requests on the completion port using processor information in step 2. In the case of this simple example, we create one worker thread per

processor because we do not expect our threads to ever get in a suspended condition in which there would not be enough threads to execute for each processor. When the *CreateThread* function is called, you must supply a worker routine that the thread executes upon creation. We will discuss the worker thread's responsibilities later in this section.

4. Prepare a listening socket to listen for connections on port 5150.
5. Accept inbound connections using the accept function.
6. Create a data structure to represent per-handle data and save the accepted socket handle in the structure.
7. Associate the new socket handle returned from *accept* with the completion port by calling *CreateIoCompletionPort*. Pass the per-handle data structure to *CreateIoCompletionPort* via the completion key parameter.
8. Start processing I/O on the accepted connection. Essentially, you want to post one or more asynchronous *WSARecv* or *WSASend* requests on the new socket using the overlapped I/O mechanism. When these I/O requests complete, a worker thread services the I/O requests and continues processing future I/O requests, as we will see later in the worker routine specified in step 3.
9. Repeat steps 5–8 until server terminates.

```
HANDLE CompletionPort;
WSADATA wsd;
SYSTEM_INFO SystemInfo;
SOCKADDR_IN InternetAddr;
SOCKET Listen;
int i;

typedef struct _PER_HANDLE_DATA
{
        SOCKET                  Socket;
                SOCKADDR_STORAGE  ClientAddr;
        // Other information useful to be associated with the handle
} PER_HANDLE_DATA, * LPPER_HANDLE_DATA;

// Load Winsock
StartWinsock(MAKEWORD(2,2), &wsd);

// Step 1:
// Create an I/O completion port

CompletionPort = CreateIoCompletionPort(
    INVALID_HANDLE_VALUE, NULL, 0, 0);

// Step 2:
// Determine how many processors are on the system

GetSystemInfo(&SystemInfo);

// Step 3:
// Create worker threads based on the number of
// processors available on the system. For this
// simple case, we create one worker thread for each
// processor.

for(i = 0; i < SystemInfo.dwNumberOfProcessors; i++)
{
    HANDLE ThreadHandle;

    // Create a server worker thread, and pass the
    // completion port to the thread. NOTE: the
    // ServerWorkerThread procedure is not defined
    // in this listing.

    ThreadHandle = CreateThread(NULL, 0,
        ServerWorkerThread, CompletionPort,
        0, NULL;

    // Close the thread handle
    CloseHandle(ThreadHandle);
}

// Step 4:
// Create a listening socket

Listen = WSASocket(AF_INET, SOCK_STREAM, 0, NULL, 0,
    WSA_FLAG_OVERLAPPED);

InternetAddr.sin_family = AF_INET;
InternetAddr.sin_addr.s_addr = htonl(INADDR_ANY);
InternetAddr.sin_port = htons(5150);
bind(Listen, (PSOCKADDR) &InternetAddr,
    sizeof(InternetAddr));

// Prepare socket for listening

listen(Listen, 5);

while(TRUE)
{
    PER_HANDLE_DATA *PerHandleData=NULL;
    SOCKADDR_IN saRemote;
    SOCKET Accept;
    int RemoteLen;
    // Step 5:
    // Accept connections and assign to the completion
    // port

    RemoteLen = sizeof(saRemote);
    Accept = WSAAccept(Listen, (SOCKADDR *)&saRemote,
    &RemoteLen);

    // Step 6:
    // Create per-handle data information structure to
    // associate with the socket
    PerHandleData = (LPPER_HANDLE_DATA)
        GlobalAlloc(GPTR, sizeof(PER_HANDLE_DATA));

    printf("Socket number %d connected\n", Accept);
    PerHandleData->Socket = Accept;
    memcpy(&PerHandleData->ClientAddr, &saRemote, RemoteLen);
```

```
        // Step 7:
        // Associate the accepted socket with the
        // completion port

        CreateIoCompletionPort((HANDLE) Accept,
            CompletionPort, (DWORD) PerHandleData, 0);

        // Step 8:
        //  Start processing I/O on the accepted socket.
        //  Post one or more WSASend() or WSARecv() calls
        //  on the socket using overlapped I/O.
        WSARecv(...);
    }

        DWORD WINAPI ServerWorkerThread(LPVOID lpParam)
        {
        // The requirements for the worker thread will be
        // discussed later.
        return 0;
    }
```

Completion Ports and Overlapped I/O

After associating a socket handle with a completion port, you can begin processing I/O requests by posting overlapped send and receive requests on the socket handle. You can now start to rely on the completion port for I/O completion notification. Basically, the completion port model takes advantage of the Windows overlapped I/O mechanism in which Winsock API calls such as *WSASend* and *WSARecv* return immediately when called. It is up to your application to retrieve the results of the calls at a later time through an *OVERLAPPED* structure. In the completion port model, this is accomplished by having one or more worker threads wait on the completion port using the *GetQueuedCompletionStatus* function, which is defined as

```
BOOL GetQueuedCompletionStatus(
    HANDLE CompletionPort,
    LPDWORD lpNumberOfBytesTransferred,
    PULONG_PTR lpCompletionKey,
    LPOVERLAPPED * lpOverlapped,
    DWORD dwMilliseconds
);
```

The *CompletionPort* parameter represents the completion port to wait on. The *lpNumberOfBytesTransferred* parameter receives the number of bytes transferred after a completed I/O operation, such as *WSASend* or *WSARecv*. The *lpCompletionKey* parameter returns per-handle data for the socket that was originally passed into the *CreateIoCompletionPort* function. As we already mentioned, we recommend saving the socket handle in this key. The *lpOverlapped* parameter receives the *WSAOVERLAPPED* structure of the completed I/O operation. This is actually an important parameter because it can be used to retrieve ***per I/O–operation data***, which we will describe shortly. The final parameter, *dwMilliseconds*, specifies the number of milliseconds that the caller is willing to wait for a completion packet to appear on the completion port. If you specify *INFINITE*, the call waits forever.

Per-handle Data and Per-I/O Operation Data

When a worker thread receives I/O completion notification from the *GetQueuedCompletionStatus* API call, the *lpCompletionKey* and *lpOverlapped* parameters contain socket information that can be used to continue processing I/O on a socket through the completion port. Two types of important socket data are available through these parameters: per-handle data and per-I/O operation data.

The *lpCompletionKey* parameter contains what we call per-handle data because the data is related to a socket handle when a socket is first associated with the completion port. This is the data that is passed as the *CompletionKey* parameter of the *CreateIoCompletionPort* API call. As we noted earlier, your application can pass any type of socket information through this parameter. Typically, applications will store the socket handle related to the I/O request here.

The *lpOverlapped* parameter contains an *OVERLAPPED* structure followed by what we call per-I/O operation data, which is anything that your worker thread will need to know when processing a completion packet (echo the data back, accept the connection, post another read, and so on). Per-I/O operation data is any number of bytes contained in a structure also containing an *OVERLAPPED* structure that you pass into a function that expects an *OVERLAPPED* structure. A simple way to make this work is to define a structure and place an *OVERLAPPED* structure as a field of the new structure. For example, we declare the following data structure to manage per-I/O operation data:

```
typedef struct
{
    OVERLAPPED Overlapped;
    char       Buffer[DATA_BUFSIZE];
    int                    BufferLen;
    int        OperationType;
} PER_IO_DATA;
```

This structure demonstrates some important data elements you might want to relate to an I/O operation, such as the type of I/O operation (a send or receive request) that just completed. In this structure, we consider the data buffer for the completed I/O operation to be useful. To call a Winsock API function that expects an *OVERLAPPED* structure, you dereference the *OVERLAPPED* element of your structure. For example,

```
PER_IO_OPERATION_DATA PerIoData;
WSABUF wbuf;
DWORD Bytes, Flags;
// Initialize wbuf ...
WSARecv(socket, &wbuf, 1, &Bytes, &Flags, &(PerIoData.Overlapped),
NULL);
```

Later in the worker thread, *GetQueuedCompletionStatus* returns with an overlapped structure and completion key. To retrieve the per-I/O data the macro *CONTAINING_RECORD* should be used. For example,

```
PER_IO_DATA  *PerIoData=NULL;
OVERLAPPED   *lpOverlapped=NULL;
```

```
ret = GetQueuedCompletionStatus(
         CompPortHandle,
         &Transferred,
         (PULONG_PTR)&CompletionKey,
         &lpOverlapped,
                        INFINITE);
// Check for successful return
PerIoData = CONTAINING_RECORD(lpOverlapped, PER_IO_DATA, Overlapped);
```

This macro should be used; otherwise, the *OVERLAPPED* member of the *PER_IO_DATA* structure would always have to appear first, which can be a dangerous assumption to make (especially with multiple developers working on the same code).

You can determine which operation was posted on this handle by using a field of the per-I/O structure to indicate the type of operation posted. In our example, the *OperationType* member would be set to indicate a read, write, etc., operation. One of the biggest benefits of per-I/O operation data is that it allows you to manage multiple I/O operations (such as read/write, multiple reads, and multiple writes) on the same handle. You might ask why you would want to post more than one I/O operation at a time on a socket. The answer is scalability. For example, if you have a multiple-processor machine with a worker thread using each processor, you could potentially have several processors sending and receiving data on a socket at the same time.

Before continuing, there is one other important aspect about Windows completion ports that needs to be stressed. All overlapped operations are guaranteed to be executed in the order that the application issued them. However, the completion notifications returned from a completion port are ***not*** guaranteed to be in that same order. That is, if an application posts two overlapped *WSARecv* operations, one with a 10 KB buffer and the next with a 12 KB buffer, the 10 KB buffer is filled first, followed by the 12 KB buffer. The application's worker thread may receive notification from *GetQueuedCompletionStatus* for the 12 KB *WSARecv* before the completion event for the 10 KB operation. Of course, this is only an issue when multiple operations are posted on a socket.

To complete this simple echo server sample, we need to supply a *ServerWorkerThread* function. The following code outlines how to develop a worker thread routine that uses per-handle data and per-I/O operation data to service I/O requests.

```
 DWORD WINAPI ServerWorkerThread(
    LPVOID CompletionPortID)
{
    HANDLE CompletionPort = (HANDLE) CompletionPortID;
    DWORD BytesTransferred;
    LPOVERLAPPED Overlapped;
    LPPER_HANDLE_DATA PerHandleData;
    LPPER_IO_DATA PerIoData;
    DWORD SendBytes, RecvBytes;
    DWORD Flags;

    while(TRUE)
    {
        // Wait for I/O to complete on any socket
        // associated with the completion port
        ret = GetQueuedCompletionStatus(CompletionPort,
            &BytesTransferred,(LPDWORD)&PerHandleData,
            (LPOVERLAPPED *) &PerIoData, INFINITE);

        // First check to see if an error has occurred
        // on the socket; if so, close the
        // socket and clean up the per-handle data
        // and per-I/O operation data associated with
        // the socket

        if (BytesTransferred == 0 &&
            (PerIoData->OperationType == RECV_POSTED ||
             PerIoData->OperationType == SEND_POSTED))
        {
            // A zero BytesTransferred indicates that the
            // socket has been closed by the peer, so
            // you should close the socket. Note:
            // Per-handle data was used to reference the
            // socket associated with the I/O operation.

            closesocket(PerHandleData->Socket);

            GlobalFree(PerHandleData);
            GlobalFree(PerIoData);
            continue;
        }

        // Service the completed I/O request. You can
        // determine which I/O request has just
        // completed by looking at the OperationType
        // field contained in the per-I/O operation data.
         if (PerIoData->OperationType == RECV_POSTED)
        {
            // Do something with the received data
            // in PerIoData->Buffer
        }

        // Post another WSASend or WSARecv operation.
        // As an example, we will post another WSARecv()
        // I/O operation.

        Flags = 0;

        // Set up the per-I/O operation data for the next
        // overlapped call
        ZeroMemory(&(PerIoData->Overlapped),
            sizeof(OVERLAPPED));

        PerIoData->DataBuf.len = DATA_BUFSIZE;
        PerIoData->DataBuf.buf = PerIoData->Buffer;
        PerIoData->OperationType = RECV_POSTED;

        WSARecv(PerHandleData->Socket,
            &(PerIoData->DataBuf), 1, &RecvBytes,
            &Flags, &(PerIoData->Overlapped), NULL);
    }
}
```

If an error has occurred for a given overlapped operation, *GetQueuedCompletionStatus* will return FALSE. Because completion ports are a Windows I/O construct, if you call *GetLastError* or *WSAGetLastError*, the error code is likely to be a Windows error code and not a Winsock error code. To retrieve the equivalent Winsock error code, *WSAGetOverlappedResult* can be called specifying the socket handle and *WSAOVERLAPPED* structure for the completed operation, after which *WSAGetLastError* will return the translated Winsock error code.

One final detail not outlined in the last two examples we have presented is how to properly close an I/O completion port—especially if you have one or more threads in progress performing I/O on several sockets. The main thing to avoid is freeing an *OVERLAPPED* structure when an overlapped I/O operation is in progress. The best way to prevent this is to call *closesocket* on every socket handle—any overlapped I/O operations pending will complete. Once all socket handles are closed, you need to terminate all worker threads on the completion port. This can be accomplished by sending a special completion packet to each worker thread using the *PostQueuedCompletionStatus* function, which informs each thread to exit immediately. *PostQueuedCompletionStatus* is defined as

```
BOOL PostQueuedCompletionStatus(
    HANDLE CompletionPort,
    DWORD dwNumberOfBytesTransferred,
    ULONG_PTR dwCompletionKey,
    LPOVERLAPPED lpOverlapped
);
```

The *CompletionPort* parameter represents the completion port object to which you want to send a completion packet. The *dwNumberOfBytesTransferred*, *dwCompletionKey*, and *lpOverlapped* parameters each will allow you to specify a value that will be sent directly to the corresponding parameter of the *GetQueuedCompletionStatus* function. Thus, when a worker thread receives the three passed parameters of *GetQueuedCompletionStatus*, it can determine when it should exit based on a special value set in one of the three parameters. For example, you could pass the value 0 in the *dwCompletionKey* parameter, which a worker thread could interpret as an instruction to terminate. Once all the worker threads are closed, you can close the completion port using the *CloseHandle* function and finally exit your program safely.

The completion port I/O model is by far the best in terms of performance and scalability. There are no limitations to the number of sockets that may be associated with a completion port and only a small number of threads are required to service the completed I/O. For more information on using completion ports to develop scalable, high-performance servers, see Chapter 6.

I/O Model Consideration

By now you might be wondering how to choose the I/O model you should use when designing your application. As we've mentioned, each model has its strengths and weaknesses. All of the I/O models do require fairly complex programming compared with developing a simple blocking-mode application with many servicing threads. We offer the following suggestions for client and server development.

Client Development

When you are developing a client application that manages one or more sockets, we recommend using overlapped I/O or *WSAEventSelect* over the other I/O models for performance reasons. But, if you are developing a Windows-based application that manages window messages, the *WSAAsyncSelect* model might be a better choice because *WSAAsyncSelect* lends itself to the Windows message model, and your application is already set up for handling messages.

Server Development

When you are developing a server that processes several sockets at a given time, we recommend using overlapped I/O over the other I/O models for performance reasons. However, if you expect your server to service a large number of I/O requests at any given time, you should consider using the I/O completion port model for even better performance.

Conclusion

At this point, we have covered the various I/O models available in Winsock. These models allow applications to tailor Winsock I/O according to their needs, from simple blocking I/O to high-performance completion port I/O for the maximum throughput possible. Up to this point, you have learned about available transport protocols, socket creation attributes, creating a basic client/server application, and other fundamental Winsock topics. Chapter 6 focuses on writing high-performance, scalable servers, and the remaining chapters introduce more specialized Winsock topics.

Chapter 6

Scalable Winsock Applications

Developing Winsock applications has always been considered to be cryptic and difficult to learn. In reality, there are only a few basic principles, such as socket creation, connecting a socket, accepting connections, and sending and receiving data. The real difficultly lies in developing a scalable Winsock application that can handle a single connection or thousands of connections. This chapter describes how to write scalable Winsock applications for Windows NT. The main focus is the server side of the client-server model; however, some of the topics apply equally to both.

This discussion of writing scalable applications applies to server applications and therefore only applies to Windows NT 4.0 and later versions. We don't include earlier versions of Windows NT because many of the features we will cover require Winsock 2, which is available only on Windows NT 4.0 and later versions. Finally, the focus of our discussion will be on the TCP/IP protocol. However, all of the topics we cover can easily apply to other connection-oriented, stream-based protocols. Some of the topics apply to UDP/IP as well (such as resource management) but connectionless, message-based protocols themselves will not be covered.

This chapter will first discuss the different Winsock API functions designed for use in scalable, high-performance applications such as *AcceptEx*, *TransmitFile*, and *ConnectEx*. Typically, these are Microsoft-specific extensions added with different versions of the operating system because the original Winsock specification leaves out several key asynchronous functions. We'll then cover the necessary steps for implementing a scalable server and discuss how to handle low resource conditions that occur when the number of connections becomes very large.

APIs and Scalability

The only I/O model that provides true scalability on Windows NT platforms is overlapped I/O using completion ports for notification. In Chapter 5, we covered the various methods of socket I/O and explained that for a large number of connections, completion ports offer the greatest flexibility and ease of implementation. Mechanisms like *WSAAsyncSelect* and *select* are provided for easier porting from Windows 3.1 and UNIX, respectively, but are not designed to scale. The event-based models are not scalable because of the operating system limit of simultaneous wait events.

The other major advantages of overlapped I/O are the several Microsoft-specific extensions that can only be called in an overlapped manner. When you use overlapped I/O there are several options for how the notifications can be received. Event-based notification is not scalable because the operating system limit of waiting on 64 objects necessitates using many threads. This is not only inefficient but requires a lot of housekeeping overhead to assign events to available worker threads. Overlapped I/O with callbacks is not an option for several reasons. First, many of the Microsoft-specific extensions do not allow Asynchronous Procedure Calls (APCs) for completion notification. Second, due to the nature of how APCs are handled on Windows, it is possible for an application thread to starve. Once a thread goes into an alertable wait, all pending APCs are handled on a first in first out (FIFO) basis. Now consider the situation in which a server has a connection established and posts an overlapped *WSARecv* with a completion function. When there is data to receive, the completion routine fires and posts another overlapped *WSARecv*. Depending on timing conditions and how much work is performed within the APC, another completion function is queued (because there is more data to be read). This can cause the server's thread to starve as long as there is pending data on that socket.

Before delving deeper into the architecture of scalable Winsock applications, let's discuss the Microsoft-specific extensions that will aid us in developing scalable servers. These APIs are *TransmitFile*, *AcceptEx*, *ConnectEx*, *TransmitPackets*, *DisconnectEx*, and *WSARecvMsg*. There is a related extension function, *GetAcceptExSockaddrs*, which is used in conjunction with *AcceptEx*.

Before describing each of the extension API functions, it is important to point out that these functions are defined in MSWSOCK.H. Also, only three of the functions (*TransmitFile*, *AcceptEx*, and *GetAcceptExSockaddrs*) are actually exported from MSWSOCK.DLL. However, applications should avoid using those. Instead, applications should dynamically load the extension function, which is required for all the remaining extension APIs. Not all providers have to support these APIs, so it is best to explicitly load these APIs from the provider you are using. See Chapter 7 and the *SIO_GET_EXTENSION_ FUNCTION_POINTER* for an example of how to load the extension APIs.

AcceptEx

Perhaps the most useful extension API for scalable TCP/IP servers is *AcceptEx*. This function allows the server to post an asynchronous call that will accept the next incoming client connection. This function is defined as

```
BOOL
PASCAL FAR
AcceptEx (
    IN SOCKET sListenSocket,
    IN SOCKET sAcceptSocket,
    IN PVOID lpOutputBuffer,
    IN DWORD dwReceiveDataLength,
    IN DWORD dwLocalAddressLength,
    IN DWORD dwRemoteAddressLength,
    OUT LPDWORD lpdwBytesReceived,
    IN LPOVERLAPPED lpOverlapped
    );
```

The first parameter is the listening socket, and *sAcceptSocket* is a valid, unbound socket handle that will be assigned to the next client connection. So the socket handle for the client needs to be created before posting the *AcceptEx* call. This is necessary because socket creation is expensive, and if a server is interested in handling client connections as fast as possible, it needs to have a pool of sockets already created on which new connections will be assigned.

The four parameters that follow *sAcceptSocket* are related. The *lpOutputBuffer* is required and is filled in with the local and remote addresses for the client connection as well as an optional buffer to receive the first data chunk received from the client. The *dwReceiveDataLength* indicates how many bytes of the supplied buffer should be used to receive data sent by the client. An application may choose not to receive data and may specify zero. The *dwLocalAddressLength* specifies the size of the socket address structure corresponding to the address family of the client socket plus 16 bytes. The local address of the client socket connection is placed in the *lpOutputBuffer* following the receive data if specified. The *dwRemoteAddressLength* is the same. The remote address of the client connection will be written to the *lpOutputBuffer* following the receive data (if specified) and the local address. Note that *dwReceiveDataLength* may be zero but *dwLocalAddressLength* and *dwRemoteAddressLength* cannot be.

The *lpdwBytesReceived* indicates the number of bytes received on the newly-established client connection if the operation succeeds immediately. Finally, *lpOverlapped* is the *WSAOVERLAPPED* structure for this overlapped operation. This parameter is required—if you want to perform a blocking accept call, just use *accept* or *WSAAccept*.

Before going any farther, let's take a quick look at an example using the *AcceptEx* function. The following code creates an IPv4 listening socket and posts a single *AcceptEx*.

```
SOCKET                             s, sclient;
HANDLE                             hCompPort;
LPFN_ACCEPTEX                              lpfnAcceptEx=NULL;
GUID                               GuidAcceptEx=WSAID_ACCEPTEX;
// The WSAOVERLAPPEDPLUS type will be described in detail in
// Chapter 12 and includes a WSAOVERLAPPED structure as well as
// context information for the overlapped operation.
WSAOVERLAPPEDPLUS ol;
SOCKADDR_IN       salocal;
DWORD                           dwBytes;
char                            buf[1024];
int                             buflen=1024;
```

```
// Create the completion port
hCompPort = CreateIoCompletionPort(INVALID_HANDLE_VALUE,
                                   NULL,
                                   (ULONG_PTR)0,
                                   0
                                   );

// Create the listening socket
s = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);

// Associate listening socket to completion port
CreateIoCompletionPort((HANDLE)s,
                        hCompPort,
                        (ULONG_PTR)0,
                        0
                        );

// Bind the socket to the local port
salocal.sin_family = AF_INET;
salocal.sin_port   = htons(5150);
salocal.sin_addr.s_addr = htonl(INADDR_ANY);
bind(s, (SOCKADDR *)&salocal, sizeof(salocal));

// Set the socket to listening
listen(s, 200);

// Load the AcceptEx function
WSAIoctl(s,
                        SIO_GET_EXTENSION_FUNCTION_POINTER,
                        &GuidAcceptEx,
                        sizeof(GuidAcceptEx),
           &lpfnAcceptEx,
            sizeof(lpfnAcceptEx),
           &dwBytes,
            NULL,
            NULL
            );
// Create the client socket for the accepted connection
sclient = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);

// Initialize our "extended" overlapped structure
memset(&ol, 0, sizeof(ol));
ol.operation = OP_ACCEPTEX;
ol.client    = sclient;

lpfnAcceptEx(s,
             sclient,
             buf,
             buflen - ((sizeof(SOCKADDR_IN) + 16) * 2),
             sizeof(SOCKADDR_IN) + 16,
             sizeof(SOCKADDR_IN) + 16,
            &dwBytes,
            &ol.overlapped
             );

// Call GetQueuedCompletionStatus within the completion function
// After the AcceptEx operation completes associate the accepted client
// socket with the completion port
```

This sample is a bit simplified but it shows the necessary steps. It shows how to set up the listening socket, which you've seen before. Then it shows how to load the *AcceptEx* function. Applications should always load the extension functions themselves to avoid the performance penalty of the exported extension functions from MSWSOCK.DLL, because for each call they simply end up loading the same function. Next, the application-specific overlapped structure is established, which contains necessary information concerning the asynchronous operation so that when it completes the server can figure out what happened. The actual declaration of this type is not included for the sake of simplicity. See Chapter 5 for more information about this. Finally, once the *AcceptEx* operation completes, the newly-accepted client socket should be associated with the completion port.

Also be aware that because of the high performance nature of *AcceptEx*, the listening socket's socket attributes are not automatically inherited by the client socket. To do this, the server must call *setsockopt* with *SO_UPDATE_ ACCEPT_CONTEXT* with the client socket handle. See Chapter 7 for more information.

Another point to be aware of, which we mentioned in Chapter 5, is that if a receive buffer is specified to *AcceptEx* (for example, *dwReceiveDataLength* is greater than zero), then the overlapped operation will not complete until at least one byte of data has been received on the connection. So a malicious client could post many connections but never send any data. Chapter 5 discusses methods to prevent this by using the *SO_CONNECT_TIME* socket option. The *AcceptEx* function is available on Windows NT 4.0 and later versions.

GetAcceptExSockaddrs

This is really a companion function to *AcceptEx* because it is required to decode the local and remote addresses contained within the buffer passed to the accept call. As you remember, a single buffer will contain any data received on the connection as well as the local and remote addresses for that connection. Any data indicated to be received will always be placed at the start of this buffer followed by the addresses. However, these addresses are in a packed form and the *GetAcceptExSockaddrs* function will decode them into the appropriate *SOCKADDR* structure for the address family. This function is defined as

```
VOID
PASCAL FAR
GetAcceptExSockaddrs (
    IN PVOID lpOutputBuffer,
    IN DWORD dwReceiveDataLength,
    IN DWORD dwLocalAddressLength,
    IN DWORD dwRemoteAddressLength,
    OUT struct sockaddr **LocalSockaddr,
    OUT LPINT LocalSockaddrLength,
    OUT struct sockaddr **RemoteSockaddr,
    OUT LPINT RemoteSockaddrLength
    );
```

The first four parameters are the same as those in the *AcceptEx* call and they must match the values passed to *AcceptEx*. That is, if 1024 was specified as *dwReceiveDataLength* in *AcceptEx* then the same value must be passed to *GetAcceptExSockaddrs*. The remaining four parameters are *SOCKADDR* pointers and their lengths for the local and remote addresses. These parameters are all output parameters. The following code illustrates how you would call *Get- AcceptExSockaddrs* after the *AcceptEx* call in our previous example completes:

```
// buf and bufflen were defined previously
SOCKADDR *lpLocalSockaddr=NULL,
         *lpRemoteSockaddr=NULL;
int LocalSockaddrLen=0,
    RemoteSockaddrLen=0;
LPFN_GETACCEPTEXSOCKADDRS lpfnGetAcceptExSockaddrs=NULL;

// Load the GetAcceptExSockaddrs function

lpfnGetAcceptExSockaddrs(
         buf,
         buflen - ((sizeof(SOCKADDR_IN) + 16) * 2),
         sizeof(SOCKADDR_IN) + 16,
         sizeof(SOCKADDR_IN) + 16,
        &lpLocalSockaddr,
        &LocalSockaddrLen,
        &lpRemoteSockaddr,
        &RemoteSockaddrLen
         );
```

After the function completes, the *lpLocalSockaddr* and *lpRemoteSockaddr* point to within the specified buffer where the socket addresses have been unpacked into the correct socket address structure.

TransmitFile

TransmitFile is an extension API that allows an open file to be sent on a socket connection. This frees the application from having to manually open the file and repeatedly perform a read from the file, followed by writing that chunk of data on the socket. Instead, an open file handle is given along with the socket connection and the file data is read and sent on the socket all within kernel mode. This prevents the multiple kernel transitions required when you perform the file read yourself. This API is defined as

```
BOOL
PASCAL FAR
TransmitFile (
    IN SOCKET hSocket,
    IN HANDLE hFile,
    IN DWORD nNumberOfBytesToWrite,
    IN DWORD nNumberOfBytesPerSend,
    IN LPOVERLAPPED lpOverlapped,
    IN LPTRANSMIT_FILE_BUFFERS lpTransmitBuffers,
    IN DWORD dwReserved
    );
```

The first parameter is the connection socket. The *hFile* parameter is a handle to an open file. This parameter can be *NULL* in which case the *lpTransmitBuffers* are transmitted. Of course it doesn't make much sense to use *TransmitFile* to send memory-only buffers. *nNumberOfBytesToWrite* is the number of bytes to send from the file. A value of zero indicates send the entire file. The *nNumberOfBytesPerSend* indicates the size of each block of data sent in each send operation. If zero is specified, the system uses the default send size. The default send size on Windows NT Workstation is 4k and on Windows Server it is 64k. The *lpOverlapped* structure is optional. Note that if the *OVERLAPPED* structure is omitted, then the file transfer begins at the current file pointer position. Otherwise, the offset values in the *OVERLAPPED* structure can indicate where the operation starts. The *lpTransmitBuffers* is a *TRANSMIT_FILE_BUFFERS* structure that contains memory buffers to transmit before and after the file is transmitted. This parameter is optional. The last parameter is optional flags, which affect the behavior of the file operation. Table 6-1 contains the possible flags and their meaning. Multiple flags may be specified.

Table 6-1 *TransmitFile Flags*

Flag	Meaning
TF_DISCONNECT	Start a transport-level disconnect after the *TransmitFile* operation has been queued.
TF_REUSE_SOCKET	Prepare the socket handle to be reused. After the *TransmitFile* completes, the socket handle may be used as the client socket in *AcceptEx*. This flag is valid only if *TF_DISCONNECT* is also specified.
TF_USE_DEFAULT_WORKER	Indicates the file transfer to use the system's default thread. This is useful for large file sends.
TF_USE_SYSTEM_THREAD	This option also indicates the *TransmitFile* operation to use system threads for processing.
TF_USE_KERNEL_APC	Indicates that kernel asynchronous procedure calls should be used instead of worker threads to process the *TransmitFile* request. Note that kernel APCs can only be scheduled to run when the application is in a wait state (not necessarily an alertable wait state though).
TF_WRITE_BEHIND	Indicates that the *TransmitFile* request should return immediately even though the data may not have been acknowledged by the remote end. This flag should not be used with *TF_DISCONNECT* or *TF_REUSE_SOCKET*.

The *TransmitFile* function is useful for file-based I/O such as Web servers. In addition, one beneficial feature of *TransmitFile* is the capability of specifying the flags *TF_DISCONNECT* and *TF_REUSE_SOCKET*. When both of these flags are specified, the file and/or memory buffers are transmitted and the socket is disconnected once the send operation has completed. Also, the socket handle passed to the API can then be used as the client socket in *AcceptEx* or the connecting socket in *ConnectEx*. This is extremely beneficial because socket creation is very expensive. A server can use *AcceptEx* to handle client connections, then use *TransmitFile* to send data (specifying these flags), and afterward the socket handle may be used in a subsequent call to *AcceptEx*.

Note that you can call *TransmitFile* with a *NULL* file handle and *NULL* *lpTransmitBuffers* but still specify *TF_DISCONNECT* and *TF_REUSE_SOCKET*. This call will not send any data but allows the socket to be reused in *AcceptEx*. This is a good workaround for platforms that do not support the *DisconnectEx* API discussed later in this chapter. Finally, the *TransmitFile* function is available on Windows NT 4.0 and later version. Also, because *TransmitFile* is geared toward server applications, it is fully functional only on server versions of Windows. On home and professional versions, there may be only two outstanding *TransmitFile* (or *TransmitPackets*) calls at any given time. If there are more, then they are queued and not processed until the executing calls are finished.

TransmitPackets

The *TransmitPackets* extension is similar to *TransmitFile* because it too is used to send data. The difference between them is that *TransmitPackets* can send both files and memory buffers in any number and order. This function is defined as

```
BOOL
(PASCAL FAR * LPFN_TRANSMITPACKETS) (
    SOCKET hSocket,
    LPTRANSMIT_PACKETS_ELEMENT lpPacketArray,
    DWORD nElementCount,
    DWORD nSendSize,
    LPOVERLAPPED lpOverlapped,
    DWORD dwFlags
    );
```

The first parameter is the connected socket on which to send the data. Also, *TransmitPackets* works over datagram and stream-oriented protocols (such as TCP/IP and UDP/IP), unlike *TransmitFile*. The *lpPacketArray* is an array of one or more *TRANSMIT_PACKETS_ELEMENT* structures, which we'll define shortly. *nElementCount* simply indicates the number of members in the *TRANSMIT_PACKETS_ELEMENT* array. *nSendSize* is the same as the *nNumberOfBytesPerSend* parameter of *TransmtFile*. *lpOverlapped* indicates the overlapped structure is optional. *dwFlags* are the same as those for *TransmitFile*. See Table 6-1 for the options. The only exception is that the flag names begin with TP instead of TF—but their meanings are the same. And because *TransmitPackets* works over datagrams, the *TP_DISCONNECT* and *TP_REUSE_SOCKET* have no meaning for datagrams and specifying them will result in an error.

The *TRANSMIT_PACKETS_ELEMENT* structure is defined as

```
typedef struct _TRANSMIT_PACKETS_ELEMENT {
    ULONG dwElFlags;
#define TP_ELEMENT_MEMORY   1
#define TP_ELEMENT_FILE     2
#define TP_ELEMENT_EOP      4
    ULONG cLength;
    union {
        struct {
            LARGE_INTEGER nFileOffset;
            HANDLE        hFile;
        };
        PVOID             pBuffer;
    };
} TRANSMIT_PACKETS_ELEMENT, *PTRANSMIT_PACKETS_ELEMENT,
FAR *LPTRANSMIT_PACKETS_ELEMENT;
```

The first field indicates the type of buffer contained in this element, either memory or file as given by *TP_ELEMENT_MEMORY* and *TP_ELEMENT_FILE*, respectively. The *TP_ELEMENT_EOP* flag can be bitwise *OR*'ed in with one of the other two flags. It indicates that this element should not be combined with the following element in a single send operation. This allows the application to shape how the traffic is placed on the wire. The *cLength* field indicates how many bytes to transfer from the file's memory buffer. If the element contains a file pointer, then a *cLength* of zero indicates transmit the entire file. The union contains either a pointer to a buffer in memory or a handle to an open file as well as an offset value into that file. It is possible to reference the same file handle in multiple elements of the *TRANSMIT_PACKETS_ELEMENT*. In this case, the offset can specify where to begin the transfer. Alternately, a value of -1 indicates begin transmitting at the current file pointer position in that file.

A word of caution about using *TransmitPackets* with datagram sockets: the system is able to process and queue the send requests extremely fast, and it is possible that too many datagrams will pile up in the protocol driver. At this point, for unreliable protocols it is perfectly acceptable for the system to drop packets before they are even sent on the wire!

The *TransmitPackets* extension API is available on Windows XP and later version and is subject to the same type of limitation that *TransmitFile* is. On a non-server version of Windows NT, there can be only two outstanding *TransmitPackets* (or *TransmitFile*) calls at any given time.

ConnectEx

The *ConnectEx* extension function is a much-needed API available with Windows XP and later versions. This function allows for overlapped connect calls. Previously, the only way to issue multiple connect calls without using one thread for each connect was to use multiple non-blocking connects, which can be cumbersome to manage. This function is defined as

```
BOOL
(PASCAL FAR *LPFN_CONNECTEX) (
    IN SOCKET s,
       IN const struct sockaddr FAR *name,
       IN int namelen,
       IN PVOID lpSendBuffer,
       IN DWORD dwSendDataLength,
       OUT LPDWORD lpdwBytesSent,
       IN LPOVERLAPPED lpOverlapped
    );
```

The first parameter is a previously bound socket. The *name* parameter indicates the remote address to connect to and *namelen* is the length of that socket address structure. The *lpSendBuffer* is an optional pointer to a block of memory to send after the connection has been established, and *dwSendDataLength* indicates the number of bytes to send. *lpdwBytesSent* is updated to indicate the number of bytes sent successfully after the connection was established, if the operation completed immediately. *lpOverlapped* is the *OVERLAPPED* structure associated with this operation. This extension function can be called only in an overlapped manner.

Like with *AcceptEx* function, because *ConnectEx* is designed for performance, any previously set socket options or attributes are not automatically copied to the connected socket. To do so, the application must call *SO_UPDATE_CONNECT_CONTEXT* on the socket after the connection is established. In addition, as with *AcceptEx*, socket handles that have been "disconnected and re-used," either by *TransmitFile*, *TransmitPackets*, or *DisconnectEx*, may be used as the socket parameter to *ConnectEx*.

There isn't anything difficult about the *ConnectEx* API, and the only requirement is the socket passed into *ConnectEx* needs to be previously bound with a call to *bind*. There are no special flags, and it simply is an overlapped version of connect with the optional bonus of sending a block of data after the connection is established.

DisconnectEx

This extension API is simple. It takes a socket handle and performs a transport level disconnect and prepares the socket handle for re-use in a subsequent *AcceptEx* call. Both the *TransmitFile* and *TransmitPackets* APIs allow the socket to be disconnected and re-used after the send operation completes, but this standalone API was introduced for those applications that don't use either of those two APIs before shutting down. This extension API is available with Windows XP or later versions. However, for Windows 2000 or Windows NT 4.0 it is possible to call *TransmitFile* with a null file handle and buffers but specify the disconnect and re-use flags, which will achieve the same results. This API is defined as

```
typedef
BOOL
(PASCAL FAR * LPFN_DISCONNECTEX) (
    IN SOCKET s,
    IN LPOVERLAPPED lpOverlapped,
    IN DWORD  dwFlags,
    IN DWORD  dwReserved
    );
```

The first two parameters are self-explanatory. The *dwFlags* parameter can specify zero or *TF_REUSE_SOCKET*. If the flags are zero, then this function simply disconnects the connection. To be able to re-use the socket in *AcceptEx*, the *TF_REUSE_SOCKET* flag must be specified. The last parameter must be zero; otherwise, *WSAEINVAL* will be returned. If this function is invoked with an overlapped structure and if there are still pending operations on the socket, the *DisconnectEx* call will return *FALSE* with the error *WSA_IO_PENDING*. The operation will complete once all pending operations are finished and the transport level disconnect has been issued. Otherwise, if it is called in a blocking manner, the function will not return until pending I/O is completed and the disconnect has been issued. Note that the *DisconnectEx* function works only on connection-oriented sockets.

WSARecvMsg

This last extension function is not too interesting in the discussion of high-performance, scalable I/O, but it is new to Windows XP (and later versions) and we chose to be consistent and cover it with the rest of the extension APIs. The *WSARecvMsg* is nothing more than a complicated *WSARecv* with the exception that it returns information about which interface the packet was received on. This is useful for datagram sockets that are bound to the local wildcard address on a multihomed machine and need to know which interface a packet arrived on. This function is defined as

```
typedef
INT
(PASCAL FAR * LPFN_WSARECVMSG) (
    IN SOCKET s,
    IN OUT LPWSAMSG lpMsg,
    OUT LPDWORD lpdwNumberOfBytesRecvd,
    IN LPWSAOVERLAPPED lpOverlapped,
    IN LPWSAOVERLAPPED_COMPLETION_ROUTINE lpCompletionRoutine
    );
```

Most of the parameters are self-explanatory. Unlike the other extension functions, which cannot be called with an overlapped completion routine, this one can. The parameter that requires explaining is *lpMsg*. This is a WSAMSG structure that contains the buffers for receiving data as well as the informational buffers that will contain information about the data received. This structure is defined as

```
typedef struct _WSAMSG {
    LPSOCKADDR    name;             /* Remote address */
    INT           namelen;          /* Remote address length */
    LPWSABUF      lpBuffers;        /* Data buffer array */
    DWORD         dwBufferCount;    /* Number of elements in the array */
    WSABUF        Control;          /* Control buffer */
    DWORD         dwFlags;          /* Flags */
} WSAMSG, *PWSAMSG, * FAR LPWSAMSG;
```

The first field is a buffer that will contain the address of the remote system and *namelen* specifies how large the address buffer is. *lpBuffers* and *dwBufferCount* are the same as in *WSARecv*. The *Control* field specifies a buffer that will contain the optional control data. Lastly, *dwFlags* is also the same as in *WSARecv* and *WSARecvFrom*. However, there are additional flags that can be returned that provide information about the packet received. These new flags are described in Table 6-2.

Table 6-2 *Flags Returned from WSARecvMsg*

Flag	Description
MSG_BCAST	Datagram was received as a link-layer broadcast or with a destination address that was a broadcast address.
MSG_TRUNC	Datagram was truncated. There was more data that could be copied to the supplied receive buffer.
MSG_CTRUNC	Control data was truncated. The buffer supplied in the WSAMSG *Control* field was too small to receive the control data.

By default, no control information is returned when *WSARecvMsg* is called. To enable control information, one or more socket options must be set on the socket, indicating the type of information to be returned. Currently, only one option is supported, which is *IP_PKTINFO* for IPv4 and

IPV6_PKTINFO for IPv6. These options return information about which local interface the packet was received on. See Chapter 7 for more information about setting these options.

Once the appropriate socket option is set and the *WSARecvMsg* completes, the control information requested is returned via the *Control* buffer specified in the *WSAMSG* parameter. Each type of information requested is preceded by a *WSACMSGHDR* structure that indicates the type of information following as well as its size. This header structure is defined as

```
typedef struct _WSACMSGHDR {
    SIZE_T      cmsg_len;
    INT         cmsg_level;
    INT         cmsg_type;
    /* followed by UCHAR cmsg_data[] */
} WSACMSGHDR, *PWSACMSGHDR, FAR *LPWSACMSGHDR;
```

Within MSWSOCK.H, several useful macros are defined that extract the message headers and their data.

Scalable Server Architecture

Now that we've introduced the Microsoft-specific extensions, we'll get into the details of implementing a scalable server. Because this chapter focuses on connection-oriented protocols such as TCP/IP, we will first discuss accepting connections followed by managing data transfers. The last section will discuss resource management in more detail.

Accepting Connections

The most common action a server performs is accepting connections. The Microsoft extension *AcceptEx* is the only Winsock function capable of accepting a client connection via overlapped I/O. As we mentioned previously, the *AcceptEx* function requires that the client socket be created beforehand by calling *socket*. The socket must be unbound and unconnected, although it is possible to re-use socket handles after calling *TransmitFile*, *TransmitPackets*, or *DisconnectEx*.

A responsive server must always have enough *AcceptEx* calls outstanding so that incoming client connections may be handled immediately. However, there is no magic number of outstanding *AcceptEx* calls that will guarantee that the server will be able to accept the connection immediately. Remember that the TCP/IP stack will automatically accept connections on behalf of the listening application, up to the backlog limit. For Windows NT Server, the maximum backlog value is currently 200. If a server posts 15 *AcceptEx* calls and then a burst of 50 clients connect to the server, none of the clients' connections will be rejected. The server's accept calls will satisfy the first 15 connections and the system will accept the remaining connections silently—this dips into the backlog amount so that the server will be able to accept 165 additional connections. Then when the server posts additional *AcceptEx* calls, they will succeed immediately because one of the system queued connections will be returned.

The nature of the server plays an important role in determining how many *AcceptEx* operations to post. For example, a server that is expected to handle many short-lived connections from a great number of clients may want to post more concurrent *AcceptEx* operations than a server that handles fewer connections with longer lifetimes. A good strategy is to allow the number of *AcceptEx* calls to vary between a low and high watermark. An application can keep track of the number of outstanding *AcceptEx* operations that are pending. Then, when one or more of those completes and the outstanding count decreases below the set watermark, additional *AcceptEx* calls may be posted. Of course, if at some point an *AcceptEx* completes and the number of outstanding accepts is greater than or equal to the high watermark then no additional calls should be posted in the handling of the current *AcceptEx*.

On Windows 2000 and later versions, Winsock provides a mechanism for determining if an application is running behind in posting adequate *AcceptEx* calls. When creating the listening socket, associate it with an event by using the *WSAEventSelect* API call and registering for *FD_ACCEPT* notification. If there are no pending *AcceptEx* operations but there are incoming client connections (accepted by the system according to the backlog value), then the event will be signaled. This can even be used as an indication to post additional *AcceptEx* operations.

One significant benefit of using *AcceptEx* is the capability to receive data in addition to accepting the client connection. For servers whose clients send an initial request this is ideal. However, as we mentioned in Chapter 5, the *AcceptEx* operation will not complete until at least one byte of data has been received. To prevent malicious attacks or stale connections, a server should cycle through all client socket handles in outstanding *AcceptEx* operations and call *getsockopt* with *SO_CONNECT_TIME*, which will return regardless of whether the socket is actually connected. If it is connected, the return value is greater than zero. A value of -1 indicates it is not connected. If the *WSAEventSelect* suggestion is implemented, then when the event is signaled it is a good time to check whether the client socket handles in outstanding accept calls are connected. Once an *AcceptEx* call accepts an incoming connection, it will then wait to receive data, and at this point there is one less outstanding accept call. Once there are no remaining accepts, the event will be signaled on the next incoming client connection. As a word of warning, applications should not under any circumstances close a client socket handle used in an *AcceptEx* call that has not been accepted because it can lead to memory leaks. For performance reasons, the kernel-mode structures associated with an *AcceptEx* call will not be cleaned up when the unconnected client handle is closed until a new client connection is established or until the listening socket is closed.

Although it may seem logical and simpler to post *AcceptEx* requests in one of the worker threads handling notification from the completion port, you should avoid this because socket creation process is expensive. In addition, any complex computations should be avoided within the worker threads so the server may process the completion notifications as fast as possible. One reason socket creation is expensive is the layered architecture of Winsock 2.0. When the server creates a socket, it may be routed through multiple providers, each performing their own tasks, before the socket is created and returned to the application. Chapter 12 discusses layered providers in detail. Instead, a server should create client sockets and post *AcceptEx* operations from a separate thread. When an overlapped *AcceptEx* completes in the worker thread, an event can be used to signal the accept issuing thread.

Data Transfers

Once clients are connected, the server will need to transfer data. This process is fairly straightforward, and once again, all data sent or received should be performed with overlapped I/O. By default, each socket has an associated send and receive buffer that is used to buffer outgoing and incoming data, respectively. In most cases these buffers should be left alone, but it is possible to change them or set them to zero by calling *setsockopt* with the *SO_SNDBUF* or *SO_RCVBUF* options.

Let's look at how the system handles a typical send call when the send buffer size is non-zero. When an application makes a send call, if there is sufficient buffer space, the data is copied into the socket's send buffers, the call completes immediately with success, and the completion is posted. On the other hand, if the socket's send buffer is full, then the application's send buffer is locked and the send call fails with *WSA_IO_PENDING*. After the data in the send buffer is processed (for example, handed down to TCP for processing), then Winsock will process the locked buffer directly. That is, the data is handed directly to TCP from the application's buffer and the socket's send buffer is completely bypassed.

The opposite is true for receiving data. When an overlapped receive call is performed, if data has already been received on the connection, it will be buffered in the socket's receive buffer. This data will be copied directly into the application's buffer (as much as will fit), the receive call returns success, and a completion is posted. However, if the socket's receive buffer is empty, when the overlapped receive call is made, the application's buffer is locked and the call fails with *WSA_IO_PENDING*. Once data arrives on the connection, it will be copied directly into the application's buffer, bypassing the socket's receive buffer altogether.

Setting the per-socket buffers to zero generally will not increase performance because the extra memory copy can be avoided as long as there are always enough overlapped send and receive operations posted. Disabling the socket's send buffer has less of a performance impact than disabling the receive buffer because the application's send buffer will always be locked until it can be passed down to TCP for processing. However, if the receive buffer is set to zero and there are no outstanding overlapped receive calls, any incoming data can be buffered only at the TCP level. The TCP driver will buffer only up to the receive window size, which is 17 KB—TCP will increase these buffers as needed to this limit; normally the buffers are much smaller. These TCP buffers (one per connection) are allocated out of non-paged pool, which means if the server has 1000 connections and no receives posted at all, 17 MB of the non-paged pool will be consumed! The non-paged pool is a limited resource, and unless the server can guarantee there are always receives posted for a connection, the per-socket receive buffer should be left intact.

Only in a few specific cases will leaving the receive buffer intact lead to decreased performance. Consider the situation in which a server handles many thousands of connections and cannot have a receive posted on each connection (this can become very expensive, as you'll see in the next section). In addition, the clients send data sporadically. Incoming data will be buffered in the per-socket receive buffer and when the server does issue an overlapped receive, it is performing unnecessary work. The overlapped operation issues an I/O request packet (IRP) that completes, immediately after which notification is sent to the completion port. In this case, the server cannot keep enough receives posted, so it is better off performing simple non-blocking receive calls.

TransmitFile and *TransmitPackets*

For sending data, servers should consider using the *TransmitFile* and *TransmitPackets* API functions where applicable. The benefit of these functions is that a great deal of data can be queued for sending on a connection while incurring just a single user-to-kernel mode transition. For example, if the server is sending file data to a client, it simply needs to open a handle to that file and issue a single *TransmitFile* instead of calling *ReadFile* followed by a *WSASend*, which would invoke many user-to-kernel mode transitions. Likewise, if a server needs to send several memory buffers, it also can build an array of *TRANSMIT_PACKETS_ELEMENT* structures and use the *TransmitPackets* API. As we mentioned, these APIs allow you to disconnect and re-use the socket handles in subsequent *AcceptEx* calls.

Resource Management

On a machine with sufficient resources, a Winsock server should have no problem handling thousands of concurrent connections. However, as the server handles increasingly more concurrent connections, a resource limitation will eventually be encountered. The two limits most likely to be encountered are the number of locked pages and non-paged pool usage. The locked pages limitation is less serious and more easily avoided than running out of the non-paged pool.

With every overlapped send or receive operation, it is probable that the data buffers submitted will be locked. When memory is locked, it cannot be paged out of physical memory. The operating system imposes a limit on the amount of memory that may be locked. When this limit is reached, overlapped operations will fail with the *WSAENOBUFS* error. If a server posts many overlapped receives on each connection, this limit will be reached as the number of connections grow. If a server anticipates handling a very high number of concurrent clients, the server can post a single zero byte receive on each connection. Because there is no buffer associated with the receive operation, no memory needs to be locked. With this approach, the per-socket receive buffer should be left intact because once the zero-byte receive operation completes, the server can simply perform a non-blocking receive to retrieve all the data buffered in the socket's receive buffer. There is no more data pending when the non-blocking receive fails with *WSAEWOULDBLOCK*. This design would be for servers that require the maximum possible concurrent connections while sacrificing the data throughput on each connection.

Of course, the more you are aware of how the clients will be interacting with the server, the better. In the previous example, a non-blocking receive is performed once the zero-byte receive completes to retrieve the buffered data. If the server knows that clients send data in bursts, then once the zero-byte receive completes, it may post one or more overlapped receives in case the client sends a substantial amount of data (greater than the per-socket receive buffer that is 8 KB by default).

Another important consideration is the page size on the architecture the server is running on. When the system locks memory passed into overlapped operations, it does so on page boundaries. On the x86 architecture, pages are locked in multiples of 4 KB. If an operation posts a 1 KB buffer, then the system is actually locking a 4 KB chunk of memory. To avoid this waste, overlapped send and receive buffers should be a multiple of the page size. The Windows API *GetSystemInfo* can be called to obtain the page size for the current architecture.

Hitting the non-paged pool limit is a much more serious error and is difficult to recover from. Non-paged pool is the portion of memory that is always resident in physical memory and can never be paged out. Kernel-mode operating system components, such as a driver, typically use the non-paged pool that includes Winsock and the protocol drivers such as *tcpip.sys*. Each socket created consumes a small portion of non-paged pool that is used to maintain socket state information. When the socket is bound to an address, the TCP/IP stack allocates additional non-paged pool for the local address information. When a socket is then connected, a remote address structure is also allocated by the TCP/IP stack. In all, a connected socket consumes about 2 KB of non-paged pool and a socket returned from *accept* or *AcceptEx* uses about 1.5 KB of non-paged pool (because an accepted socket needs only to store the remote address). In addition, each overlapped operation issued on a socket requires an I/O request packet to be allocated, which uses approximately 500 non-paged pool bytes.

As you can see, the amount of non-paged pool each connection uses is not great; however, as the number of clients connecting increases, the amount of non-paged pool the server uses can be significant. For example, consider a server running Windows 2000 (or greater) with 1 GB physical memory. For this amount of memory there will be 256 MB set aside for the non-paged pool. In general, the amount of non-paged pool allocated is one quarter the amount of physical memory with a 256 MB limit on Windows 2000 and later versions and a limit of 128 MB on Windows NT 4.0. With 256 MB of non-paged pool, it is possible to handle 50,000 or more connections, but care must be taken to limit the number of overlapped operations queued for accepting new connections as well as sending and receiving on existing connections. In this example, the connected sockets alone consume 75 MB on non-paged pool (assuming each socket uses 1.5 KB of non-paged pool as mentioned). Therefore, if the zero-byte overlapped receive strategy is used, then a single IRP is allocated for each connection, which uses another 25 MB of non-paged pool.

If the system does run out of non-paged pool, there are two possibilities. In the best-case scenario, Winsock calls will fail with *WSAENOBUFS*. The worst-case scenario is the system crashes with a terminal error. This typically occurs when a kernel mode component (such as a third-party driver) doesn't handle a failed memory allocation correctly. As such there is no guaranteed way to recover from exhausting the non-paged pool, and furthermore, there is no reliable way of monitoring the available amount of non-paged pool because any kernel mode component can chew up non-paged pool. The main point of this discussion is that there is no magical or programmatic method of determining how many concurrent connections and overlapped operations are acceptable. Also, it is virtually impossible to determine whether the system has run out of non-paged pool or exceeded the locked page count because both will result in Winsock calls failing with *WSAENOBUFS*. Testing must be performed on the server. Because of these factors, the developer must test the server's performance with varying numbers of concurrent connections and overlapped operations in order to find a happy medium. If programmatic limits are imposed to prevent the server from exhausting non-paged pool, you will know that any *WSAENOBUFS* failures are generally the result of exceeding the locked page limit, and that can be handled in a graceful manner programmatically, such as further restricting the number of outstanding operations or closing some of the connections.

Server Strategies

In this section, we'll take a look at several strategies for handling resources depending on the nature of the server. Also, the more control you have over the design of the client and server allows you to design both accordingly to avoid the limitations and bottlenecks discussed previously. Again, there is no foolproof method that will work 100 percent in all situations. Servers can be divided roughly into two categories: high throughput and high connections. A high throughput server is more concerned with pushing data on a small number of connections. Of course, the meaning of the phrase “small number of connections” is relative to the amount of resources available on the server. A high connection server is more concerned with handling a large number of connections and is not attempting to push large data amounts.

In the next two sections, we'll discuss both high throughput and high connection server strategies. After that, we'll look at performance numbers gathered from the server samples provided on the companion CD.

High Throughput

An FTP server is an example of a high throughput server. It is concerned with delivering bulk content. In this case, the server is concerned with processing each connection to minimize the amount of time required to transfer the data. To do so, the server must limit the number of concurrent connections because the greater the simultaneous connections, the lower the throughput will be on each connection. An example would be an FTP server that refuses a connection because it is too busy.

The goal for this strategy is I/O. The server should keep enough receives or sends posted to maximize throughput. Because each overlapped I/O requires memory to be locked as well as a small portion of non-paged pool for each IRP associated with the operation, it is important to limit I/O to a small set of connections. It is possible for the server to continually accept connections and have a relatively high number of established connections, but I/O must be limited to a smaller set.

In this case, the server may post a number of sends or receives on a subset of the established clients. For example, the server could handle client connections in a first-in, first-out manner and post a number of overlapped sends and/or receives on the first 100 connections. After those clients are handled, the server can move on the next set of clients in the queue. In this model, the number of outstanding send and receive operations are limited to a smaller set of connections. This prevents the server from blindly posting I/O operations on every connection, which could quickly exhaust the server's resources.

The server should take care to monitor the number of operations outstanding on each connection so it may prevent malicious clients from attacking it. For example, a server designed to receive data from a client, process it, and send some sort of response should keep track of how many sends are outstanding. If the client is simply flooding the server with data but not posting any receives, the server may end up posting dozens of overlapped sends that will never complete. In this case, once the server finds that there are too many outstanding operations, it can close the connection.

Maximizing Connections

Maximizing the number of concurrent client connections is the more difficult of the two strategies. Handling the I/O on each connection becomes difficult. A server cannot simply post one or more sends or receives on each connection because the amount of memory (both in terms of locked pages and non-paged pool) is great. In this scenario, the server is interested in handling many connections at the expense of throughput on each connection. An example of this would be an instant messenger server. The server would handle many thousands of connections but would need to send or receive only a small number of bytes at a time.

For this strategy, the server does not necessarily want to post an overlapped receive on each connection because this would involve locking many pages for each of the receive buffers. Instead, the server can post an overlapped zero-byte receive. Once the receive completes, the server would perform a non-blocking receive until *WSAEWOUDLBLOCK* is returned. This allows the server to immediately receive all buffered data received on that connection. Because this model is geared toward clients that send data intermittently, it minimizes the number of locked pages but still allows processing of data on each connection.

Performance Numbers

This section covers performance numbers from the different servers provided in Chapters 5 and 6. The various servers tested are those using blocking sockets, non-blocking with *select, WSAAsyncSelect*, *WSAEventSelect*, overlapped I/O with events, and overlapped I/O with completion ports. Table 6-3 summarizes the results of these tests. For each I/O model, there are a couple of entries. The first entry is where 7000 connections were attempted from three clients. For all of these tests, the server is an echo server. That is, for each connection that is accepted, data is received and sent back to the client. The first entry for each I/O model represents a high-throughput server where the client sends data as fast as possible to the server. Each of the sample servers does not limit the number of concurrent connections. The remaining entries represent the connections when the clients limit the rate in which they send data so as to not overrun the bandwidth available on the network. The second entry for each I/O model represents 12,000 connections from the client, which is rate limiting the data sent. If the server was able to handle the majority of the 12,000 connections, then the third entry is the maximum number of clients the server was able to handle.

As we mentioned, the servers used are those provided from Chapter 5 except for the I/O completion port server, which is a slightly modified version of the Chapter 5 completion port server except that it limits the number of outstanding operations. This completion port server limits the number of outstanding send operations to 200 and posts just a single receive on each client connection. The client used in this test is the I/O completion port client from Chapter 5. Connections were established in blocks of 1000 clients by specifying the '-c 1000' option on the client. The two *x86*-based clients initiated a maximum of 12,000 connections and the *Itanium* system was used to establish the remaining clients in blocks of 4000. In the tests that were rate limited, each client block was limited to 200,000 bytes per second (using the '-r 200000' switch). So the average send throughput for that entire block of clients was limited to 200,000 bytes per second (not that each client was limited to this amount).

Table 6-3 *I/O Method Performance Comparison*

I/O Model	Attempted/Connected	Memory Used (KB)	Non-Paged Pool	CPU Usage	Threads	Throughput (Send/ Receive Bytes Per Second)
Blocking	7000/ 1008	25,632	36,121	10–60%	2016	2,198,148/ 2,198,148
	12,000/ 1008	25,408	36,352	5–40%	2016	404,227/ 402,227
Non-blocking	7000/ 4011	4208	135,123	95–100%*	1	0/0
	12,000/ 5779	5224	156,260	95–100%*	1	0/0
WSA-Async Select	7000/ 1956	3640	38,246	75–85%	3	1,610,204/ 1,637,819
	12,000/ 4077	4884	42,992	90–100%	3	652,902/ 652,902
WSA-Event Select	7000/ 6999	10,502	36,402	65–85%	113	4,921,350/ 5,186,297
	12,000/ 11,080	19,214	39,040	50–60%	192	3,217,493/ 3,217,493
	46,000/ 45,933	37,392	121,624	80–90%	791	3,851,059/ 3,851,059
Overlapped (events)	7000/ 5558	21,844	34,944	65–85%	66	5,024,723/ 4,095,644
	12,000/12,000	60,576	48,060	35–45%	195	1,803,878/ 1,803,878
	49,000/48,997	241,208	155,480	85–95%	792	3,865,152/ 3,834,511
Overlapped (completion port)	7000/ 7000	36,160	31,128	40–50%	2	6,282,473/ 3,893,507
	12,000/12,000	59,256	38,862	40–50%	2	5,027,914/ 5,027,095
	50,000/49,997	242,272	148,192	55–65%	2	4,326,946/ 4,326,496

The server was a Pentium 4 1.7 GHz Xeon with 768 MB memory. Clients were established from three machines: Pentium 2 233MHz with 128 MB memory, Pentium 2 350 MHz with 128 MB memory, and an Itanium 733 MHz with 1 GB memory. The test network was a 100 MB isolated hub. All of the machines tested had Windows XP installed.

The blocking model is the poorest performing of all the models. The blocking server spawns two threads for each client connection: one for sending data and one for receiving it. In both test cases, the server was unable to handle a fraction of the connections because it hit a system resource limit on creating threads. Thus the *CreateThread* call was failing with *ERROR_NOT_ENOUGH_MEMORY*. The remaining client connections failed with *WSAECONNREFUSED*.

The non-blocking model faired only somewhat better. It was able to accept more connections but ran into a CPU limitation. The non-blocking server puts all the connected sockets into an *FD_SET*, which is passed into *select*. When *select* completes, the server uses the *FD_ISSET* macro to search to determine if that socket is signaled. This becomes inefficient because the number of connections increases. Just to determine if a socket is signaled, a linear search through the array is required! To partially alleviate this problem, the server can be redesigned so that it iteratively steps through the *FD_SET*s returned from *select*. The only issue is that the server then needs to be able to quickly find the *SOCKET_INFO* structure associated with that socket handle. In this case, the server can provide a more sophisticated cataloging mechanism, such as a hash tree, which allows quicker lookups. Also note that the non-paged pool usage is extremely high. This is because both AFD and TCP are buffering data on the client connections because the server is unable to read the data fast enough (as indicated by the zero-byte throughput) as indicated by the high CPU usage.

The *WSAAsyncSelect* model is acceptable for a small number of clients but does not scale well because the overhead of the message loop quickly bogs down its capability to process messages fast enough. In both tests, the server is able to handle only about a third of the connections made. The clients receive many *WSAECONNREFUSED* errors indicating that the server cannot handle the *FD_ACCEPT* messages quickly enough so the listen backlog is not exhausted. However, even for those connections accepted, you will notice that the average throughput is rather low (even in the case of the rate limited clients).

Surprisingly, the *WSAEventSelect* model performed very well. In all the tests, the server was, for the most part, able to handle all the incoming connections while obtaining very good data throughput. The drawback to this model is the overhead required to manage the thread pool for new connections. Because each thread can wait on only 64 events, when new connections are established new threads have to be created to handle them. Also, in the last test case in which more than 45,000 connections were established, the machine became very sluggish. This was most likely due to the great number of threads created to service the many connections. The overhead for switching between the 791 threads becomes significant. The server reached a point at which it was unable to accept any more connections due to numerous *WSAENOBUFS* errors. In addition, the client application reached its limitation and was unable to sustain the already established connections (we'll discuss this in detail later).

The overlapped I/O with events model is similar to the *WSAEventSelect* in terms of scalability. Both models rely on thread pools for event notification, and both reach a limit at which the thread switching overhead becomes a factor in how well it handles client communication. The performance numbers for this model almost exactly mirror that of *WSAEventSelect*. It does surprisingly well until the number of threads increases.

The last entry is for overlapped I/O with completion ports, which is the best performing of all the I/O models. The memory usage (both user and non-paged pool) and accepted clients are similar to both the overlapped I/O with events and *WSAEventSelect* model. However, the real difference is in CPU usage. The completion port model used only around 60 percent of the CPU, but the other two models required substantially more horsepower to maintain the same number of connections. Another significant difference is that the completion port model also allowed for slightly better throughput.

While carrying out these tests, it became apparent that there was a limitation introduced due to the nature of the data interaction between client and server. The server is designed to be an echo server such that all data received from the client was sent back. Also, each client continually sends data (even if it's at a lower rate) to the server. This results in data always pending on the server's socket (either in the TCP buffers or in AFD's per-socket buffers, which are all non-paged pool). For the three well-performing models, only a single receive is performed at a time; however, this means that for the majority of the time, there is still data pending. It is possible to modify the server to perform a non-blocking receive once data is indicated on the connection. This would drain the data buffered on the machine. The drawback to this approach in this instance is that the client is constantly sending and it is possible that the non-blocking receive could return a great deal of data, which would lead to starvation of other connections (as the thread or completion thread would not be able to handle other events or completion notices). Typically, calling a non-blocking receive until *WSAEWOULDBLOCK* works on connections where data is transmitted in intervals and not in a continuous manner.

From these performance numbers it is easily deduced that *WSAEventSelect* and overlapped I/O offer the best performance. For the two event based models, setting up a thread pool for handling event notification is cumbersome but still allows for excellent performance for a moderately stressed server. Once the connections increase and the number of threads increases, then scalability becomes an issue as more CPU is consumed for context switching between threads. The completion port model still offers the ultimate scalability because CPU usage is less of a factor as the number of clients increases.

Winsock Direct and Sockets Direct Protocol

Winsock Direct is a high-speed interconnect introduced on Windows 2000 Datacenter Server. It is a protocol that runs over special hardware available from several vendors, such as Giganet, Compaq, and others. What is so special about Winsock Direct is that it completely bypasses the TCP stack and goes directly to the network interface card, which allows for extremely high-speed data communications. The advantage of Winsock Direct is that it is completely transparent to a TCP Winsock application. That is, if a TCP application is run on a machine with a Winsock Direct capable card, it transparently goes over the Winsock Direct route (given that it is the appropriate route) instead of over a regular Ethernet connection.

The Sockets Direct Protocol is the next evolution of the Winsock Direct protocol. It is designed to run over Infiniband-compatible hardware available in future releases of the Windows operating system. The on the wire protocol is slightly different than that of Winsock Direct but it is still transparent to the applications.

Because Winsock Direct is designed to be transparent, the same issues encountered with “regular” Winsock applications still apply when running over Winsock Direct. Applications still have to manage the number of outstanding overlapped operations so as to not exceed the locked pages or non-paged pool limits.

Conclusion

This chapter focused on writing high-performance, scalable Winsock servers for Windows NT–based operating systems. We discussed several of the Microsoft-specific Winsock extensions that greatly aid programmers in developing these servers. In addition, we covered several approaches to accepting connections so as to minimize the chance a client will receive a connection refused as well as how throughput can be maximized. Afterward we covered resource management, which is the core concept required to writing high performance servers. Finally, we compared the performance of the various I/O models introduced in Chapter 5 to see how well they scale when many client connections are attempted.

Chapter 7

Socket Options and Ioctls

Once a socket has been created, various attributes can be manipulated with socket options and ioctl commands to affect the socket's behavior. Some of these options simply return information, and others affect the way the socket behaves in your application. An ioctl is an I/O control command that also affects the behavior of the socket. This chapter is dedicated to discussing four Winsock functions: *getsockopt*, *setsockopt*, *ioctlsocket*, and *WSAIoctl*. Each function has numerous commands, many of which have never been properly documented. In the following sections, we will discuss the required parameters and available options for each function as well as the platforms that support those options. Every option is assumed to work on all Windows platforms (Windows CE, Windows 95, Windows 98, Windows Me, Windows NT, Windows 2000, and Windows XP) unless otherwise noted. The only exception occurs when an option requires Winsock 2. Because Winsock 2 is not available on every platform, Winsock 2 ioctl commands and options are not supported on Windows CE or Windows 95 (unless the Winsock 2 update has been applied to Windows 95). Furthermore, remember that Windows CE does not support any protocol-specific options not pertaining to TCP/IP.

Most of these ioctl commands and options are defined in either WINSOCK.H or WINSOCK2.H, depending on whether they are specific to Winsock 1 or Winsock 2; however, a few of the options are specific either to the Microsoft provider or to a particular transport protocol. Microsoft-specific extensions are defined in WINSOCK2.H and MSWSOCK.H. Transport provider extensions are defined in their protocol-specific header files. For the transport-specific options, we will indicate the correct header file along with the option. Applications using the Microsoft-specific extensions must link with MSWSOCK.LIB.

Socket Options

The *getsockopt* function is most frequently used to get information about the given socket. The prototype for this function is

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

The first parameter, *s*, is the socket on which you want to perform the specified option. This must be a valid socket for the given protocol you are using. A number of options are specific to a particular protocol and socket type, while others pertain to all types of sockets. This ties in with the second parameter, *level*. An option of level *SOL_SOCKET* means it is a generic option that isn't necessarily specific to a given protocol. We say "necessarily" because not all protocols implement each socket option of level *SOL_SOCKET*. For example, *SO_BROADCAST* puts the socket into broadcast mode, but not all supported protocols support the notion of broadcast sockets. The *optname* parameter is the actual option you are interested in. These option names are constant values defined in the Winsock header files. The most common and protocol-independent options (such as those with the *SOL_SOCKET* level) are defined in WINSOCK.H and WINSOCK2.H. Each specific protocol has its own header file that defines options specific to it. Finally, the *optval* and *optlen* parameters are the variables returned with the value of the desired option. In most cases—but not all—the option value is an integer.

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

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

The parameters are the same as in *getsockopt* except that you pass in a value as the *optval* and *optlen* parameters, which are the values to set for the specified option. As with *getsockopt*, *optval* is often, but not always, an integer. Consult each option for the specifics on what is passed as the option value.

The most common mistake associated with calling either *getsockopt* or *setsockopt* is attempting to obtain socket information for a socket whose underlying protocol doesn't possess that particular characteristic. For example, a socket of type *SOCK_STREAM* is not capable of broadcasting data; therefore, attempting to set or get the *SO_BROADCAST* option results in the error *WSAENOPROTOOPT*.

SOL_SOCKET Option Level

This section describes the socket options that return information based on the socket's characteristics and are not specific to that socket's protocol.

SO_ACCEPTCONN

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

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

SO_BROADCAST

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

If the given socket has been configured for sending or receiving broadcast data, querying this socket option returns *TRUE*. Use *setsockopt* with *SO_BROADCAST* to enable broadcast capabilities on the socket. This option is valid for sockets that aren't of type *SOCK_STREAM*.

Broadcasting is the capability to send data so that every machine on the local subnet receives the data. Of course, there must be some process on each machine that listens for incoming broadcast data. The drawback of broadcasting is that if many processes are all sending broadcast data, the network can become saturated and network performance suffers. To receive a broadcast message, you must enable the broadcast option and then use one of the datagram receive functions, such as *recvfrom* or *WSARecvfrom*. You can also connect the socket to the broadcast address by calling *connect* or *WSAConnect* and then use *recv* or *WSARecv*. For UDP broadcasts, you must specify a port number to send the datagram to; likewise, the receiver must request to receive the broadcast data on that port also. The following code example illustrates how to send a broadcast message with UDP.

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

s = WSASocket(AF_INET, SOCK_DGRAM, 0, NULL, 0, WSA_FLAG_OVERLAPPED);
bBroadcast = TRUE;
setsockopt(s, SOL_SOCKET, SO_BROADCAST, (char *)&bBroadcast,
    sizeof(BOOL));
bcast.sin_family = AF_INET;
bcast.sin_addr.s_addr = inet_addr(INADDR_BROADCAST);
bcast.sin_port = htons(5150);
sendto(s, sMsg, strlen(sMsg), 0, (SOCKADDR *)&bcast, sizeof(bcast));
```

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

AppleTalk is another protocol capable of sending broadcast messages. AppleTalk also has a special address used by broadcast data. You learned in Chapter 4 that an AppleTalk address has three parts: network, node, and socket (destination). For broadcasting, set the destination to *ATADDR_BROADCAST* (0xFF), which causes the datagram to be sent to all endpoints on the given network.

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

SO_CONDITIONAL_ACCEPT

***optval* Type**	**Get/Set**	**Winsock Version**	**Description**
BOOL	Both	2+	Allows true connection acceptance or rejection from *WSAAccept*.

This option allows applications to conditionally accept or reject incoming connections at the protocol level. For example, by default, this option is off for TCP and any incoming SYN packet is acknowledged with an ACK+SYN even if the application has not called the *accept*, *WSAAccept*, or *AcceptEx* functions. When this option is enabled, the connection is not acknowledged until the application calls one of the accept functions. In the case of *WSAAccept* and a conditional accept function, the connection is not acknowledged until the conditional accept function returns CF_ACCEPT. This option must be set before the *listen* call.

The drawback to enabling this is if the application does not post an accept or return CF_ACCEPT in a timely fashion, the client's connection request will time out with an error (*WSAETIMEDOUT*). For TCP/IP this option is off by default while for ATM, it is enabled by default. This option is available on Windows 2000 and later versions.

SO_CONNECT_TIME

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

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

This option is especially relevant in the case of *AcceptEx*. If an application posts an *AcceptEx* with a receive buffer, then the *AcceptEx* will not complete until data is received on the client connection. A malicious application could perform a denial of service attack by making many connections without sending data. To prevent this, the server should cycle through all client sockets outstanding in *AcceptEx* calls to see if they have been connected but the accept has not completed. Refer to Chapter 6 for more details.

SO_DEBUG

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

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

SO_DONTLINGER

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

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

SO_DONTROUTE

optval		Winsock	

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

The *SO_DONTROUTE* option tells the underlying network stack to ignore the routing table and to send the data out on the interface the socket is bound to. For example, if you create a IPv4 UDP socket and bind it to interface A and then send a packet destined for a machine on the network attached to interface B, the packet will in fact be routed so that it is sent on interface B. Using *setsockopt* with the Boolean value *TRUE* prevents this because the packet goes out on the bound interface. The *getsockopt* function can be called to determine if routing is enabled (which it is by default).

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

SO_ERROR

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

The *SO_ERROR* option returns and resets the per-socket–based error code, which is different from the per-thread–based error code that is handled using the *WSAGetLastError* and *WSASetLastError* function calls. A successful call using the socket does not reset the per-socket–based error code returned by the *SO_ERROR* option. Calling this option will not fail; however, the error value is not always updated immediately, so there is a possibility of this option returning 0 (indicating no error). It is best to use *WSAGetLastError*.

SO_ EXCLUSIVEADDRUSE

optval Type	Get/Set	Winsock Version	Description
BOOL	Set only	2+	If *TRUE*, the local port that the socket is bound to cannot be reused by another process.

This option is the complement of *SO_REUSEADDR*, which we will describe shortly. This option exists to prevent other processes from using the *SO_REUSEADDR* on a local address that your application is using. If two separate processes are bound to the same local address (assuming that *SO_REUSEADDR* is set earlier), which of the two sockets receives notifications for incoming connections is not defined. The *SO_EXCLUSIVEADDRUSE* option locks down the local address to which the socket is bound, so if any other process tries to use *SO_REUSEADDR* with the same local address, that process fails. Administrator rights are required to set this option. It is available on only Windows 2000 or later versions.

SO_KEEPALIVE

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

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

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

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

In Windows NT, store the keys under

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

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

SO_LINGER

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

SO_LINGER controls the action taken when unsent data is queued on a socket and a *closesocket* is performed. A call to *getsockopt* with this socket option returns the current linger times in a linger structure, which is defined as

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

A nonzero value for *l_onoff* means that lingering is enabled, and *l_linger* is the timeout in seconds, at which point any pending data to be sent or received is discarded and the connection with the peer is reset. Conversely, you can call *setsockopt* to turn lingering on and specify the length of time before discarding any queued data. This is accomplished by setting the desired values in a variable of type *struct linger*. When setting a linger value with *setsockopt*, you must set the *l_onoff* field of the structure to a nonzero value. To turn lingering off once it has been enabled, you can call *setsockopt* with the *SO_LINGER* option and the *l_onoff* field of the *linger* structure set to 0, or call *setsockopt* with the *SO_DONTLINGER* option, passing the value *TRUE* for the *optval* parameter. Sockets of type *SOCK_DGRAM* do not support this option.

Setting the linger option directly affects how a connection behaves when the *closesocket* function is called. Table 7-1 lists these behaviors.

Table 7-1 *Linger Options*

Option	Interval	Type of Close	Wait for Close?
SO_DONTLINGER	Not applicable	Graceful	No
SO_LINGER	0	Hard	No
SO_LINGER	Non-zero	Graceful	Yes

If *SO_LINGER* is set with a zero timeout interval (that is, the *linger* structure member *l_onoff* is not 0 and *l_linger* is 0), *closesocket* is not blocked, even if queued data has not yet been sent or acknowledged. This is called a hard, or abortive, close because the socket's virtual circuit is reset immediately and any unsent data is lost. Any receive call on the remote side of the circuit fails with *WSAECONNRESET*.

If *SO_LINGER* is set with a nonzero timeout interval on a blocking socket, the *closesocket* call blocks on a blocking socket until the remaining data has been sent or until the timeout expires. This is called a graceful disconnect. If the timeout expires before all data has been sent, the Windows Sockets implementation terminates the connection before *closesocket* returns.

SO_MAX_MSG_SIZE

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

This is a get-only socket option that indicates the maximum outbound (send) size of a message for message-oriented socket types as implemented by a particular service provider. It has no meaning for byte-stream–oriented sockets.

SO_OOBINLINE

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

By default, OOB data is not inlined, so a call to a receive function (with the appropriate *MSG_OOB* flag set) returns the OOB data in a single call. If this option is set, the OOB data appears within the data stream returned from a receive call, and a call to *ioctlsocket* with the *SIOCATMARK* option is required to determine which byte is the OOB data. Sockets of type *SOCK_DGRAM* do not support this option. See Chapter 1 for more details on OOB data.

SO_OPENTYPE

optval Type	Get/Set	Winsock Version	Description
BOOL	Both	1+	When set, subsequent calls to *socket* will return non-overlapped handles.

The *socket* API returns handles that are overlapped capable by default. However, if the socket is to be used in the C runtime routines, the socket must have been created without the overlapped flag. With Winsock 2, the *WSASocket* function can be called without specifying the *WSA_FLAG_OVERLAPPED* flag. Otherwise, this socket option may be set, which affects all subsequent calls to *socket* from the current thread so that non-overlapped handles are returned. To set this option, *INVALID_SOCKET* is passed for the *SOCKET* parameter to *setsockopt* with a non-zero *optval*. To change it back, specify a zero *optval*. A socket created non-overlapped cannot be used with an overlapped fashion (such as completion routines or completion ports)—the operation will fail.

SO_PROTOCOL_INFO

optval Type	Get/Set	Winsock Version	Description
WSAPROTOCOL_INFO	Get only	2+	Returns the Winsock catalog entry for the socket's protocol

This is another get-only option that fills in the supplied *WSAPROTOCOL_ INFO* structure with the characteristics of the protocol associated with the socket. See Chapter 2 for a description of the *WSAPROTOCOL_INFO* structure and its member fields.

SO_RCVBUF

optval Type	Get/Set	Winsock Version	Description
			Gets or sets the per-socket buffer size for receive

Int	Both	1+	operations

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

One possible reason for changing the buffer size is to tailor buffer sizes according to your application's behavior. For example, when writing code to receive UDP datagrams, you should generally make the receive buffer size an even multiple of the datagram size. For overlapped I/O, setting the buffer sizes to 0 can increase performance in certain situations; when the buffers are non-zero, an extra memory copy is involved in moving data from the system buffer to the user-supplied buffer. If there is no intermediate buffer, data is immediately copied to the user-supplied buffer. The one caveat is that this is efficient only with multiple outstanding receive calls. Posting only a single receive can hurt performance because the local system cannot accept any incoming data unless you have a buffer posted and ready to receive the data. Performance considerations are covered in Chapter 6.

SO_RCVTIMEO

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

The *SO_RCVTIMEO* option sets the receive timeout value on a blocking socket. The timeout value is an integer in milliseconds that indicates how long a Winsock receive function should block when attempting to receive data. If you need to use the *SO_RCVTIMEO* option and you use the *WSASocket* function to create the socket, you must specify *WSA_FLAG_OVERLAPPED* as part of *WSASocket's dwFlags* parameter. Subsequent calls to any Winsock receive function (such as *recv*, *recvfrom*, *WSARecv*, or *WSARecvFrom*) block only for the amount of time specified. If no data arrives within that time, the call fails with the error 10060 (*WSAETIMEDOUT*). If the receiver operation does time out the socket is in an indeterminate state and should not be used.

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

SO_REUSEADDR

optval		Winsock	

Type	Get/Set	Version	Description
BOOL	Both	1+	If *TRUE*, the socket can be bound to an address already in use by another socket or to an address in the *TIME_WAIT* state.

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

The only exception is for a listening socket. Two separate sockets cannot bind to the same local interface (and port, in the case of TCP/IP) to await incoming connections. If two sockets are actively listening on the same port, the behavior is undefined as to which socket will receive notification of an incoming connection. The *SO_REUSEADDR* option is most useful in TCP when a server shuts down or exits abnormally so that the local address and port are in the *TIME_WAIT* state, which prevents any other sockets from binding to that port. By setting this option, the server can listen on the same local interface and port when it is restarted.

SO_SNDBUF

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

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

As with *SO_RCVBUF*, you can use the *SO_SNDBUF* option to set the size of the send buffer to 0. The advantage of the buffer size being 0 for blocking send calls is that when the call completes you know that your data is on the wire. Also, as in the case of a receive operation with a zero-length buffer, there is no extra memory copy of your data to system buffers. The drawback is that you lose the pipelining gained by the default stack buffering when the send buffers are nonzero in size. In other words, if you have a loop performing sends, the local network stack can copy your data to a system buffer to be sent when possible (depending on the I/O model being used). On the other hand, if your application is concerned with other logistics, disabling the send buffers can save you a few machine instructions in the memory copy. See Chapter 6 for more information.

SO_SNDTIMEO

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

The *SO_SNDTIMEO* option sets the timeout value on a blocking socket when calling a Winsock send function. The timeout value is an integer in milliseconds that indicates how long the send function should block when attempting to send data. If you need to use the *SO_SNDTIMEO* option and you use the *WSASocket* function to create the socket, you must specify *WSA_FLAG_OVERLAPPED* as part of *WSASocket's dwFlags* parameter. Subsequent calls to any Winsock send function (such as *send*, *sendto*, *WSASend*, or *WSASendTo*) block for only the amount of time specified. If the send operation cannot complete within that time, the call fails with error 10060 (*WSAETIMEDOUT*). If the send operation times out the socket is in an indeterminate state and should not be used.

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

SO_TYPE

optval Type	Get/Set	Winsock Version	Description
Int	Get only	1+	Returns the socket type (for example, *SOCK_DGRAM*, *SOCK_STREAM*) of the given socket

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

SO_UPDATE_ACCEPT_CONTEXT

optval Type	Get/Set	Winsock Version	Description
SOCKET	Both	1+	Updates a client socket with the same properties of the listening socket

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

SOL_APPLETALK Option Level

The following options are socket options specific to the AppleTalk protocol and can be used only with sockets created using *socket* or *WSASocket* with the *AF_APPLETALK* flag. A majority of the options listed here deal with either setting or obtaining AppleTalk names. For more information on the AppleTalk address family, refer back to Chapter 4. Some AppleTalk socket options—such as *SO_DEREGISTER_NAME*—have more than one option name. In such cases, all of the option's names can be used interchangeably.

SO_CONFIRM_NAME

***optval* Type**	**Get/Set**	**Winsock Version**	**Description**
WSH_NBP_TUPLE	Get only	1	Confirms that the given AppleTalk name is bound to the given address

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

SO_DEREGISTER_NAME, SO_REMOVE_NAME

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

This option is used to deregister a name from the network. If the name does not currently exist on the network, the call will return indicating success. Refer to the section entitled “Registering an AppleTalk Name” in Chapter 4 for a description of the *WSH_REGISTER_NAME* structure, which is simply another name for the *WSH_NBP_NAME* structure.

SO_LOOKUP_MYZONE, SO_GETMYZONE

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

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

SO_LOOKUP_NAME

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

This option is used to look up a specified name on the network (for example, when a client wants to connect to a server). The well-known textual name must be resolved to an AppleTalk address before a connection can be established. See the section "Resolving an AppleTalk Name" in Chapter 4 for sample code on how to look up an AppleTalk name.

One point to be aware of is that upon successful return, the *WSH_NBP_ TUPLE* structures occupy the space in the supplied buffer after the *WSH_LOOKUP_NAME* information. That is, you should supply *getsockopt* with a buffer large enough to hold the *WSH_LOOKUP_NAME* information at the start of the buffer and a number of *WSH_NBP_TUPLE* structures in the remaining space. Figure 7-1 illustrates how the buffer should be prepared prior to the call (with respect to *WSH_LOOKUP_NAME*) and where the *WSH_NBP_TUPLE* structures are placed upon return.

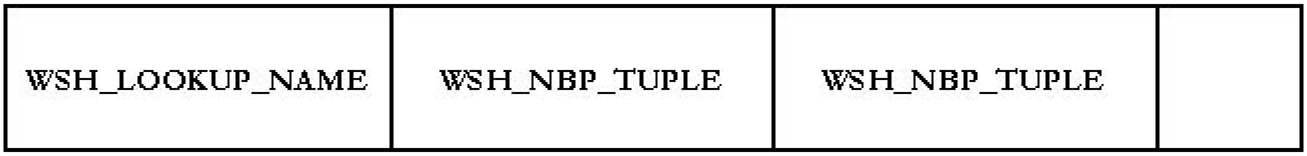


Figure 7-1 *SO_LOOKUP_NAME buffer*

SO_LOOKUP_ZONES, SO_GETZONELIST

optval Type	Get/Set	Winsock Version	Description
WSH_LOOKUP_ZONES	Get only	1	Returns zone names from the Internet zone list

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

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

atlookup = (PWSH_LOOKUP_NAME)cLookupBuffer;
zonelookup = (PWSH_LOOKUP_ZONES)cLookupBuffer;
ret = getsockopt(s, SOL_APPLETALK, SO_LOOKUP_ZONES, (char *)atlookup,
    &dwSize);
pTupleBuffer = (char *)cLookupBuffer + sizeof(WSH_LOOKUP_ZONES);
for(i = 0; i < zonelookup->NoZones; i++)
```

```
{
    printf("%3d: '%s'\n", i + 1, pTupleBuffer);
    while (*pTupleBuffer++);
}
```

SO_LOOKUP_ZONES_ON_ADAPTER, SO_GETLOCALZONES

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

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

SO_LOOKUP_NETDEF_ON_ADAPTER, SO_GETNETINFO

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

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

SO_PAP_GET_SERVER_STATUS

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

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

```
#define    MAX_PAP_STATUS_SIZE        255
#define    PAP_UNUSED_STATUS_BYTES        4

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

} WSH_PAP_GET_SERVER_STATUS, *PWSH_PAP_GET_SERVER_STATUS;
```

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

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

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

SO_PAP_PRIME_READ

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

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

SO_PAP_SET_SERVER_STATUS

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

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

SO_REGISTER_NAME

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

This option is used to register the supplied name on the AppleTalk network. If the name already exists on the network, the error *WSAEADDRINUSE* is returned. Refer to Chapter 4 for a description of the *WSH_REGISTER_NAME* structure.

SOL_IRLMP Option Level

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

IRLMP_9WIRE_MODE

optval Type	Get/Set	Winsock Version	Description
BOOL	Both	1+	Puts the IrDA socket into IrCOMM mode

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

IRLMP_ENUMDEVICES

optval Type	Get/Set	Winsock Version	Description
			Returns a list of IrDA device IDs for IR-capable devices

DEVICELIST	Get only	1+	within range

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

The *DEVICELIST* structures are different on the various platforms that support IrSock because the latest platforms that added support also added functionality. Recall that Windows CE offered IrSock support first and as a result the data structures are somewhat different. The *DEVICELIST* structure definition for Windows 98 and later versions is

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

typedef struct _WINDOWS_IRDA_DEVICE_INFO
{
    u_char  irdaDeviceID[4];
    char      irdaDeviceName[22];
    u_char  irdaDeviceHints1;
    u_char  irdaDeviceHints2;
    u_char  irdaCharSet;
} WINDOWS_IRDA_DEVICE_INFO, *PWINDOWS_IRDA_DEVICE_INFO,
  FAR *LPWINDOWS_IRDA_DEVICE_INFO;
```

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

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

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

Each of these structures contains a field, *irdaDeviceID*, which is a 4-byte identification tag used to uniquely identify that device. You need this field to fill out the *SOCKADDR_IRDA* structure used to connect to a specific device or to manipulate or obtain an IAS entry with the options *IRLMP_IAS_SET* and *IRLMP_IAS_QUERY*.

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

IRLMP_EXCLUSIVE_MODE

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

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

IRLMP_IAS_QUERY

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

This socket option is the complement of *IRLMP_IAS_SET* because it retrieves information about a class name and its service. Before making the call to *getsockopt*, you must first fill out the *irdaDeviceID* field to the device you are querying. Set the *irdaAttribName* field to the property string on which you want to retrieve its value. The most common query would be for the LSAP-SEL number; its property string is “IrDA:IrLMP:LsapSel”. Next, you need to set the *irdaClassName* field to the name of the service that the given property string applies to. Once these fields are filled, make the call to *getsockopt*. Upon success, the *irdaAttribType* field indicates which field in the union to obtain the information from. Use the identifiers in Table 7-2 to decode this entry. The most common error is *WSASERVICE_NOT_FOUND*, which is returned when the given service is not found on that device.

IRLMP_IAS_SET

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

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

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

The IAS query structure for Windows CE is

```
typedef struct _WCE_IAS_QUERY
{
    u_char     irdaDeviceID[4];
    char       irdaClassName[61];
    char       irdaAttribName[61];
    u_short    irdaAttribType;
    union
    {
        int    irdaAttribInt;
         struct
        {
            int      Len;
            u_char   OctetSeq[1];
            u_char   Reserved[3];
        } irdaAttribOctetSeq;
        struct
        {
            int      Len;
            u_char   CharSet;
            u_char   UsrStr[1];
            u_char   Reserved[2];
        } irdaAttribUsrStr;
    } irdaAttribute;
} WCE_IAS_QUERY, *PWCE_IAS_QUERY;
```

Table 7-2 provides the different constants for the *irdaAttribType* field, which indicates which type the attribute belongs to. The last two entries are not values that you can set, but values that a call to *getsockopt* with an *IRLMP_IAS_QUERY* socket option can return in the *irdaAttribType* field. These are included in the table for the sake of completeness.

Table 7-2 *IAS Attribute Types*

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

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

IRLMP_IRLPT_MODE

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

It is possible to connect to an infrared printer using Winsock and send data to be printed. This is accomplished by putting the socket in IRLPT mode before establishing the connection. Simply pass the Boolean value *TRUE* to this option after socket creation. You can use the option *IRLMP_ENUMDEVICES* to find infrared-capable printers within range. Note that some legacy IR printers do not register themselves with IAS; you might need to connect to them directly using the “LSAP-SEL-*xxx*” identifier. See Chapter 4 and its discussion of IrSock for more details on bypassing IAS. This option is available on Windows CE, and Windows 2000 and later.

IRLMP_SEND_PDU_LEN

optval Type	Get/Set	Winsock Version	Description
int	Get only	1+	Gets the maximum PDU length

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

IPPROTO_IP Option Level

The socket options on the *IPPROTO_IP* level pertain to attributes specific to the IPv4 protocol, such as modifying certain fields in the IPv4 header and adding a socket to an IPv4 multicast group. Many of these options are declared in both WINSOCK.H and WINSOCK2.H with different values. Note that if you load Winsock 1, you must include the correct header and link with Wsock32.lib. Likewise for Winsock 2, you should include the Winsock 2 header file and link with Ws2_32.lib. This is especially relevant to multicasting, which is available under both versions. Multicasting is supported on all Windows platforms except Windows CE, in which it is available on versions 2.1 and later. The new IGMPv3-related multicasting options are defined in Ws2tcpip.h.

IP_OPTIONS

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

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

- Security and handling restrictions. RFC 1108.
- Record route. Each router adds its IPv4 address to the header (see the ping sample in Chapter 11).
- Timestamp. Each router adds its IPv4 address and time.
- Loose source routing. The packet is required to visit each IPv4 address listed in the option header.
- Strict source routing. The packet is required to visit ***only*** those IPv4 addresses listed in the option header.

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

When setting an IPv4 option, the data that you pass into the *setsockopt* call follows the structure shown in Figure 7-2. The IPv4 option header can be up to 40 bytes long.

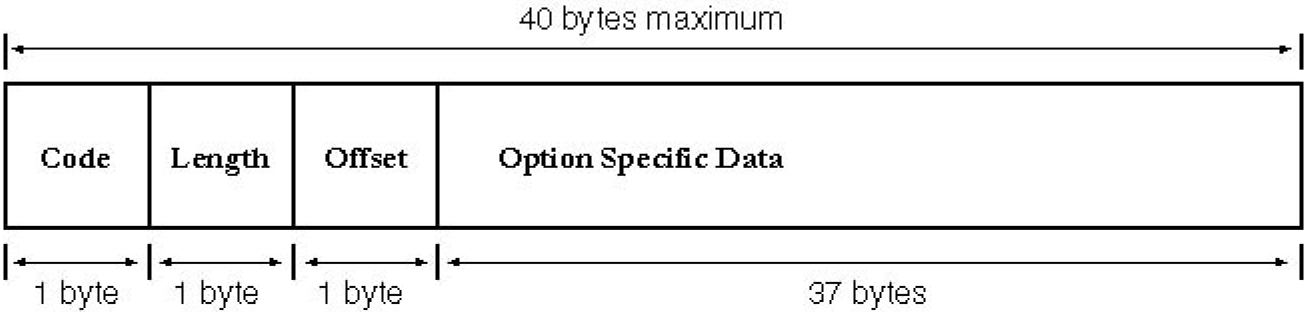


Figure 7-2 *IP option header format*

The code field indicates which type of IP option is present. For example, the value 0x7 represents the record route option. Length is simply the length of the option header, and offset is the offset value into the header where the data portion of the header begins. The data portion of the header is specific to the particular option. In the following code snippet, we set up the record route option. Notice that we declare a structure (*struct ip_option_hdr*) that contains the first three option values (code, length, offset), and then we declare the option-specific data as an array of nine unsigned long integers because the data to be recorded is up to nine IPv4 addresses. Remember that the maximum size of the IPv4 option header is 40 bytes; however, our structure occupies only 39 bytes. The system will pad the header to a multiple of a 32-bit word for you (up to 40 bytes).

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

...

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

Once the option is set, it applies to any packets sent on the given socket. At any pointer thereafter, you can call *getsockopt* with *IP_OPTIONS* to retrieve which options were set; however, this will not return any data filled into the option-specific buffers. To retrieve the data set in the IPv4 options, either the socket must be created as a raw socket (*SOCK_RAW*) or the *IP_HDRINCL* option should be set—in which case, the IPv4 header is returned along with data after a call to a Winsock receive function.

IP_HDRINCL

***optval* Type**	**Get/Set**	**Winsock Version**	**Description**
BOOL	Both	2+	If *TRUE*, IP header is submitted with data to Winsock send calls.

Setting the *IP_HDRINCL* option to *TRUE* causes the send function to include the IPv4 header ahead of the data Thus, when you call a Winsock send function, you must include the entire IPv4 header ahead of the data and fill each field of the IP header correctly. Note that the IPv4 network stack will fragment the data portion of the packet if necessary when this option is set. This option is valid only for sockets of type *SOCK_RAW*. Figure 7-3 shows what the IPv4 header should look like. This option is available only in Windows 2000 and later versions.

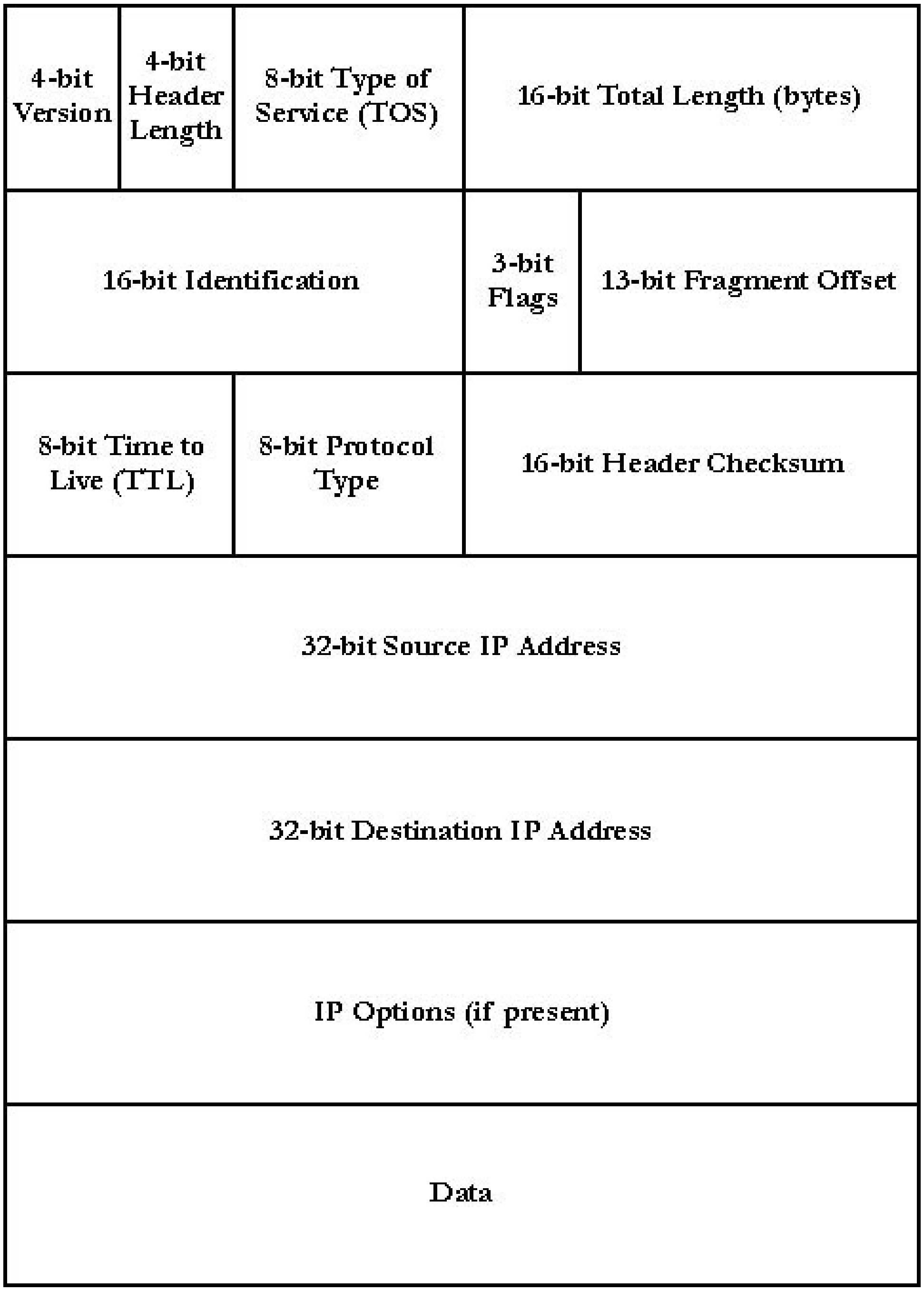


Figure 7-3 *The IPv4 header*

The first field of the header is the IPv4 version, which is 4. The header length is the number of 32-bit words in the header. An IP header must always be a multiple of 32 bits. The next field is the type of service field. Consult the *IP_TOS* socket option, discussed in the next section, for more information. The total length field is the length in bytes of the IP header and data. The identification field is a unique value used to identify each IPv4 packet sent. Normally, the system increments this value with each packet sent. The flags and fragmentation offset fields are used when IPv4 packets are fragmented into smaller packets. The TTL limits the number of routers through which the packet can pass. Each time a router forwards the packet, the TTL is decremented by 1. Once the TTL is 0, the packet is dropped. This limits the amount of time a packet can be live on the network. The protocol field is used to demultiplex incoming packets. Some of the valid protocols that use IP addressing are TCP, UDP, IGMP, and ICMP. The checksum is the 16-bit one's complement sum of the header. It is calculated over the header only and not the data. The next two fields are the 32-bit IP source and destination addresses. The IPv4 options field is a variable length field that contains optional information, usually regarding security or routing.

The easiest way to include an IPv4 header with the data you are sending is to define a structure that contains the IP header and the data and pass it into the Winsock *send* call. See Chapter 11 for more details and an example of this option. This option works only in Windows 2000 and later versions.

IP_TOS

optval Type	Get/Set	Winsock Version	Description
int	Both	1+	IPv4 type of service

The type of service (TOS) is a field present in the IPv4 header that is used to signify certain characteristics of a packet. The field is eight bits long and is broken into three parts: a 3-bit precedence field (which is ignored), a 4-bit TOS field, and the remaining bit (which must be 0). The four TOS bits are minimize delay, maximize throughput, maximize reliability, and minimize monetary costs. Only one bit can be set at a time. All four bits being 0 implies normal service. RFC 1340 specifies the recommended bits to set for various standard applications such as TCP, SMTP, and NNTP. In addition, RFC 1349 contains some corrections to the original RFC.

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

There is another issue when you attempt to set the TOS bits on a QOS-enabled socket. Because IP precedence is used by QOS to differentiate levels of service, it is undesirable to allow developers the capability to change these values. As a result, when you call *setsockopt* with *IP_TOS* on a QOS-enabled socket, the QOS service provider intercepts the call to verify whether the change can take place. See Chapter 10 for more information about QOS.

IP_TTL

optval Type	Get/Set	Winsock Version	Description
int	Both	1+	IP TTL parameter

The TTL field is present in an IP header. Datagrams use the TTL field to limit the number of routers through which the datagram can pass. The purpose of this limitation is to prevent routing loops in which datagrams can spin in circles forever. The idea behind this is that each router that the datagram passes through decrements the datagram's TTL value by 1. When the value equals 0, the datagram is discarded. This option is not available in Windows CE.

IP_MULTICAST_IF

optval Type	Get/Set	Winsock Version	Description

unsigned long	Both	1+	Gets or sets the local interface for outgoing multicast data

The IP multicast interface (IF) option sets which local interface multicast data will be sent from. This option is only of interest on machines that have more than one connected network interface (such as network card or modem). The *optval* parameter should be an unsigned long integer representing the binary IP address of the local interface. The function *inet_addr* can be used to convert a string IP dotted decimal address to an unsigned long integer, as in the following sample:

```
DWORD    mcastIF;

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

IP_MULTICAST_TTL

optval Type	Get/Set	Winsock Version	Description
int	Both	1+	Gets or sets the TTL on multicast packets for this socket

Similar to the IP TTL, this option performs the same function except that it applies only to multicast data sent using the given socket. Again, the purpose of the TTL is to prevent routing loops, but in the case of multicasting, setting the TTL narrows the scope of how far the data will travel. Therefore, multicast group members must be within “range” to receive datagrams. The default TTL value for multicast datagrams is 1.

IP_MULTICAST_LOOP

optval Type	Get/Set	Winsock Version	Description
BOOL	Both	1+	If *TRUE*, data sent to a multicast address will be echoed to the socket's incoming buffer.

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

IP_ADD_MEMBERSHIP

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

This option is the Winsock 1 method of adding a socket to an IP multicast group. This is done by creating a socket of address family *AF_INET* and the socket type *SOCK_DGRAM* with the *socket* function. To add the socket to a multicast group, use the following structure.

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

In the *ip_mreq* structure, *imr_multiaddr* is the binary address of the multicast group to join, while *imr_interface* is the local interface on which the group is joined. See Chapter 9 for more information about valid multicast addresses. The *imr_interface* field is either the binary IPv4 address of a local interface or the value *INADDR_ANY*, which can be used to select the default interface (according to the routing table).

IP_DROP_MEMBERSHIP

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

This option is the opposite of *IP_ADD_MEMBERSHIP*. By calling this option with an *ip_mreq* structure that contains the same values used when joining the given multicast group, the socket *s* will be removed from the given group. Chapter 9 contains much more detailed information on IP multicasting.

IP_ADD_SOURCE_MEMBERSHIP

***optval* Type**	**Get/Set**	**Winsock Version**	**Description**
struct ip_mreq_source	Set only	2+	Joins a multicast group but accepts data from only the given source

This multicast option joins the specified multicast group on the given interface but will accept data from only the given source IPv4 address (this is know as an include list). This option may be called multiple times to build an include list of multiple acceptable sources. The input structure is defined as

```
struct ip_mreq_source {
        struct in_addr  imr_multiaddr;
        struct in_addr  imr_sourceaddr;
        struct in_addr  imr_interface
};
```

The first field is the 32-bit multicast address, the second field is the 32-bit IPv4 address of the acceptable source, and the last field is the 32-bit IPv4 address of the local interface on which to join the group.

This option is supported on Windows XP and later versions and requires the local network to be IGMPv3 enabled. See Chapter 9 for more details.

IP_DROP_SOURCE_MEMBERSHIP

optval Type	Get/Set	Winsock Version	Description
struct ip_mreq_source	Set only	2+	Remove the given IPv4 source from the list of acceptable sources

This option is the complement to *IP_ADD_SOURCE_MEMBERSHIP*. Once one or more sources for a particular multicast group are added via *IP_ADD_SOURCE_MEMBERSHIP*, this option can be used to remove selected sources from the include list. Using these two options, an application can manage the list of sources to accept multicast data from a given multicast address. This option requires Windows XP and an IGMPv3-enabled network. See Chapter 9 for more details.

IP_BLOCK_SOURCE

optval Type	Get/Set	Winsock Version	Description
struct ip_mreq_source	Set only	2+	Joins a multicast group but accepts data from everyone except the given IPv4 source

This and the *IP_UNBLOCK_SOURCE* options are used to build an exclude list of sources for multicast traffic. The *IP_BLOCK_SOURCE* specifies a source from which multicast data will not be accepted. This option may be called multiple times to exclude additional sources. This option requires Windows XP and an IGMPv3-enabled network. See Chapter 9 for more details.

IP_UNBLOCK_SOURCE

optval Type	Get/Set	Winsock Version	Description
struct ip_mreq_source	Both	2+	Adds the given IPv4 source to the list of acceptable sources

This option removes the given source from the exclude list so that multicast data received from the removed source will be propagated to the socket. This option requires Windows XP and an IGMPv3-enabled network. See Chapter 9 for more details.

IP_DONTFRAGMENT

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

This flag tells the network not to fragment the IPv4 datagram during transmission. However, if the size of the IPv4 datagram exceeds the maximum transmission unit (MTU) and the IP don't fragment flag is set within the IPv4 header, the datagram will be dropped and an ICMP error message (“fragmentation needed but don't fragment bit set”) will be returned to the sender. This option is not available on Windows CE.

IP_PKTINFO

optval Type	Get/Set	Winsock Version	Description
int	Set only	2+	Indicates whether packet information should be returned from *WSARecvMsg*

This option indicates that IPv4 packet information should be returned as a part of the control buffer passed to the *WSARecvMsg* API. This function was discussed in Chapter 6. For IPv4, the structure returned is the *IN_PKTINFO* defined as

```
typedef struct in_pktinfo {
    IN_ADDR     ipi_addr;
    UINT        ipi_ifindex;
} IN_PKTINFO;
```

The first field is the 32-bit binary IPv4 address on which the packet was received. The second field is the interface index that can be correlated to a particular network adapter via the IP Helper APIs discussed in Chapter 16.

IPPROTO_IPV6 Option Level

The *IPPROTO_IPV6* level indicates socket options that pertain to the IPv6 protocol. Many of these options mirror the IPv4 socket options. These values are defined in WS2TCPIP.H.

IPV6_HDRINCL

optval Type	Get/Set	Winsock Version	Description
BOOL	Both	2+	If *TRUE*, IPv6 header is submitted to Winsock send calls.

This option indicates that the application will supply the IPv6 header for each packet of data sent by a Winsock send call. The IPv6 stack will not perform any fragmentation for packets larger than the MTU. Applications must build the appropriate fragmentation headers for each packet sent that is larger than the MTU. This socket option is valid only for sockets of type *SOCK_RAW*. On IPv6, raw sockets are truly raw. Figure 7-4 shows the IPv6 header.

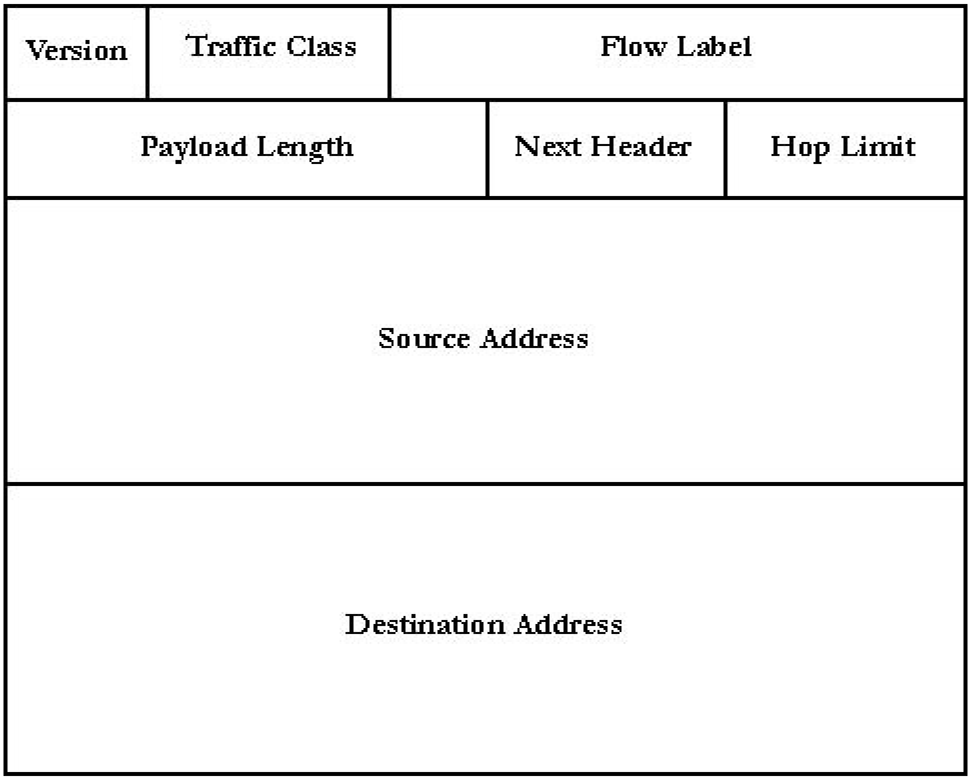


Figure 7-4 *IPv6 header*

The first field is four bits in length and is the IP version which is six. The next 8-bit field defines the traffic class. Currently, there are no well-defined values for this field. The 20-bit flow label is used to label sequences of packets that request special treatment. Again, there are no predefined values for this field. The 16-bit payload field is the length of the payload in octets. This includes any extension headers but not the basic IPv6 header itself. The 8-bit next header field indicates the protocol value of the next header. This can either be an IPv6 extension header (such as the fragmentation header) or the next encapsulated protocol (such as UDP or TCP). The 8-bit hop limit field indicates the TTL value for the packet. The last two fields are the 128-bit source and destination IPv6 addresses. See Chapter 11 for examples of using the *IPV6_HDRINCL* option.

IPV6_UNICAST_HOPS

optval **Type**	**Get/Set**	**Winsock Version**	**Description**
int	Both	2+	IPv6 TTL parameter

This option is used to set the TTL value to be used on unicast traffic sent on the socket. This is analogous to the IPv4 option *IP_TTL*.

IPV6_MULTICAST_IF

optval **Type**	**Get/Set**	**Winsock Version**	**Description**
int	Both	2+	IPv6 multicast interface for outgoing multicast data

This option is analogous to the *IP_MULTICAST_IF* option except that it sets the outgoing interface for IPv6 multicast traffic. Also, instead of specifying the local IPv6 interface address for the outgoing traffic, the interface's scope ID is specified instead. The IP Helper function *GetAdaptersAddress* can be called to obtain the local interface indices. Chapter 9 covers IPv6 multicasting in more detail and Chapter 16 discusses the IP Helper APIs.

IPV6_MULTICAST_HOPS

optval Type	Get/Set	Winsock Version	Description
int	Both	2+	IPv6 TTL parameter for multicast data

This option is analogous to the *IP_MULTICAST_TTL* option except that it sets the TTL option for IPv6 multicast traffic.

IPV6_MULTICAST_LOOP

optval Type	Get/Set	Winsock Version	Description
int	Both	2+	If *TRUE*, data sent to a multicast address will be echoed to the socket's incoming buffer.

This option enables outgoing IPv6 multicast traffic to be echoed back to the sending socket if the sending socket is also a member of the multicast group. This option is analogous to the IPv4 multicast option *IP_MULTICAST_LOOP*.

IPV6_ADD_MEMBERSHIP, IPV6_JOIN_GROUP

optval Type	Get/Set	Winsock Version	Description
struct ipv6_mreq	Both	2+	Joins an IPv6 multicast group on the specified interface

This option joins the IPv6 multicast group on a particular interface. The input parameter is defined as

```
typedef struct            ipv6_mreq {
         struct in6_addr ipv6mr_multiaddr;
         unsigned int    ipv6mr_interface;
} IPV6_MREQ;
```

The first field is the IPv6 address of the multicast address to join, and the second field is the interface index (or scope ID) on which to join the group. See Chapter 9 for more information.

IPV6_DROP_MEMBERSHIP, IPV6_LEAVE_GROUP

optval Type	Get/Set	Winsock Version	Description
struct ipv6_mreq	Both	2+	Drops the multicast group from the given interface

This option drops membership of the IPv6 multicast group from the given interface. The input parameter is the *struct ipv6_mreq* containing the multicast group to drop from the given interface (scope ID). See Chapter 9 for more information.

IPV6_PKTINFO

***optval* Type**	**Get/Set**	**Winsock Version**	**Description**
int	Both	2+	Indicates whether packet information should be returned from *WSARecvMsg*

This option indicates that IPv6 packet information should be returned as a part of the control buffer passed to the *WSARecvMsg* API. This function was discussed in Chapter 6. For IPv6, the structure returned is the *IN6_PKTINFO* defined as

```
typedef struct in6_pktinfo {
    IN6_ADDR   ipi6_addr;
    UINT       ipi6_ifindex;
} IN6_PKTINFO;
```

The first field is the binary IPv6 address on which the packet was received. The second field is the interface index that can be correlated to a particular network adapter via the IP Helper APIs discussed in Chapter 16.

IPPROTO_RM Option Level

Socket options of the *IPPROTO_RM* level are used by the reliable multicast transport. These options are declared in wsrm.h. For more information on the reliable multicast protocol available on Windows XP, see Chapter 9.

RM_RATE_WINDOW_SIZE

***optval* Type**	**Get/Set**	**Winsock Version**	**Description**
RM_SEND_WINDOW	Both	2+	Specifies the data rate and window size

This option sets the size of the send window, which determines the amount or time data remains valid for repairs. The *RM_SEND_WINDOW* structure is declared as

```
typedef struct _RM_SEND_WINDOW {
        ULONG   RateKbitsPerSec;
        ULONG   WindowSizeInMSecs;
        ULONG WindowSizeInBytes;
} RM_SEND_WINDOW
```

The first field indicates the data rate in kilobits per second. The second field is the size of the window in milliseconds. The last field indicates the window size in bytes. The values supplied for these three fields must satisfy the equation Rate-KbitsPerSec = (WindowSizeInBytes 1024) / (WindowSizeInMSecs 1000).

RM_SET_MESSAGE_BOUNDARY

optval Type	Get/Set	Winsock Version	Description
int	Both	2+	Indicates the logical size of the packet in bytes to follow

This option is used to send large messages in multiple chunks. For example, if a sender wishes to send a 2 MB buffer as a single packet, for performance reasons it is undesirable to submit a 2 MB buffer in a single call to send. Instead, this option is set to indicate the logical size of the forthcoming packet. Afterward, the sender may send the 2 MB packet in small chunks.

RM_FLUSHCACHE

optval Type	Get/Set	Winsock Version	Description
BOOL	Set only	2+	Flushes the entire contents of the send window

This option flushes the entire contents of the send window so that no more Negative Acknowledgements (NAKs) can be satisfied. This option is currently not implemented.

RM_SENDER_WINDOW_ADVANCE_METHOD

optval Type	Get/Set	Winsock Version	Description
eWINDOW_ADVANCE_METHOD	Both	2+	Indicates how the send window advances

This option specifies how the send window advances. The possible values are an enumerated type:

```
enum eWINDOW_ADVANCE_METHOD
{
    E_WINDOW_ADVANCE_BY_TIME = 1,       // Default mode
    E_WINDOW_USE_AS_DATA_CACHE
};
```

The first value indicates that the window advances with time. That is, the send data is valid for the amount of time specified in *WindowSizeInMSecs* field of the *RM_SEND_WINDOW* structure. The second value indicates that data is discarded only after the send window becomes full as specified by the *WindowSizeInBytes* field of the *RM_SEND_WINDOW* structure.

RM_SENDER_STATISTICS

optval Type	Get/Set	Winsock Version	Description

RM_SENDER_STATS	Get only	2+	Returns statistics for a sender socket

This option retrieves an *RM_SENDER_STATS* structure that contains various network statistics for the session. This structure is defined as

```
typedef struct _RM_SENDER_STATS
{
    ULONGLONG   DataBytesSent;          // # client data bytes sent
                                        // out so far
    ULONGLONG   TotalBytesSent;         // SPM, OData and RData bytes
    ULONGLONG   NaksReceived;           // # NAKs received so far
    ULONGLONG   NaksReceivedTooLate;    // # NAKs recvd after window
                                        // advanced
    ULONGLONG   NumOutstandingNaks;     // # NAKs yet to be
                                        // responded to
    ULONGLONG   NumNaksAfterRData;      // # NAKs yet to be
                                        // responded to
    ULONGLONG   RepairPacketsSent;      // # Repairs (RDATA)
                                        // sent so far
    ULONGLONG   BufferSpaceAvailable;   // # partial messages dropped
    ULONGLONG   TrailingEdgeSeqId;      // smallest (oldest) Sequence
                                        // Id in the window
    ULONGLONG   LeadingEdgeSeqId;       // largest (newest) Sequence
                                        // Id in the window
    ULONGLONG   RateKBitsPerSecOverall; // Internally calculated
                                        // send-rate from the
                                        // beginning
    ULONGLONG  RateKBitsPerSecLast;      // Send-rate calculated every
                                        // INTERNAL_RATE_CALCULATION_
                                        // FREQUENCY
} RM_SENDER_STATS;
```

The comments for each field from the header file explain the structure so we won't repeat them.

RM_LATEJOIN

optval Type	Get/Set	Winsock Version	Description
int	Both	2+	Allows receivers to NAK for any packet in the window

When this option is set, a receiver may NAK any sequence number in the advertise window upon session join.

RM_SET_SEND_IF

optval Type	Get/Set	Winsock Version	Description
ULONG	Both	2+	Sets the outgoing interface for reliable multicast traffic

This option sets the outgoing interface for reliable multicast traffic. The 32-bit binary address of the local outgoing interface is supplied as the input parameter.

RM_ADD_RECEIVE_IF

optval Type	Get/Set	Winsock Version	Description
ULONG	Set only	2+	Adds the given interface as a receiving interface

By default, when a reliable multicast receiver is created, it listens for traffic on the default interface (as indicated by the routing table). This option can be used to specify the 32-bit binary IPv4 address of the local interface to listen for traffic. This option can be specified multiple times to add more than one listening interface in the event of a multihomed machine.

RM_DEL_RECEIVE_IF

optval Type	Get/Set	Winsock Version	Description
ULONG	Set only	2+	Removes the given interface from the list of receiving interfaces

This option removes the supplied interface from the list of valid receiving interfaces for reliable multicast traffic. The address is supplied as a 32-bit binary IPv4 address.

RM_SEND_WINDOW_ADV_RATE

optval Type	Get/Set	Winsock Version	Description
int	Both	2+	Sets the increment percentage for the send window

The send window advances incrementally over time so that a portion of the oldest data is discarded to make room for new data sent by the application. This option controls how much data is discarded with each incremental advance. The maximum possible increment value as a percentage of the whole window is defined as *MAX_WINDOW_INCREMENT_PERCENTAGE*.

RM_USE_FEC

optval Type	Get/Set	Winsock Version	Description
RM_FEC_INFO	Both	2+	Enables forward error correction on the session

This option enables the use of forward error correction (FEC) on the session. This *RM_FEC_INFO* structure is defined as

```
typedef struct _RM_FEC_INFO
{
    USHORT              FECBlockSize;
    USHORT              FECProActivePackets;
    UCHAR               FECGroupSize;
    BOOLEAN             fFECOnDemandParityEnabled;
} RM_FEC_INFO;
```

The *FECGroupSize* indicates the number of original packets to build parity packets from and the *FECBlockSize* indicates the number of original packets plus the number of parity packets generated. *FECProActivePackets* indicates to transmit FEC packets always (instead of only for repair data). The *fFECOnDemandParityEnabled* allows receivers to request FEC repair data on demand.

RM_SET_MCAST_TTL

optval **Type**	**Get/Set**	**Winsock Version**	**Description**
int	Both	2+	Sets the TTL for reliable multicast data sent

This option sets the TTL value for a reliable multicast sender.

RM_RECEIVER_STATISTICS

optval **Type**	**Get/Set**	**Winsock Version**	**Description**
RM_RECEIVER_STATS	Both	2+	Retrieves session statistics on a receiver socket

This option retrieves an *RM_SENDER_STATS* structure, which contains various network statistics for the session. This structure is defined as

```
typedef struct _RM_RECEIVER_STATS
{
    ULONGLONG   NumODataPacketsReceived;// # OData sequences received
    ULONGLONG   NumRDataPacketsReceived;// # RData sequences received
    ULONGLONG   NumDuplicateDataPackets;// # RData sequences received
    ULONGLONG   DataBytesReceived;      // # client data bytes
                                        // received out so far
    ULONGLONG   TotalBytesReceived;     // SPM, OData and RData bytes
    ULONGLONG   RateKBitsPerSecOverall; // Internally calculated
                                        // Receive-rate from the
                                        // beginning
    ULONGLONG   RateKBitsPerSecLast;    // Receive-rate calculated
                                        // every INTERNAL_RATE_
                                        // CALCULATION_FREQUENCY
    ULONGLONG   TrailingEdgeSeqId;      // smallest (oldest)
                                        // Sequence Id in the window
    ULONGLONG   LeadingEdgeSeqId;       // largest (newest) Sequence
                                        // Id in the window
    ULONGLONG   AverageSequencesInWindow;
    ULONGLONG   MinSequencesInWindow;
    ULONGLONG   MaxSequencesInWindow;

    ULONGLONG   FirstNakSequenceNumber; // # First Outstanding NAK
    ULONGLONG   NumPendingNaks;         // # Sequences waiting
                                        // for NCFs
    ULONGLONG   NumOutstandingNaks;     // # Sequences for which
                                        // NCFs have been received,
                                        // but no RDATA
    ULONGLONG   NumDataPacketsBuffered; // # Data packets currently
```

```
                                        // buffered by transport
    ULONGLONG   TotalSelectiveNaksSent; // # Selective NAKs sent
                                        // so far
    ULONGLONG   TotalParityNaksSent;    // # Parity NAKs sent so far
} RM_RECEIVER_STATS;
```

The comments for each field from the header file explain the structure so we won't repeat them.

IPPROTO_TCP Option Level

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

TCP_NODELAY

optval Type	Get/Set	Winsock Version	Description
BOOL	Both	1+	If *TRUE*, the Nagle algorithm is disabled on the socket.

To increase performance and throughput by minimizing overhead, the system implements the Nagle algorithm. When an application requests to send a chunk of data, the system might hold on to that data for a while and wait for other data to accumulate before actually sending it on the wire. Of course, if no other data accumulates in a given period of time, the data will be sent regardless. This results in more data in a single TCP packet, as opposed to smaller chunks of data in multiple TCP packets. The overhead is that the TCP header for each packet is 20 bytes long. Sending a couple bytes here and there with a 20-byte header is wasteful. The other part of this algorithm is the delayed acknowledgments. Once a system receives TCP data it must send an ACK to the peer. However, the host will wait to see if it has data to send to the peer so that it can piggyback the ACK on the data to be sent—resulting in one less packet on the network.

The purpose of this option is to disable the Nagle algorithm because its behavior can be detrimental in a few cases. This algorithm can adversely affect any network application that sends relatively small amounts of data and expects a timely response. A classic example is Telnet. Telnet is an interactive application that allows the user to log on to a remote machine and send it commands. Typically, the user hits only a few keystrokes per second. The Nagle algorithm would make such a session seem sluggish and unresponsive.

NSPROTO_IPX Option Level

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

IPX_PTYPE

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

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

IPX_FILTERPTYPE

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

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

IPX_STOPFILTERPTYPE

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

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

IPX_DSTYPE

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

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

IPX_EXTENDED_ADDRESS

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

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

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

IPX_RECVHDR

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

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

IPX_MAXSIZE

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

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

IPX_ADDRESS

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

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

```
typedef struct _IPX_ADDRESS_DATA
{
    INT     adapternum;  // Input: 0-based adapter number
    UCHAR   netnum[4];   // Output: IPX network number
    UCHAR   nodenum[6];  // Output: IPX node address
    BOOLEAN wan;         // Output: TRUE = adapter is on a WAN link
    BOOLEAN status;      // Output: TRUE = WAN link is up (or adapter
                         // is not WAN)
    INT     maxpkt;      // Output: max packet size, not including IPX
                         // header
    ULONG   linkspeed;   // Output: link speed in 100 bytes/sec
                         // (i.e., 96 == 9600 bps)
} IPX_ADDRESS_DATA, *PIPX_ADDRESS_DATA;
```

IPX_GETNETINFO

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

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

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

IPX_GETNETINFO_NORIP

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

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

IPX_SPXGETCONNECTIONSTATUS

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

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

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

IPX_ADDRESS_NOTIFY

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

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

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

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

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

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

IPX_MAX_ADAPTER_NUM

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

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

IPX_RERIPNETNUMBER

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

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

IPX_RECEIVE_BROADCAST

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

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

IPX_IMMEDIATESPXACK

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

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

Ioctlsocket, *WSAIoctl*, and *WSANSPIoctl*

The socket *ioctl* functions are used to control the behavior of I/O on the socket, as well as to obtain information about I/O pending on that socket. The first function, *ioctlsocket*, originated in the Winsock 1 specification and is declared as

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

The parameter *s* is the socket descriptor to act on, and *cmd* is a predefined flag for the I/O control command to execute. The last parameter, *argp*, is a pointer to a variable specific to the given command. When each command is described, the type of the required variable is given. Winsock 2 introduced a new *ioctl* function that adds quite a few new options. First, it breaks the single *argp* parameter into a set of input parameters for values passed into the function and a set of output parameters used to return data from the call. In addition, the function call can use overlapped I/O. This function is *WSAIoctl*, which is defined as

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

The first two parameters are the same as those in *ioctlsocket*. The second two, *lpvInBuffer* and *cbInBuffer*, describe the input parameters. The *lpvInBuffer* parameter is a pointer to the value passed in, and *cbInBuffer* is the size of that data in bytes. Likewise, *lpvOutBuffer* and *cbOutBuffer* are used for any data returned from the call. The *lpvOutBuffer* parameter points to a data buffer in which any information returned is placed. The *cbOutBuffer* parameter is the size in bytes of the buffer passed in as *lpvOutBuffer*. Note that some calls might use only input or output parameters, and others will use both. The seventh parameter, *lpcbBytesReturned*, is the number of bytes actually returned. The last two parameters, *lpOverlapped* and *lpCompletionRoutine*, are used when calling this function with overlapped I/O. Consult Chapter 5 for detailed information on using overlapped I/O.

Lastly, a new *ioctl* function, *WSANSPIoctl*, has been introduced for Windows XP. This function is used exclusively for making I/O control calls to name space providers. The function is defined as

```
int WSANSPIoctl(
                HANDLE hLookup,
                DWORD   dwControlCode,
                LPVOID lpvInBuffer,
                DWORD   cbInBuffer,
                LPVOID lpvOutBuffer,
                DWORD   cbOutBuffer,
                LPDWORD lpcbBytesReturned,
                LPWSACOMPLETION lpCompletion
        );
```

The parameters to this function are the same as *WSAIoctl* except for *hLookup* and *lpCompletion*. The handle passed into this function is the handle returned from *WSALookupServiceBegin*. This handle identifies a name space query to a particular name space provider. The *lpCompletion* parameter specifies how the application should be notified if there is a change to the name space provider or query. This new *ioctl* function is used by the Network Location Awareness (NLA) service, which is described in detail in Chapter 8, so we won't explain it in full here.

Standard Ioctl Commands

There are three ioctl commands that are the most common and are carryovers from the Unix world. They are available on all Windows platforms. Also, these three commands can be called using either *ioctlsocket* or *WSAIoctl*.

FIONBIO

Which Function?	Input	Output	Winsock Version	Description
ioctlsocket/WSAIoctl	*unsigned int*	None	1+	Puts socket in non-blocking mode

This command enables or disables non-blocking mode on socket *s*. By default, all sockets are blocking sockets upon creation. When you call *ioctlsocket* with the FIONBIO ioctl command, set *argp* to pass a pointer to an unsigned long integer whose value is nonzero if non-blocking mode is to be enabled. The value 0 places the socket in blocking mode. If you use *WSAIoctl* instead, simply pass the unsigned long integer in as the *lpvInBuffer* parameter.

Calling the *WSAAsyncSelect* or *WSAEventSelect* function automatically sets a socket to non-blocking mode. If either of these functions has been called, any attempt to set the socket back to blocking mode fails with *WSAEINVAL*. To set the socket back to blocking mode, an application must first disable *WSAAsyncSelect* by calling *WSAAsyncSelect* with the *lEvent* parameter equal to 0 or disable *WSAEventSelect* by calling *WSAEventSelect* with the *lNetworkEvents* parameter equal to 0.

FIONREAD

Which Function?	Input	Output	Winsock Version	Description
Both	None	*unsigned long*	1+	Returns the amount of data to be read on the socket

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

SIOCATMARK

Which Function?	Input	Output	Winsock Version	Description
Both	None	*BOOL*	1+	Determines whether OOB data has been read

When a socket has been configured to receive OOB data and has been set to receive this data inline (by setting the *SO_OOBINLINE* socket option), this ioctl command returns a Boolean value indicating *TRUE* if the OOB data is to be read next. Otherwise, *FALSE* is returned and the next receive operation returns all or some of the data that precedes the OOB data. For *ioctlsocket*, *argp* returns with a pointer to a Boolean variable, while for *WSAIoctl*, the pointer to the Boolean variable returns in *lpvOutBuffer*. Remember that a receive call will never mix OOB data and normal data in the same call. Refer back to Chapter 1 for more information on OOB data.

Other Ioctl Commands

These ioctl commands are specific to Winsock 2 except for those dealing with Secure Sockets Layer (SSL), which are available only on Windows CE. If you examine the Winsock 2 headers, you might actually see other ioctl commands declared; however, the ioctls listed in this section are the only ones that are meaningful or available to a user's application. In addition, as you will see, not all ioctl commands work on all (or any) Windows platform, but of course this could change with operating system updates. For Winsock 2, a majority of these commands are defined in WINSOCK2.H. Some of the newer ioctls are defined in MSTCPIP.H.

SIO_ENABLE_CIRCULAR_QUEUEING

Which Function?	Input	Output	Winsock Version	Description
WSAIoctl	*BOOL*	*BOOL*	2+	If the incoming buffer queue overflows, discard oldest message first.

This ioctl command controls how the underlying service provider handles incoming datagram messages when the queues are full. By default, when the incoming queue is full, any datagram messages subsequently received are dropped. When this option is set to *TRUE*, it indicates that the newly arrived messages should never be dropped as a result of buffer overflow; instead, the oldest message in the queue should be discarded to make room for the newly arrived message. This command is valid only for sockets associated with unreliable, message-oriented protocols. If this ioctl command is used on a socket of another type (such as a stream-oriented protocol socket), or if the service provider doesn't support the command, the error *WSAENOPROTOOPT* is returned. This option is supported only on Windows NT.

This ioctl command can be used either to set circular queuing on or off or to query the current state of the option. When you are setting the option, only the input parameters need to be used. When you are querying the current value of the option, only the output *BOOL* parameter needs to be supplied.

SIO_FIND_ROUTE

Which Function?	Input	Output	Winsock Version	Description
WSAIoctl	*SOCKADDR*	*BOOL*	2+	Verifies that a route to the given address exists

This ioctl command is used to check whether a particular address can be reached via the network. The *lpvInBuffer* parameter points to a *SOCKADDR* structure for the given protocol. If the address already exists in the local cache, it is invalidated. For IPX, this call initiates an IPX *GetLocalTarget* call that queries the network for the given remote address. Unfortunately, the Microsoft provider for current Windows platforms does not implement this ioctl command.

SIO_FLUSH

Which Function?	Input	Output	Winsock Version	Description
WSAIoctl	None	None	2+	Discards the send buffers

This ioctl command discards the current contents of the sending queue associated with the given socket. There are no input or output parameters for this option. Currently, this option is supported on Windows NT 4 Service Pack 4, Windows 2000, and Windows XP.

SIO_GET_BROADCAST_ADDRESS

Which Function?	Input	Output	Winsock Version	Description
WSAIoctl	None	*SOCKADDR*	2+	Returns a broadcast address for the address family of the socket

This ioctl command returns a *SOCKADDR* structure (via *lpvOutBuffer*) that contains the broadcast address for the address family of socket *s* that can be used in *sendto* or *WSASendTo*. This ioctl works only on Windows NT. Windows 95, Windows 98, and Windows Me return *WSAEINVAL*.

SIO_GET_EXTENSION_FUNCTION_POINTER

Which Function?	Input	Output	Winsock Version	Description
WSAIoctl	GUID	Function pointer	2+	Retrieves a function pointer specific to the underlying provider

This ioctl command is used to obtain functions that are provider-specific but are not part of the Winsock specification. If a provider chooses, it can make functions available to programmers through this ioctl command by assigning each function a GUID. Then an application can obtain a pointer to this function by using the *SIO_GET_EXTENSION_FUNCTION_POINTER* ioctl. The header file Mswsock.h defines those Winsock functions that Microsoft has added, including their globally unique identifiers (GUIDs). For example, to query whether the installed Winsock provider supports the *TransmitFile* function, you can query the provider by using its GUID, which is given by the following define:

```
#define WSAID_TRANSMITFILE \
  {0xb5367df0,0xcbac,0x11cf,{0x95,0xca,0x00,0x80,0x5f,0x48,0xa1,0x92}}
```

Once you obtain the function pointer for an extension function, such as *TransmitFile*, you can call it directly without having to link your application to the Mswsock.lib library. This will actually reduce one intermediate function call that is made in Mswsock.lib.

You can look through Mswsock.h for other Microsoft-specific extensions that have these GUIDs defined for them. See Chapter 6 for more information on the Microsoft-specific extensions. Also, this ioctl command is an important part of developing a layered service provider. See Chapter 12 for more details about the service provider interface.

SIO_CHK_QOS

Which Function?	Input	Output	Winsock Version	Description
WSAIoctl	*DWORD*	*DWORD*	2+	Checks QOS attributes for the given socket

This ioctl command can be used to check the status of six states within QOS and is currently supported only on Windows 2000 and later versions. Six flags correspond to these states: *ALLOWED_TO_SEND_DATA*, *ABLE_TO_RECV_ RSVP*, *LINE_RATE*, *LOCAL_TRAFFIC_CONTROL*, *LOCAL_QOSABILITY*, and *END_ TO_END_QOSABILITY*.

The first flag, *ALLOWED_TO_SEND_DATA*, is used once QOS levels are set on a socket using *SIO_SET_QOS* but before any RSVP reservation request (RESV) message has been received. Receiving a RESV message indicates that the desired bandwidth requirements have been allocated to your flow. Prior to receiving the RESV message, the flow corresponding to the socket is given only best-effort service. The RSVP protocol and reservation of network resources are covered in greater detail in Chapter 10. Use the *ALLOWED_TO_SEND_DATA* flag to see if the current best-effort service is sufficient for the levels of QOS requested by *SIO_SET_QOS*. The return value will be either 1—meaning that the current best-effort bandwidth is sufficient—or 0, meaning that the bandwidth cannot accommodate the requested levels. If the *ALLOWED_TO_SEND_DATA* flag returns 0, the sending application should wait until a RESV message is received before sending data.

The second flag, *ABLE_TO_RECV_RSVP*, indicates whether the host is able to receive and process RSVP messages on the interface that the given socket is bound to. The return value is either 1 or 0, corresponding to whether RSVP messages can or cannot be received, respectively.

The next flag, *LINE_RATE*, returns the best-effort line rate in kilobits per second (kbps). If the line rate is not known, the value *INFO_NOT_AVAILABLE* is returned.

The last three flags indicate whether certain capabilities exist on the local machine or the network. All three options return 1 to indicate the option is supported, 0 if it is not supported, or *INFO_NOT_AVAILABLE* if there is no way to check. *LOCAL_TRAFFIC_CONTROL* is used to determine if the Traffic Control component is installed and available on the machine. *LOCAL_QOSABILITY* determines whether QOS is supported on the local machine. Finally, *END_TO_END_QOSABILITY* indicates whether the local network is QOS-enabled.

SIO_GET_QOS

Which Function?	Input	Output	Winsock Version	Description
WSAIoctl	None	*QOS*	2+	Returns the *QOS* structure associated with the socket

This ioctl command retrieves the *QOS* structure associated with the socket. The supplied buffer must be large enough to contain the whole structure, which in some cases is larger than *sizeof(QOS)* because the structure might contain provider-specific information. For more information on QOS, see Chapter 10. If this ioctl command is used on a socket whose address family does not support QOS, the error *WSAENOPROTOOPT* is returned. This option and *SIO_SET_QOS* are available only on platforms that provide a QOS-capable transport, such as Windows 98, Windows Me, and Windows 2000 and later versions.

SIO_SET_QOS

Which Function?	Input	Output	Winsock Version	Description
WSAIoctl	*QOS*	None	2+	Sets the QOS attributes for the given socket

This ioctl command is the companion to *SIO_GET_QOS*. The input parameter for this call is a *QOS* structure that defines the bandwidth requirements for this socket. This call does not return any output values. See Chapter 10 for more information about QOS. This option and *SIO_GET_QOS* are available only on those platforms that provide a QOS-capable transport, such as Windows 98, Windows Me, and Windows 2000 and later versions.

SIO_MULTIPOINT_LOOPBACK

Which Function?	Input	Output	Winsock Version	Description
WSAIoctl	*BOOL*	*BOOL*	2+	Sets or gets whether multicast data will be looped back to the socket

When sending multicast data, the default behavior is to have any data sent to the multicast group posted as incoming data on the socket's receive queue. Of course, this loopback is in effect only if the socket is also a member of the multicast group that it is sending to. Currently, this loopback behavior is seen only in IP multicasting and is not present in ATM multicasting. To disable this loopback, pass a Boolean variable with the value *FALSE* into the input parameter *lpvInBuffer*. To obtain the current value of this option, leave the input value as *NULL* and supply a Boolean variable as the output parameter.

SIO_MULTICAST_SCOPE

Which Function?	Input	Output	Winsock Version	Description
WSAIoctl	*int*	*int*	2+	Gets or sets the TTL value for multicast data

This ioctl command controls the lifetime, or scope, of multicast data. The scope is the number of routed network segments that data is allowed to traverse; by default, the value is only 1. When a multicast packet hits a router, the TTL value is decremented by 1. Once the TTL reaches 0, the packet is discarded. To set the value, pass an integer with the desired TTL as *lpvInBuffer*; otherwise, to get the current TTL value, call *WSAIoctl* with the *lpvOutBuffer* pointing to an integer.

SIO_KEEPALIVE_VALS

Which Function?	Input	Output	Winsock Version	Description
WSAIoctl	*tcp_keepalive*	*tcp_keepalive*	2+	Sets the TCP keepalive active on a per-connection basis

This ioctl command allows setting the TCP keepalive active on a per-connection basis and allows you to specify the keepalive interval. The socket option *SO_KEEPALIVE* also enables TCP keepalives, but the interval on which they are sent is set in the Registry. Changing the Registry value will change the keepalive interval for all processes on the machine. This ioctl command allows you to set the interval on a per-socket basis. To set the keepalive active on the given connected socket, initialize a *tcp_keepalive* structure and pass it as the input buffer. The structure is defined as

```
 struct tcp_keepalive
{
    u_long    onoff;
    u_long    keepalivetime;
    u_long    keepaliveinterval;
}
```

The meaning of the structure fields *keepalivetime* and *keepaliveinterval* are identical to the Registry values discussed in the *SO_KEEPALIVE* option presented earlier in this chapter. Once a keepalive is set, you can query for the current keepalive values by calling *WSAIoctl* with the *SIO_KEEPALIVE_VALS* ioctl command and supplying a *tcp_keepalive* structure as the output buffer. This ioctl command is available on Windows 2000 and later versions.

SIO_RCVALL

Which Function?	Input	Output	Winsock Version	Description
WSAIoctl	*unsigned int*	None	2+	Receives all packets on the network

Using this ioctl command with the value *TRUE* allows the given socket to receive all IPv4 packets on the network. This option is currently not implemented for IPv6 sockets, so the socket handle that must be passed to *WSAIoctl* must be of address family *AF_INET*, socket type *SOCK_RAW*, and protocol *IPPROTO_IP*. In addition, the socket must be bound to an explicit local interface. That is, you cannot bind to *INADDR_ANY*. Once the socket is bound and the ioctl is set, calls to *recv*/*WSARecv* return IPv4 datagrams. Keep in mind that these are datagrams—you must supply a sufficiently large buffer. Because the total length field of the IPv4 header is a 16-bit quantity, the maximum theoretical limit is 65,535 bytes; however, in practice the maximum transmission unit (MTU) of networks is much lower. Using this ioctl command requires Administrator privileges on the local machine. In addition, this ioctl command is available in Windows 2000 and later versions.

> A sample application on the companion CD, RCVALL.C, illustrates using this and the other two *SIO_RCVALL* ioctl commands.

SIO_RCVALL_MCAST

Which Function?	Input	Output	Winsock Version	Description
WSAIoctl	*unsigned int*	None	2+	Receives all multicast packets on the network

This ioctl command is similar to the *SIO_RCVALL* command just described. The same usage rules mentioned for *SIO_RCVALL* also apply to this command except that the socket passed to *WSAIoctl* should be created with the protocol equal to *IPPROTO_UDP*. The one difference is that only multicast IPv4 traffic is returned, as opposed to all IP packets. This means that only IPv4 packets destined for addresses in the range 224.0.0.0 through 239.255.255.255 are returned. This ioctl command is available on Windows 2000 and later versions.

SIO_RCVALL_IGMPMCAST

Which Function?	Input	Output	Winsock Version	Description
WSAIoctl	*unsigned int*	None	2+	Receives all IGMP packets on the network

Again, this ioctl command is the same as *SIO_RCVALL*, except that the socket passed into *WSAIoctl* should be created with the protocol equal to *IPPROTO_IGMP*. Setting this option returns only IGMP packets. See the *SIO_RCVALL* entry for instructions about how to use this option. This ioctl is available in Windows 2000 and later versions.

SIO_ROUTING_INTERFACE_QUERY

Which Function?	Input	Output	Winsock Version	Description
Both	*SOCKADDR*	None	2+	Returns the local interface used to reach the supplied destination

This ioctl command allows you to find the address of the local interface that should be used to send data to a remote machine. The remote machine's address should be supplied in the form of a *SOCKADDR* structure as the *lpvInBuffer* parameter. In addition, a sufficiently large buffer needs to be supplied as the *lpvOutBuffer*, which will contain an array of one or more *SOCKADDR* structures describing the local interface(s) that can be used. This command can be used for either unicast or multicast endpoints, and the interface returned from this call can be used in a subsequent call to *bind*.

The Windows 2000 and Windows XP plug-and-play capabilities are the motivation for having an ioctl like this. The user can insert or remove a PCMCIA network card, affecting which interfaces an application can use. A well-written application in Windows 2000 should take this into account.

Therefore, applications cannot rely on the information returned by *SIO_ROUTING_INTERFACE_QUERY* to be persistent. To handle this situation, you should also use the *SIO_ROUTING_INTERFACE_CHANGE* ioctl command, which notifies your application when the interfaces change. Once this occurs, call *SIO_ROUTING_INTERFACE_QUERY* once again to obtain the latest information.

See the code sample in the directory SIO_ROUTING_INTERFACE on the companion C for an example on how to use these ioctls.

SIO_ROUTING_INTERFACE_CHANGE

Which Function?	Input	Output	Winsock Version	Description
WSAIoctl	*SOCKADDR*	None	2+	Sends notification when an interface to an endpoint has changed

Using this ioctl command indicates that you want to be notified of any change in the local routing interface that is used to access the specified remote address. When you use this command, a *SOCKADDR* structure to the remote address in question is submitted in the input buffer and no data is returned upon successful completion. However, if the interface to that route changes in some way, the application will be notified, at which point the application can call *SIO_ROUTING_INTERFACE_QUERY* to determine which interface to use as a result.

There are several ways to make a call to this command. If the socket is blocking, the *WSAIoctl* call will not complete until some point at which the interface changes. If the socket is in non-blocking mode, the error *WSAEWOULDBLOCK* is returned. Then the application can wait for routing-change events through either *WSAEventSelect* or *WSAAsyncSelect*, with the *FD_ROUTING_INTERFACE_CHANGE* flag set in the network event bitmask. Overlapped I/O can also be used to make the call. With this method, you supply an event handle in the *WSAOVERLAPPED* structure, which is signaled upon a routing change.

The address specified in the input *SOCKADDR* structure can be a specific address, or you can use the wildcard *INADDR_ANY*, indicating that you want to be notified of any routing changes. Note that because routing information remains fairly static, providers have the option of ignoring the information that the application supplied in the input buffer and simply sending a notification upon any interface change. As a result, it is probably a good idea to register for notification on any change and simply call *SIO_ROUTING_INTERFACE_QUERY* to see whether the change affects your application.

SIO_ADDRESS_LIST_QUERY

Which Function?	Input	Output	Winsock Version	Description
WSAIoctl	None	*SOCKET_ADDRESS_LIST*	2+	Returns a list of interfaces that the socket can bind to

This ioctl is used to obtain a list of local transport addresses matching the socket's protocol family that the application can bind to. The output buffer is a *SOCKET_ADDRESS_LIST* structure, defined as

```
typedef struct _SOCKET_ADDRESS_LIST
{
    INT            iAddressCount;
    SOCKET_ADDRESS Address[1];
} SOCKET_ADDRESS_LIST, FAR * LPSOCKET_ADDRESS_LIST;

typedef struct _SOCKET_ADDRESS
{
    LPSOCKADDR lpSockaddr;
    INT          iSockaddrLength;
} SOCKET_ADDRESS, *PSOCKET_ADDRESS, FAR * LPSOCKET_ADDRESS;
```

The *iAddressCount* field returns the number of address structures in the list, and the *Address* field is an array of protocol family–specific addresses.

In Windows plug-and-play environments, the number of valid addresses can change dynamically; therefore, applications cannot rely on the information this ioctl command returns to remain constant. To take this into account, applications should first call *SIO_ADDRESS_LIST_QUERY* to obtain current interface information and then call *SIO_ADDRESS_LIST_CHANGE* to receive notification of future changes. If the address list changes, the application should again make a query.

If the supplied output buffer is not of sufficient size, *WSAIoctl* fails with *WSAEFAULT*, and the *lcbBytesReturned* parameter indicates the required buffer size. This ioctl is supported in Windows 2000 or later versions.

See the code sample in the directory SIO_ADDRESS_LIST on the companion CD for ar example of how to use these ioctls.

SIO_ADDRESS_LIST_SORT

Which Function?	Input	Output	Winsock Version	Description
WSAIoctl	*SOCKET_ ADDRESS_LIST*	*SOCKET_ ADDRESS_LIST*	2+	Sorts the address list by preference

This option takes the *SOCKET_ADDRESS_LIST* structure returned from an *SIO_ADDRESS_LIST_QUERY* command and sorts the addresses as well as fills in the appropriate scope IDs in the case of IPv6 addresses. Note that the scope IDs are only set for site-local addresses when there is a global address in the list, and the global address falls within one of the global site-prefixes. This iocl is available in Windows XP and later versions.

SIO_ADDRESS_LIST_CHANGE

Which Function?	Input	Output	Winsock Version	Description
WSAIoctl	None	None	2+	Notifies when local interfaces change

An application can use this command to receive notification of changes in the list of local transport addresses of the given socket's protocol family that the application can bind to. No information is returned in the output parameters upon successful completion of the call.

There are several ways to make a call to this command. If the socket is blocking, the *WSAIoctl* call will not complete until some point at which the interface changes. If the socket is in non-blocking mode, the error *WSAEWOULDBLOCK* is returned. Then the application can wait for routing-change events through either *WSAEventSelect* or *WSAAsyncSelect* with the *FD_ADDRESS_LIST_CHANGE* flag set in the network event bitmask. In addition, overlapped I/O can be used to make the call. With this method, you supply an event handle in the *WSAOVERLAPPED* structure, which is signaled on a routing change. This ioctl is supported in Windows 2000 and later versions.

SIO_GET_INTERFACE_LIST

Which Function?	Input	Output	Winsock Version	Description
WSAIoctl	None	*INTERFACE_INFO []*	2+	Returns a list of local interfaces

This ioctl is defined in Ws2tcpip.h. It is used to return information about each interface on the local machine. Nothing is required on input, but on output, an array of *INTERFACE_INFO* structures is returned. These structures are defined as

```
typedef struct _INTERFACE_INFO
{
    u_long          iiFlags;             /* Interface flags   */
    sockaddr_gen    iiAddress;           /* Interface address */
    sockaddr_gen    iiBroadcastAddress; /* Broadcast address */
    sockaddr_gen    iiNetmask;           /* Network mask      */
} INTERFACE_INFO, FAR * LPINTERFACE_INFO;

#define IFF_UP           0x00000001 /* Interface is up             */
#define IFF_BROADCAST    0x00000002 /* Broadcast is supported      */
#define IFF_LOOPBACK     0x00000004 /* This is loopback interface   */
#define IFF_POINTTOPOINT 0x00000008 /* This is point-to-point interface*/
#define IFF_MULTICAST    0x00000010 /* Multicast is supported      */

typedef union sockaddr_gen
{
    struct sockaddr Address;
    struct sockaddr_in  AddressIn;
    struct sockaddr_in6 AddressIn6;
} sockaddr_gen;
```

The *iiFlags* member returns a bitmask of flags indicating whether the interface is up (*IFF_UP*) as well as whether broadcast (*IFF_BROADCAST*) or multicast (*IFF_MULTICAST*) is supported. It also indicates whether the interface is loopback (*IFF_LOOPBACK*) or point-to-point (*IFF_POINTTOPOINT*). The other three fields contain the address of the interface, the broadcast address, and the corresponding netmask.

SIO_GET_INTERFACE_LIST_EX

Which Function?	Input	Output	Winsock Version	Description

WSAIoctl	None	*INTERFACE_INFO_EX []*	2+	Returns a list of local interfaces

This ioctl is the same as *SIO_GET_INTERFACE_LIST* except the structure returned contains embedded *SOCKET_ADDRESS* structure to describe each local interface, as opposed to *SOCKADDR_GEN* structure. This removes the dependency the size of the socket address structure. The *INTERFACE_INFO_EX* structure is defined as

```
typedef struct _INTERFACE_INFO_EX
{
    u_long          iiFlags;            /* Interface flags */
    SOCKET_ADDRESS  iiAddress;          /* Interface address */
    SOCKET_ADDRESS  iiBroadcastAddress; /* Broadcast address */
    SOCKET_ADDRESS  iiNetmask;          /* Network mask */
} INTERFACE_INFO_EX, FAR * LPINTERFACE_INFO_EX;
```

The fields have the same meaning as the *INTERFACE_INFO* structure described previously. This ioctl is supported in Windows XP and later versions.

SIO_GET_MULTICAST_FILTER

Which Function?	Input	Output	Winsock Version	Description
WSAIoctl	None	*struct ip_msfilter*	2+	Returns the multicast filter set on a socket

This ioctl retrieves the multicast filter set on a given socket. The multicast state is set with the *SIO_SET_MULTICAST_FILTER* ioctl. This ioctl requires an IGMPv3-enabled network and is supported in only Windows XP. See Chapter 9 for more information about multicasting.

SIO_SET_MULTICAST_FILTER

Which Function?	Input	Output	Winsock Version	Description
WSAIoctl	*struct ip_msfilter*	None	2+	Sets a multicast filter on a socket

This ioctl sets the multicast state. The input parameter is a *struct ip_msfilter*, which is defined as

```
struct ip_msfilter {
         struct in_addr imsf_multiaddr;
         struct in_addr imsf_interface;
         u_long         imsf_fmode;
         u_long         imsf_numsrc;
         struct in_addr imsf_slist[1];
};
```

The first field is the multicast address to join and the second field is the local interface to join the group. The *imsf_fmode* indicates whether the filter state is include or exclude by the defines *MCAST_INCLUDE* and *MCAST_EXCLUDE*, respectively. The *imsf_numsrc* indicates the number of IPv4 source addresses contained in the *imsf_slist* array.

This ioctl can be used to set the multicast state in a single call instead of multiple calls to *IP_ADD_SOURCE_MEMBERSHIP*, *IP_DROP_SOURCE_MEMBERSHIP*, *IP_BLOCK_SOURCE*, and *IP_UNBLOCK_SOURCE*. This ioctl requires an IGMPv3-enabled network and Windows XP. See Chapter 9 for more information.

SIO_INDEX_BIND

Which Function?	Input	Output	Winsock Version	Description
WSAIoctl	*int*	None	2+	Binds the socket to the given interface index

This ioctl binds a socket to an interface index specified as the input parameter instead of an address. This ioctl is supported in Windows 2000 and later versions.

SIO_INDEX_MCASTIF

Which Function?	Input	Output	Winsock Version	Description
WSAIoctl	*Int*	None	2+	Sets the multicast send interface to the specified index

This ioctl sets the outgoing interface for multicast traffic via an interface index as the input parameter instead of an address. This ioctl is supported on Windows 2000 and later versions.

SIO_INDEX_ADD_MCAST

Which Function?	Input	Output	Winsock Version	Description
WSAIoctl	*struct ip_mreq*	None	2+	Joins a multicast group on the specified interface index

This ioctl joins a multicast group using an interface index instead of an address. The input parameter is a *struct ip_mreq* structure except that the *imr_interface* field contains the interface index. See Chapter 9 for more information. This ioctl is supported in Windows 2000 and later versions.

SIO_INDEX_DEL_MCAST

Which Function?	Input	Output	Winsock Version	Description
WSAIoctl	*struct*	None	2+	Drops a multicast group from the specified

	ip_mreq			interface index

This ioctl drops membership to the specified multicast group from the selected interface. This ioctl is supported in Windows 2000 and later versions. See Chapter 9 for more details.

SIO_NSP_NOTIFY_CHANGE

Which Function?	Input	Output	Winsock Version	Description
WSANSPIoctl	None	None	2+	Notifies when a name space query is no longer valid

This ioctl is used to receive notification when the available networks change. This ioctl is supported in Windows XP and later versions. See Chapter 8 for more information about this ioctl.

SIO_QUERY_TARGET_PNP_HANDLE

Which Function?	Input	Output	Winsock Version	Description
WSAIoctl	None	*SOCKET*	2+	Returns the underlying provider's *SOCKET* handle

This ioctl queries the underlying provider for a handle that can be used to receive plug and play event notifications. This ioctl is supported in Windows 2000 and later versions.

SIO_UDP_CONNRESET

Which Function?	Input	Output	Winsock Version	Description
WSAIoctl	*BOOL*	None	2+	Enables ICMP errors to be propagated to the socket

By default, any ICMP errors generated by traffic sent on a UDP socket are not propagated to the socket. For example, if data is sent to an endpoint where there is no socket listening, an ICMP error is returned. When this ioctl is set to TRUE, these errors are propagated to the sending socket, usually in the form of a *WSAECONNRESET* error. This ioctl is supported in Windows 2000 and later versions.

Secure Socket Layer Ioctl Commands

SSL commands are supported only in Windows CE. Currently, Windows 95, Windows 98, Windows Me, Windows NT, Windows 2000, and Windows XP do not provide an SSL-capable provider. Because these commands are available only in Windows CE, the only version of Winsock currently supported for these options is version 1.

SO_SSL_GET_CAPABILITIES

Which Function?	Input	Output	Description
WSAIoctl	None	*DWORD*	Returns the Winsock security provider's capabilities

This command retrieves a set of flags describing the Windows Sockets security provider's capabilities. The output buffer must be a pointer to a *DWORD* bit field. At present, only the flag *SO_CAP_CLIENT* is defined.

SO_SSL_GET_FLAGS

Which Function?	Input	Output	Description
WSAIoctl	None	*DWORD*	Returns *s*-channel–specific flags associated with socket

This command retrieves *s*-channel–specific flags associated with a particular socket. The output buffer must be a pointer to a *DWORD* bit field. See *SO_SSL_SET_FLAGS* below for details of valid flags.

SO_SSL_SET_FLAGS

Which Function?	Input	Output	Description
WSAIoctl	*DWORD*	None	Sets the socket's *s*-channel–specific flags

The input buffer here must be a pointer to a *DWORD* bit field. Currently, the only flag defined is *SSL_FLAG_DEFER_HANDSHAKE*, which allows the application to send and receive plain text data before switching to cipher text. This flag is required for setting up communication through proxy servers.

Normally, the Windows Sockets security provider performs the secure handshake in the Windows Sockets *connect* function. However, if this flag is set, the handshake is deferred until the application issues the *SO_SSL_PERFORM_HANDSHAKE* control code. After the handshake, this flag is reset.

SO_SSL_GET_PROTOCOLS

Which Function?	Input	Output	Description
WSAIoctl	None	*SSLPROTOCOLS*	Returns a list of protocols that the security provider supports

This command retrieves a list of protocols that the provider currently supports on this socket. The output buffer must be a pointer to a *SSLPROTOCOLS* structure, as described here:

```
typedef struct _SSLPROTOCOL
{
    DWORD dwProtocol;
    DWORD dwVersion;
    DWORD dwFlags;
} SSLPROTOCOL, *LPSSLPROTOCOL;
typedef struct _SSLPROTOCOLS
{
    DWORD       dwCount;
    SSLPROTOCOL ProtocolList[1];
} SSLPROTOCOLS, FAR *LPSSLPROTOCOLS;
```

Valid protocols for the *dwProtocol* field include *SSL_PROTOCOL_SSL2*, *SSL_PROTOCOL_SSL3*, and *SSL_PROTOCOL_PCT1*.

SO_SSL_SET_PROTOCOLS

Which Function?	Input	Output	Description
WSAIoctl	*SSLPROTOCOLS*	None	Sets a list of protocols that the underlying provider should support

This ioctl command specifies a list of protocols that the provider is to support on this socket. The input buffer must be a pointer to the *SSLPROTOCOLS* structure described previously.

SO_SSL_SET_VALIDATE_CERT_HOOK

Which Function?	Input	Output	Description
WSAIoctl	*SSLVALIDATECERTHOOK*	None	Sets the validation function for accepting SSL certificates

This ioctl command sets the pointer to the socket's certificate validation hook. It is used to specify the callback function the Windows Sockets security provider invokes when it receives a set of credentials from the remote party. The input buffer must be a pointer to the *SSLVALIDATECERTHOOK* structure, described as follows:

```
typedef struct
{
    SSLVALIDATECERTFUNC    HookFunc;
    LPVOID                 pvArg;
} SSLVALIDATECERTHOOK, *PSSLVALIDATECERTHOOK;
```

The *HookFunc* field is a pointer to a certificate validation callback function; *pvArg* is a pointer to application-specific data and can be used by the application for any purpose.

SO_SSL_PERFORM_HANDSHAKE

Which Function?	Input	Output	Description

WSAIoctl	None	None	Initiates a secure handshake on a connected socket

This ioctl command initiates the secure handshake sequence on a connected socket in which the *SSL_FLAG_DEFER_HANDSHAKE* flag has been set prior to the connection. Data buffers are not required, but the *SSL_FLAG_DEFER_HANDSHAKE* flag will be reset.

ATM Ioctl Commands

The ioctl commands in this section are specific to the ATM protocol family. They are fairly basic, dealing mainly with obtaining the number of ATM devices and ATM addresses of the local interfaces. See Chapter 4 for more detailed information about the addressing mechanisms for ATM.

SIO_GET_NUMBER_OF_ATM_DEVICES

Which Function?	Input	Output	Winsock Version	Description
WSAIoctl	None	*DWORD*	2+	Returns the number of ATM adapters

This ioctl command fills the output buffer pointed to by *lpvOutBuffer* with a *DWORD* containing the number of ATM devices in the system. Each specific device is identified by a unique ID, in the range 0 to the number returned by this ioctl command minus 1.

SIO_GET_ATM_ADDRESS

Which Function?	Input	Output	Winsock Version	Description
WSAIoctl	*DWORD*	*ATM_ADDRESS*	2+	Returns the ATM address for the given device

This ioctl command retrieves the local ATM address associated with the specified device. A device ID of type *DWORD* is specified in the input buffer for this ioctl command, and the output buffer pointed to by *lpvOutBuffer* will be filled with an *ATM_ADDRESS* structure containing a local ATM address suitable for use with *bind*.

SIO_ASSOCIATE_PVC

Which Function?	Input	Output	Winsock Version	Description
WSAIoctl	*ATM_PVC_PARAMS*	None	2+	Associates socket with a permanent virtual circuit

This ioctl command associates the socket with a permanent virtual circuit (PVC), as indicated in the input buffer, which contains the *ATM_PVC_PARAMS* structure. The socket should be of the *AF_ATM* address family. After successfully returning from this function, the application is able to start sending and receiving data as if the connection has been set up.

The *ATM_PVC_PARAMS* structure is defined as

```
typedef struct
{
    ATM_CONNECTION_ID   PvcConnectionId;
    QOS                 PvcQos;
} ATM_PVC_PARAMS;

typedef struct
{
    DWORD  DeviceNumber;
    DWORD  VPI;
    DWORD  VCI;
} ATM_CONNECTION_ID;
```

SIO_GET_ATM_CONNECTION_ID

Which Function?	Input	Output	Winsock Version	Description
Both	None	*ATM_CONNECTION_ID*	2+	Determines whether OOB data has been read

This ioctl command retrieves the ATM Connection ID associated with the socket. Upon successfully returning from this function, the output buffer pointed to by *lpvOutBuffer* is filled with an *ATM_CONNECTION_ID* structure containing the device number and Virtual Path/Channel Identifier (VPI/VCI) values, which are defined in the earlier entry for *SIO_ASSOCIATE_PVC*.

Conclusion

Such an enormous variety of socket options and ioctl commands might seem overwhelming at first, but they do allow applications to access protocol-specific characteristics, as well as offer you the capability to fine-tune an application. In some cases, an application must use one or more socket options or ioctls in order to operate, as in the case of AppleTalk or IrDA. Even so, an application will most likely use only a few options at a time. Of course, one of the more frustrating aspects of socket options and ioctls is that not all options or ioctls are available on every Windows platform, causing trouble for those applications that are attempting to be cross-platform–compatible.

Chapter 8

Registration and Name Resolution

This chapter covers the protocol-independent name registration and resolution model introduced by Winsock 2. The method introduced by Winsock 1 using *GetService* and *SetService* is now obsolete; therefore, we will not cover it. We will first give a bit of background on the importance and uses of name registration and resolution. Then we will move into the different types of name registration models available, followed by a description of the functions that Winsock 2 provides that can be used to resolve names. We will also cover how to register your own services for others to look up, how to query DNS names and how to query for IP network names using the NLA service.

Background

Name registration is the process of associating a user-friendly name with a protocol-specific address. Host names and their IP addresses are a good example. Most people find it cumbersome to remember a workstation's address, such as “157.54.185.186.” They would rather name their machines something easier to remember, such as “MP3Server.” In the case of IP, DNS maps IP addresses to names. Other protocols offer ways of registering their specific addresses to friendlier names. Name spaces will be discussed in more detail in the next section.

Not only do you want to be able to register and resolve host names, you would also like the ability to map your Winsock server's address so that clients can retrieve it to connect to the server. For example, you might have a server running on machine 157.64.185.186 off port 5000. If the server runs on only that machine, you can always hardcode the server address in the client application.

But what if you wanted a more dynamic server—one that can run on multiple machines, perhaps a distributed application with fault tolerance? If one server crashed or was too busy, another instance could be started somewhere else to service clients. In this case, finding out where the servers are possibly running can create headaches. Ideally, you want the ability to register your server—named “Fault-Tolerant Distributed Server”—with multiple addresses. In addition, you want to be able to dynamically update the registered service and its addresses. This is what name registration and resolution is all about, and this chapter will address the facilities Winsock offers to accommodate distributed server registration and name resolution.

Name Space Models

Before we begin to explore the Winsock function, we need to introduce the various name space models that most of the protocols adhere to. A name space offers the capability to associate the protocol and its addressing attributes with a user-friendly name. Some of the more common name spaces are DNS for IP and the NetWare Directory Services (NDS) from Novell for IPX. These name spaces vary widely in their organization and implementation. Some of their properties are particularly important in understanding how to register and resolve names from Winsock.

There are three different types of name spaces: dynamic, static, and persistent. A dynamic name space allows you to register a service on the fly. Also, clients can look up the service at run time. Typically, a dynamic name space relies on periodically broadcasting service information to signal that the service is continuously available. Examples of dynamic name spaces include SAP—used in an IPX networking environment—and AppleTalk's NBP name space.

Static name spaces are the least flexible of the three types. Registering a service in a static name space requires that it be manually registered ahead of time. There is no way to register a name with a static name space from Winsock—there is only a method of resolving names. DNS is an example of a static name space. For example, with DNS you manually enter IP addresses and host names into a file that the DNS service uses to handle resolution requests. The Windows XP platform supports dynamic DNS, but in general, the Winsock interface does not provide a method to update DNS.

Persistent name spaces, like dynamic name spaces, allow services to register on the fly. Unlike dynamic name spaces, however, the persistent model maintains the registration information in nonvolatile storage, such as a file on a disk. Only when the service requests that it be removed will a persistent name space delete its entry. The advantage of a persistent name space is that it is flexible yet does not continually broadcast any kind of availability information. The drawback is that if a service is not well behaved (or is poorly written), it can go away without ever notifying the name space provider to remove its service entry, leading clients to believe incorrectly that the service is still available. NDS is an example of a persistent name space.

Enumerating Name Spaces

Now that you are acquainted with the various attributes of a name space, let's examine how to find out which name spaces are available on a machine. Most of the predefined name spaces are declared in the NSPAPI.H header file. Each name space has an integer value assigned to it. Table 8-1 contains some of the more commonly supported name spaces available on Windows platforms. The name spaces returned depend on which protocols are installed on the workstation. For example, unless IPX/SPX is installed on a workstation, the *NS_SAP* name space will not be returned.

Table 8-1 *Supported Name Spaces*

Name Space	Value	Description

NS_SAP	1	SAP name space; used on IPX networks
NS_NDS	2	NDS name space; used on IPX networks
NS_DNS	11	DNS name space; most commonly found on TCP/IP networks and on the Internet
ND_NTDS	32	Windows NT domain space; protocol-independent name space found on Windows 2000 and Windows XP

When you install IPX/SPX on a machine, the SAP name space is supported for queries only. If you want to register your own service, you also need to install the SAP Agent service. In some cases, the Client Services for NetWare are required to display local IPX interface addresses correctly. Without this service, the local addresses show up as all zeros. In addition, you must add an NDS client to use the NDS name space. You can add all of these protocols and services from the Network Control Panel.

Winsock 2 provides a method of programmatically obtaining a list of the name spaces available on a system. This is accomplished by calling the function *WSAEnumNameSpaceProviders*, which is defined as

```
INT WSAEnumNameSpaceProviders (
    LPDWORD lpdwBufferLength,
    LPWSANAMESPACE_INFO lpnspBuffer
);
```

The first parameter is the size of the buffer submitted as *lpnspBuffer*, which is a sufficiently large array of *WSANAMESPACE_INFO* structures. If the function is called with an insufficiently large buffer, it fails, sets *lpdwBufferLength* to the required minimum size, and causes *WSAGetLastError* to return *WSAEFAULT*. The function returns the number of *WSANAMESPACE_INFO* structures returned, or *SOCKET_ERROR* upon any error.

The *WSANAMESPACE_INFO* structure describes an individual name space installed on the machine. This structure is defined as

```
typedef struct _WSANAMESPACE_INFO {
   GUID NSProviderId;
   DWORD dwNameSpace;
   BOOL fActive;
   DWORD dwVersion;
   LPTSTR lpszIdentifier;
} WSANAMESPACE_INFO, *PWSANAMESPACE_INFO,
*LPWSANAMESPACE_INFO;
```

There are actually two definitions for this structure: Unicode and ANSI. The Winsock 2 header file type defines the appropriate structure to *WSANAMESPACE_INFO* according to how you build your project. Actually, all structures and Winsock 2 registration and name resolution functions have both ANSI and UNICODE versions. The first member of this structure, *NSProviderId*, is a GUID that describes this particular name space. The *dwNameSpace* field is the name space's integer constant, such as *NS_DNS* or *NS_SAP*. The *fActive* member is a Boolean value, which if true indicates that the name space is available and ready to take queries; otherwise, the provider is inactive and unable to take queries that specifically reference this provider. The *dwVersion* field simply identifies the version of this provider. Finally, the *lpszIdentifier* is a descriptive string identifier for this provider.

Registering a Service

The next step is to find out how to set up your own service and make it available and known to other machines on the network. This is known as registering an instance of your service with the name space provider so that it can either be advertised or queried by clients that want to communicate with it. Registering a service is actually a two-step process. The first step is to install a ***service class*** that describes your service's characteristics.

It is important to distinguish between a service class and the actual service. For example, the service class describes which name spaces your service is to be registered with as well as certain characteristics about the service, such as whether it is connection-oriented or connectionless. The service class in no way describes how a client can establish a connection. Once the service class is registered, you register an actual instance of your service that references the correct service class it belongs to. Once this occurs, a client can perform a query to find out where your service instance is running and therefore can attempt communication.

Installing a Service Class

Before you register an instance of a service, you need to define the service class that your service will belong to. A service class defines what name spaces that a service belonging to this class is registered with. The Winsock function that registers a service class is *WSAInstallServiceClass*, which is defined as

```
INT WSAInstallServiceClass (LPWSASERVICECLASSINFO lpServiceClassInfo);
```

The single parameter *lpServiceClassInfo* points to a *WSASERVICECLASSINFO* structure that defines the attributes of this class. The structure is defined as

```
typedef struct _WSAServiceClassInfo {
    LPGUID              lpServiceClassId;
    LPTSTR              lpszServiceClassName;
    DWORD               dwCount;
    LPWSANSCLASSINFO    lpClassInfos;
} WSASERVICECLASSINFO, *PWSASERVICECLASSINFO, *LPWSASERVICECLASSINFO;
```

The first field is a GUID that uniquely identifies this particular service class. There are a couple of ways to generate a GUID to use here. One way is to use the utility UUIDGEN.EXE and create a GUID for this service class. The problem with this method is that if you need to refer back to this GUID, you basically have to hardcode its value into a header file somewhere. This is where the second solution is useful. Within the header file SVCGUID.H, several macros generate a GUID based on a simple attribute. For example, if you install a service class for SAP that will be used to advertise your IPX application, you can use the *SVCID_NETWARE* macro. The only parameter is the SAP ID number you assign to your “class” of applications. A number of SAP IDs are predefined in NetWare, such as 0x4 for file servers and 0x7 for a print server. Using this method, all you need is the easy-to-remember SAP ID to generate the GUID for the corresponding service class. In addition, several macros exist that accept a port number as a parameter and return the corresponding service's GUID. Take a look at the header file SVCGUID.H, which contains other useful macros for the reverse operation—extracting the service port number from a GUID. Table 8-2 lists the most commonly used macros for generating GUIDs from simple protocol attributes such as port numbers or SAP IDs. The header file also contains constants for well-known port numbers for services such as FTP and Telnet.

Table 8-2 *Common Service ID Macros*

Macro	Description
SVCID_TCP(Port)	Generates a GUID from a TCP port number
SVCID_DNS(RecordType)	Generates a GUID from a DNS record type
SVCID_UDP(Port)	Generates a GUID from a UDP port number
SVCID_NETWARE(SapId)	Generates a GUID from an SAP ID number

The second field of the *WSASERVICECLASSINFO* structure, *lpszServiceClassName*, is simply a string name for this particular service class. The last two fields are related. The *dwCount* field refers to the number of *WSANSCLASSINFO* structures passed in the *lpClassInfos* field. These structures define the name spaces and protocol characteristics that apply to the services that register under this service class. The structure is defined as

```
typedef struct _WSANSClassInfo {
    LPSTR   lpszName;
    DWORD   dwNameSpace;
    DWORD   dwValueType;
    DWORD   dwValueSize;
    LPVOID  lpValue;
}WSANSCLASSINFO, *PWSANSCLASSINFO, *LPWSANSCLASSINFO;
```

The *lpszName* field defines the attribute that the service class possesses. Table 8-3 lists the various attributes available. Every attribute listed has a value type of *REG_DWORD*.

Table 8-3 *Service Types*

String Value	Constant Define	Name Space	Description
"SapId"	*SERVICE_TYPE_VALUE_SAPID*	*NS_SAP*	SAP identifier
"Connection-Oriented"	*SERVICE_TYPE_VALUE_CONN*	Any	Indicates whether service is connection-oriented or connectionless
"TcpPort"	*SERVICE_TYPE_VALUE_TCPPORT*	*NS_DNS* *NS_NTDS*	TCP port
"UdpPort"	*SERVICE_TYPE_VALUE_UDPPORT*	*NS_DNS* *NS_NTDS*	UDP port

The *dwNameSpace* is the name space this attribute applies to. Table 8-3 also lists the name spaces to which the various service types usually apply. The last three fields, *dwValueType*, *dwValueSize*, and *lpValue*, all describe the value associated with the service type. The *dwValueType* field signifies the type of data associated with this entry and therefore can be one of the registry type values. For example, if the value is a *DWORD*, the value type is *REG_DWORD*. The next field, *dwValueSize*, is simply the size of the data passed as *lpValue*, which is a pointer to the data.

The following code example illustrates how to install a service class named “Widget Server Class.”

```
WSASERVICECLASSINFO   sci;
WSANSCLASSINFO        aNameSpaceClassInfo[4];
DWORD                 dwSapId = 200,
                      dwUdpPort = 5150,
                      dwZero = 0;
int                   ret;

memset(&sci, 0, sizeof(sci));

SET_NETWARE_SVCID(&sci.lpServiceClassId, dwSapId);
sci.lpszServiceClassName = (LPSTR)"Widget Server Class";
sci.dwCount = 4;
sci.lpClassInfos = aNameSpaceClassInfo;

memset(aNameSpaceClassInfo, 0, sizeof(WSANSCLASSINFO) * 4);
// NTDS name space setup
aNameSpaceClassInfo[0].lpszName = SERVICE_TYPE_VALUE_CONN;
aNameSpaceClassInfo[0].dwNameSpace = NS_NTDS;
aNameSpaceClassInfo[0].dwValueType = REG_DWORD;
aNameSpaceClassInfo[0].dwValueSize = sizeof(DWORD);
aNameSpaceClassInfo[0].lpValue = &dwZero;

aNameSpaceClassInfo[1].lpszName = SERVICE_TYPE_VALUE_UDPPORT;
aNameSpaceClassInfo[1].dwNameSpace = NS_NTDS;
aNameSpaceClassInfo[1].dwValueType = REG_DWORD;
aNameSpaceClassInfo[1].dwValueSize = sizeof(DWORD);
aNameSpaceClassInfo[1].lpValue = &dwUdpPort;

// SAP name space setup
aNameSpaceClassInfo[2].lpszName = SERVICE_TYPE_VALUE_CONN;
aNameSpaceClassInfo[2].dwNameSpace = NS_SAP;
aNameSpaceClassInfo[2].dwValueType = REG_DWORD;
aNameSpaceClassInfo[2].dwValueSize = sizeof(DWORD);
aNameSpaceClassInfo[2].lpValue = &dwZero;

aNameSpaceClassInfo[3].lpszName = SERVICE_TYPE_VALUE_SAPID;
aNameSpaceClassInfo[3].dwNameSpace = NS_SAP;
aNameSpaceClassInfo[3].dwValueType = REG_DWORD;
aNameSpaceClassInfo[3].dwValueSize = sizeof(DWORD);
aNameSpaceClassInfo[3].lpValue = &dwSapId;

WSAInstallServiceClass(&sci);
```

The first noticeable thing this example illustrates is to pick a GUID that this class will be registered under. The services you are designing all belong to the class “Widget Server Class,” and this service class describes the general attributes belonging to an instance of the service. In this example, we chose to register this class with the NetWare SAP ID of 200. This is only for convenience. We could have picked an arbitrary GUID or even the GUID based on the UDP port number. In addition, the service can use the UDP protocol, in which case the clients are listening on port 5150.

The next step of note is setting the *dwCount* field of the *WSASERVICECLASSINFO* to 4. In this example, you will register this service class with both the SAP name space (*NS_SAP*) and the Windows NT domain space (*NS_NTDS*). The odd part you'll notice is that we use four

WSANSCLASSINFO structures even though we are registering the service class with only two name spaces. This is because we define two attributes for each name space and each attribute requires a separate *WSANSCLASSINFO* structure. For each name space, we define whether the service will be connection-oriented. In this example, the name space is connectionless because we set the value for *SERVICE_TYPE_VALUE_CONN* to be a Boolean 0. For the Windows NT domain space, we also set the UDP port number this service normally runs under by using the service type *SERVICE_TYPE_VALUE_UDPPORT*. For the SAP name space, we set the SAP ID of our service with service type *SERVICE_TYPE_VALUE_SAPID*.

For every *WSANSCLASSINFO* entry, you must set the name space identifier that this service type applies to, as well as the type and size of the value. Table 8-3 contains the types required for the service types, which all turn out to be *DWORD* in the example. The last step is simply to call *WSAInstallServiceClass* and pass the *WSASERVICECLASSINFO* structure as the parameter. If *WSAInstallServiceClass* is successful, the function returns 0; otherwise, it returns *SOCKET_ERROR*. If *WSASERVICECLASSINFO* is invalid or improperly formed, *WSAGetLastError* returns *WSAEINVAL*. If the service class already exists, then *WSAGetLastError* returns *WSAEALREADY*. In this case, a service class can be removed by calling *WSARemoveServiceClass*, which is declared as

```
INT WSARemoveServiceClass( LPGUID lpServiceClassId );
```

This function's only parameter is a pointer to the GUID that defines the given service class.

Service Registration

Once you have a service class installed that describes the general attributes of your service, you can register an instance of your service so that it is available for lookup by other clients on remote machines. The Winsock function to register an instance of a service is *WSASetService*.

```
INT WSASetService (
    LPWSAQUERYSET lpqsRegInfo,
    WSAESETSERVICEOP essOperation,
    DWORD dwControlFlags
);
```

The first parameter, *lpqsRegInfo*, is a pointer to a *WSAQUERYSET* structure that defines the particular service. We'll discuss what goes in this structure shortly. The *essOperation* parameter specifies the action to take place, such as registration or deregistration. Table 8-4 describes the three valid flags.

Table 8-4 *Set Service Flags*

Operation Flag	Meaning
RNRSERVICE_REGISTER	Register the service. For dynamic name providers, this means to begin actively advertising the service. For persistent name providers, this means updating the database. For static name providers, this does nothing.
	Remove the entire service from the registry. For dynamic name providers, this means to stop advertising the service. For persistent

RNRSERVICE_DEREGISTER	name providers, this means removing the service from the database. For static name providers, this does nothing.
RNRSERVICE_DELETE	Remove only the given instance of the service from the name space. A service might be regist-ered that contains multiple instances (using the *SERVICE_MULTIPLE* flag upon registration), and this command removes only the given instance of the service (as defined by a *CSADDR_INFO* structure). Again, this applies only to dynamic and persistent name providers.

The third parameter, *dwControlFlags*, is either 0 or the flag *SERVICE_ MULTIPLE*. This flag is used if multiple addresses will be registered under the given service instance. For example, say you have a service that you want to run on five machines. The *WSAQUERYSET* structure passed into *WSASetService* would reference five *CSADDR_INFO* structures, each describing the location of one instance of the service. This requires the *SERVICE_MULTIPLE* flag to be set. In addition, at some later point you can deregister a single instance of the service by using the *RNRSERVICE_DELETE* service flag. Table 8-5 gives the possible combinations of the operation and control flags and describes the result of the command, depending on whether the service already exists.

Table 8-5 *WSASetService Flag Combinations*

RNRSERVICE_REGISTER **Flags**	**Meaning**	
	If the Service Already Exists	**If the Service Does Not Exist**
None	Overwrite the existing service instance.	Add a new service entry on the given address.
SERVICE_MULTIPLE	Update the service instance by adding the new addresses.	Add a new service entry on the given addresses.
RNRSERVICE_DEREGISTER **Flags**	**Meaning**	
	If the Service Already Exists	**If the Service Does Not Exist**
None	Remove all instances of the service, but do not remove the service. (Basically, *WSAQUERYSET* remains, but the number of *CSADDR_INFO* structures is 0.)	This is an error, and *WSASERVICE_NOT_FOUND* is returned.
SERVICE_MULTIPLE	Update the service by removing the given addresses. The service remains registered, even if no addresses remain.	This is an error, and *WSASERVICE_NOT_FOUND* is returned.
RNRSERVICE_DELETE **Flags**	**Meaning**	

	If the Service Already Exists	If the Service Does Not Exist
None	The service is removed completely from the name space.	This is an error, and *WSASERVICE_NOT_FOUND* is returned.
SERVICE_MULTIPLE	Update the service by removing the given addresses. If no addresses remain, the service is completely removed from the name space.	This is an error, and *WSASERVICE_NOT_FOUND* is returned.

Now that you have an understanding of what *WSASetService* does, let's take a look at the *WSAQUERYSET* structure that needs to be filled out and passed into the function. This structure is defined as

```
typedef struct _WSAQuerySetW {
    DWORD           dwSize;
    LPTSTR          lpszServiceInstanceName;
    LPGUID          lpServiceClassId;
    LPWSAVERSION    lpVersion;
    LPTSTR          lpszComment;
    DWORD           dwNameSpace;
    LPGUID          lpNSProviderId;
    LPTSTR          lpszContext;
    DWORD           dwNumberOfProtocols;
    LPAFPROTOCOLS   lpafpProtocols;
    LPTSTR          lpszQueryString;
    DWORD           dwNumberOfCsAddrs;
    LPCSADDR_INFO   lpcsaBuffer;
    DWORD           dwOutputFlags;
    LPBLOB          lpBlob;
} WSAQUERYSETW, *PWSAQUERYSETW, *LPWSAQUERYSETW;
```

The *dwSize* field should be set to the size of the *WSAQUERYSET* structure. The *lpszServiceInstanceName* field contains a string identifier naming this instance of the server. The *lpServiceClassId* field is the GUID for the service class that this service instance belongs to. The *lpVersion* field is optional. You can use it to supply version information that could be useful when a client queries for a service. The *lpszComment* field is also optional. You can specify any kind of comment string here. The *dwNameSpace* field specifies the name spaces to register your service with. If you're using only a single name space, use that value only; otherwise, use *NS_ALL*. It is possible to reference a custom name space provider. (Writing your own name space is discussed in Chapter 12.) For a custom name space provider, the *dwNameSpace* field is set to 0 and *lpNSProviderId* specifies the GUID representing the custom provider. The *lpszContext* field specifies the starting point of the query in a hierarchical name space such as NDS.

The *dwNumberOfProtocols* and *lpafpProtocols* fields are optional parameters used to narrow the search to return only the supplied protocols. The *dwNumberOfProtocols* field references the number of *AFPROTOCOLS* structures contained in the *lpafpProtocols* array. The structure is defined as

```
typedef struct _AFPROTOCOLS {
    INT iAddressFamily;
    INT iProtocol;
} AFPROTOCOLS, *PAFPROTOCOLS, *LPAFPROTOCOLS;
```

The first field, *iAddressFamily*, is the address family constant, such as *AF_INET* or *AF_IPX*. The second field, *iProtocol*, is the protocol from the given address family, such as *IPPROTO_TCP* or *NSPROTO_IPX*.

The next field in the *WSAQUERYSET* structure, *lpszQueryString*, is optional and used only by name spaces supporting enriched Structured Query Language (SQL) queries such as Whois++. This parameter is used to specify that string.

The next two fields are the most important when registering a service. The *dwNumberOfCsAddrs* field simply provides the number of *CSADDR_INFO* structures passed in *lpcsaBuffer*. The *CSADDR_INFO* structure defines the address family and the address where the service is located. If multiple structures are present, multiple instances of the service are available. The structure is defined as

```
typedef struct _CSADDR_INFO {
    SOCKET_ADDRESS LocalAddr;
    SOCKET_ADDRESS RemoteAddr;
    INT            iSocketType;
    INT            iProtocol;
} CSADDR_INFO;

typedef struct _SOCKET_ADDRESS {
    LPSOCKADDR lpSockaddr;
    INT        iSockaddrLength;
} SOCKET_ADDRESS, *PSOCKET_ADDRESS, FAR * LPSOCKET_ADDRESS;
```

In addition, the definition of *SOCKET_ADDRESS* is included. When registering a service, you can specify the local and remote addresses. The local address field (*LocalAddr*) is used to specify the address that an instance of this service should bind to, and the remote address field (*RemoteAddr*) is the address a client should use in a *connect* or a *sendto* call. The other two fields, *iSocketType* and *iProtocol*, specify the socket type (for example, *SOCK_STREAM* or *SOCK_DGRAM*) and the protocol family (for example, *AF_INET, AF_IPX*) for the given addresses.

The last two fields of the *WSAQUERYSET* structure are *dwOutputFlags* and *lpBlob*. These two fields are generally not needed for service registration; they are more useful when querying for a service instance (covered in the next section). Only the name space provider can return a *BLOB* structure. That is, when registering a service you cannot add your own *BLOB* structure to be returned in client queries.

Table 8-6 lists the fields of the *WSAQUERYSET* structure and identifies which are required or optional depending on whether a query or a registration is being performed.

Table 8-6 *WSAQUERYSET Fields*

Field	Query	Registration

dwSize	Required	Required
lpszServiceInstanceName	String or “*” required	Required
lpServiceClassId	Required	Required
lpVersion	Optional	Optional
lpszComment	Ignored	Optional
dwNameSpace *lpNSProviderId*	One of these two fields must be specified	One of these two fields must be specified
lpszContext	Optional	Optional
dwNumberOfProtocols	Zero or more	Zero or more
lpafpProtocols	Optional	Optional
lpszQueryString	Optional	Ignored
dwNumberOfCsAddrs	Ignored	Required
lpcsaBuffer	Ignored	Required
dwOutputFlags	Ignored	Optional
lpBlob	Ignored, can be returned by the query	Ignored

Service Registration Example

In this section, we'll show you how to register your own service under both the SAP and NTDS name spaces. The Windows NT domain space is quite powerful, which is why we want to include it in our example. However, there are a few features to be aware of before you begin. First, the Windows NT domain space requires Windows 2000 or Windows XP because it is based on the Active Directory directory service. This also means that the Windows 2000 or Windows XP workstation on which you hope to register and/or look up services must have a machine account in that domain in order to access Active Directory. The other feature to note is that the Windows NT domain space is capable of registering socket addresses from any protocol family. Your IP and IPX services can all be registered in the same name space. Also, there is a dynamic way of adding and removing IP-based services. The following code example illustrates the basic steps required to register an instance of a service. For simplicity, no error checking is performed.

```
SOCKET              socks[2];
WSAQUERYSET         qs;
CSADDR_INFO         lpCSAddr[2];
SOCKADDR_IN         sa_in;
SOCKADDR_IPX        sa_ipx;
IPX_ADDRESS_DATA    ipx_data;
GUID                guid = SVCID_NETWARE(200);
int                 ret, cb;

memset(&qs, 0, sizeof(WSAQUERYSET));
qs.dwSize = sizeof(WSAQUERYSET);
qs.lpszServiceInstanceName = (LPSTR)"Widget Server";
qs.lpServiceClassId = &guid;
qs.dwNameSpace = NS_ALL;
qs.lpNSProviderId = NULL;
qs.lpcsaBuffer = lpCSAddr;
qs.lpBlob = NULL;
//
// Set the IP address of our service
//
memset(&sa_in, 0, sizeof(sa_in));
sa_in.sin_family = AF_INET;
sa_in.sin_addr.s_addr =  htonl(INADDR_ANY);
sa_in.sin_port = 5150;

socks[0] = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);
ret = bind(socks[0], (SOCKADDR *)&sa_in, sizeof(sa_in));

cb = sizeof(sa_in);
getsockname(socks[0], (SOCKADDR *)&sa_in, &cb);

lpCSAddr[0].iSocketType = SOCK_DGRAM;
lpCSAddr[0].iProtocol = IPPROTO_UDP;
lpCSAddr[0].LocalAddr.lpSockaddr = (SOCKADDR *)&sa_in;
lpCSAddr[0].LocalAddr.iSockaddrLength = sizeof(sa_in);
lpCSAddr[0].RemoteAddr.lpSockaddr = (SOCKADDR *)&sa_in;
lpCSAddr[0].RemoteAddr.iSockaddrLength = sizeof(sa_in);
//
// Set up the IPX address for our service
//
memset(sa_ipx.sa_netnum, 0, sizeof(sa_ipx.sa_netnum));
memset(sa_ipx.sa_nodenum, 0, sizeof(sa_ipx.sa_nodenum));
sa_ipx.sa_family = AF_IPX;
sa_ipx.sa_socket = 0;

socks[1] = socket(AF_IPX, SOCK_DGRAM, NSPROTO_IPX);

ret = bind(socks[1], (SOCKADDR *)&sa_ipx, sizeof(sa_ipx));

cb = sizeof(IPX_ADDRESS_DATA);
memset (&ipx_data, 0, cb);
ipx_data.adapternum = 0;

ret = getsockopt(socks[1], NSPROTO_IPX, IPX_ADDRESS,
    (char *)&ipx_data, &cb);

cb = sizeof(SOCKADDR_IPX);
getsockname(socks[1], (SOCKADDR *)sa_ipx, &cb);

memcpy(sa_ipx.sa_netnum, ipx_data.netnum, sizeof(sa_ipx.sa_netnum));
memcpy(sa_ipx.sa_nodenum, ipx_data.nodenum, sizeof(sa_ipx.sa_nodenum));

lpCSAddr[1].iSocketType = SOCK_DGRAM;
lpCSAddr[1].iProtocol = NSPROTO_IPX;
lpCSAddr[1].LocalAddr.lpSockaddr = (struct sockaddr *)&sa_ipx;
lpCSAddr[1].LocalAddr.iSockaddrLength = sizeof(sa_ipx);
lpCSAddr[1].RemoteAddr.lpSockaddr = (struct sockaddr *)&sa_ipx;
lpCSAddr[1].RemoteAddr.iSockaddrLength = sizeof(sa_ipx);

qs.dwNumberOfCsAddrs = 2;

WSASetService(&qs, RNRSERVICE_REGISTER, 0L);
```

The example illustrates how to set an instance of a service so that a client of that service can find out the address that it needs to communicate with the service. The first order of business is to initialize the *WSAQUERYSET* structure. We also need to give a name to the instance of our service. In this case, we simply call it “Widget Server.” The other critical step is to use the same GUID we used to register our service class. Whenever you register an instance of a service, that service must belong to a service class. In this case, we use the “Widget Service Class” (defined in the previous section), whose GUID is *SVCID_NETWARE(200)*. The next step is to set the name spaces that we are interested in. Because our service runs over both IPX and UDP, we specify *NS_ALL*. Because we're specifying a preexisting name space, *lpNSProviderId* must be set to *NULL*.

The next step is to set up the *SOCKADDR* structures within the *CSADDR_INFO* array that *WSASetService* passes as the *lpcsaBuffer* field of the *WSAQUERYSET* structure. You'll notice that in our example we actually create the sockets and bind them to a local address before we set up the *SOCKADDR* structure. This is because we need to find the exact local address that clients need to connect to. For example, when creating our UDP socket for the server, we bind to *INADDR_ANY*, which doesn't give us the actual IP address until we call *getsockname*. Using the information returned from *getsockname*, we can build a *SOCKADDR_IN* structure. Within the *CSADDR_INFO* structure, we set the socket type and the protocol. The other two fields are the local and remote address information. The local address is the address that a server should bind to, and the remote address is the address that a client should use to connect to the service.

After setting up the *SOCKADDR_IN* structure for our UDP-based server, we set up the IPX-based service. In Chapter 4, you saw that servers should bind to the internal network number by setting the network and node number to 0. Again, this doesn't give you the address that clients need, so call the socket option *IPX_ADDRESS* to get the actual address. In filling the *CSADDR_INFO* structure for IPX, use *SOCK_DGRAM* and *NSPROTO_IPX* for the socket type and the protocol, respectively. The last step is to set the *dwNumberOfCsAddrs* field in the *WSAQUERYSET* structure to 2 because there are two addresses—UDP and IPX—that clients can use to establish a connection. Finally, call *WSASetService* with our *WSAQUERYSET* structure, the *RNRSERVICE_REGISTER* flag, and no control flags. You do not specify the *SERVICE_MULTIPLE* control flag so that if you choose to deregister our service, all instances of the service (both the IPX and UDP addresses) will be deregistered.

There is one consideration that the example does not take into account: multihomed machines. If you create a UDP-based server that binds to *INADDR_ANY* on a multihomed machine, the client can connect to the server on any of the available interfaces. With IP, *getsockname* is not sufficient; you must obtain all of the local IP interfaces. There are a number of methods for getting this information, depending on the platform you are on. One method common to all platforms is calling *gethostbyname* to return a list of IP addresses for our name. Under Winsock 2, you can also call the ioctl command *SIO_GET_INTERFACE_LIST*. For Windows 2000 and Windows XP, the ioctl *SIO_ADDRESS_LIST_QUERY* is available. Finally, the IP helper functions discussed in Chapter 16 can be used as well. Simple TCP/IP name resolution and *gethostbyname* are presented in Chapter 3, and ioctl commands are found in Chapter 7.

There is a full-fledged example that addresses multihomed machines in a file named RNRCS.CPP on the companion CD.

Querying a Service

Now that you know how to register a service within a name space, let's look at how a client can query the name space for a given service so it can get information about the service for communication purposes. Name resolution is quite a bit simpler than service registration, even though name resolution uses three functions for querying: *WSALookupServiceBegin*, *WSALookupServiceNext*, and *WSALookupServiceEnd*.

The first step when performing a query is to call *WSALookupServiceBegin*, which initiates the query by setting up the constraints within which the query will act. The function prototype is as follows:

```
INT WSALookupServiceBegin (
    LPWSAQUERYSET lpqsRestrictions,
    DWORD dwControlFlags,
    LPHANDLE lphLookup
);
```

The first parameter is a *WSAQUERYSET* structure that places constraints on the query, such as limiting the name spaces to query. The second parameter, *dwControlFlags*, determines the depth of the search. Table 8-7 contains the various possible flags and their meanings. These flags affect how the query behaves as well as which data is returned from the query. The last parameter is of type *HANDLE* and is initialized upon function return. The return value is 0 on success; otherwise, *SOCKET_ERROR* is returned. If one or more parameters are invalid, *WSAGetLastError* returns *WSAEINVAL*. If the name is found in the name space but no data matches the given restrictions, the error is *WSANO_DATA*. If the given service does not exist, *WSASERVICE_NOT_FOUND* is the error.

Table 8-7 *Control Flags*

Flag	Meaning
LUP_DEEP	In hierarchical name spaces, query deep as opposed to the first level.
LUP_CONTAINERS	Retrieve container objects only. This flag pertains to hierarchical name spaces only.
LUP_NOCONTAINERS	Do not return any containers. This flag pertains to hierarchical name spaces only.
LUP_FLUSHCACHE	Ignore cached information, and query the name space directly. Note that not all name providers cache queries.
LUP_FLUSHPREVIOUS	Instruct the name provider to discard the information set previously returned. This flag is typically used after *WSALookupServiceNext* returns *WSA_NOT_ENOUGH_MEMORY*. The information that was too big for the supplied buffer is discarded, and the next information set is to be retrieved.
LUP_NEAREST	Retrieve the results in order of distance. Note that it is up to the name provider to calculate this distance metric, as there is no specific provision for this information when a service is registered. Name providers are not required to support this concept.
	Specifies that the local address be returned in the *CSADDR_INFO*

LUP_RES_SERVICE	structure
LUP_RETURN_ADDR	Retrieve the addresses as *lpcsaBuffer*.
LUP_RETURN_ALIASES	Retrieve only alias information. Each alias is returned in successive calls to *WSALookupServiceNext* and will have the *RESULT_IS_ALIAS* flag set.
LUP_RETURN_ALL	Retrieve all available information.
LUP_RETURN_BLOB	Retrieve the private data as *lpBlob*.
LUP_RETURN_COMMENT	Retrieve the comment as *lpszComment*.
LUP_RETURN_NAME	Retrieve the name as *lpszServiceInstanceName*.
LUP_RETURN_TYPE	Retrieve the type as *lpServiceClassId*.
LUP_RETURN_VERSION	Retrieve the version as *lpVersion*.

When you make a call to *WSALookupServiceBegin*, a handle for the query is returned that you pass to *WSALookupServiceNext*, which returns the data to you. The function is defined as

```
INT WSALookupServiceNext (
    HANDLE hLookup,
    DWORD dwControlFlags,
    LPDWORD lpdwBufferLength,
    LPWSAQUERYSET lpqsResults
);
```

The handle *hLookup* is returned from *WSALookupServiceBegin*. The *dwControlFlags* parameter has the same meaning as in *WSALookupServiceBegin* except that only *LUP_FLUSHPREVIOUS* is supported. The parameter *lpdwBufferLength* is the length of the buffer passed as *lpqsResults*. Because the *WSAQUERYSET* structure could contain binary large object (BLOB) data, it is often required that you pass a buffer larger than the structure itself. If the buffer size is insufficient for the data to be returned, the function call fails with *WSA_NOT_ENOUGH_MEMORY*.

Once you have initiated the query with *WSALookupServiceBegin*, call *WSALookupServiceNext* until the error *WSA_E_NO_MORE* (10110) is generated. Caution: in earlier implementations of Winsock, the error code for no more data is *WSAENOMORE* (10102), so robust code should check for both error codes. Once all the data has been returned or you have finished querying, call *WSALookupServiceEnd* with the *HANDLE* variable used in the queries. The function is

```
INT WSALookupServiceEnd ( HANDLE hLookup );
```

Forming a Query

Let's look at how you can query the service you registered in the previous section. The first thing to do is set up a *WSAQUERYSET* structure that defines the query. Look at the following code.

```
WSAQUERYSET   qs;
GUID          guid = SVCID_NETWARE(200);
AFPROTOCOLS   afp[2] = {{AF_IPX,  NSPROTO_IPX}, {AF_INET, IPPROTO_UDP}};
HANDLE        hLookup;
```

```
int           ret;

memset(&qs, 0, sizeof(qs));
qs.dwSize = sizeof (WSAQUERYSET);
qs.lpszServiceInstanceName = "Widget Server";
qs.lpServiceClassId = &guid;
qs.dwNameSpace = NS_ALL;
qs.dwNumberOfProtocols = 2;
qs.lpafpProtocols = afp;

ret = WSALookupServiceBegin(&qs, LUP_RETURN_ADDR | LUP_RETURN_NAME,
         &hLookup);
if (ret == SOCKET_ERROR)
    // Error
```

Remember that all service lookups are based on the service class GUID that the service you are searching for is based on. The variable *guid* is set to the service class ID of our server. You first initialize *qs* to 0 and set the *dwSize* field to the size of the structure. The next step is to give the name of the service you are searching for. The service name can be the exact name of the server, or you can specify a wildcard (*) that will return all services of the given service class GUID. Next, tell the query to search all name spaces by using the *NS_ALL* constant. Last, set up the protocols that the client is capable of connecting with, which are IPX and UDP/IP. This is done by using an array of two *AFPROTOCOLS* structures.

To begin the query, you must call *WSALookupServiceBegin*. The first parameter is our *WSAQUERYSET* structure, and the next parameters are flags defining which data should be returned if a matching service is found. Here you specify that you want addressing information and the service name by logically *OR*ing the two flags *LUP_RETURN_ADDR* and *LUP_RETURN_NAME*. The flag *LUP_RETURN_NAME* is necessary only if you're specifying the wildcard (*) for the service name; otherwise, you already know the service name. The last parameter is a *HANDLE* variable that identifies this particular query. It will be initialized upon successful return.

Once the query is successfully opened, you call *WSALookupServiceNext* until *WSA_E_NO_MORE* is returned. Each successful call returns information about a service that matches our criteria. Here is what the code looks like for this step:

```
char          buff[sizeof(WSAQUERYSET) + 2000];
DWORD         dwLength, dwErr;
WSAQUERYSET  *pqs = NULL;
SOCKADDR     *addr;
int           i;

pqs = (WSAQUERYSET *)buff;
dwLength = sizeof(WSAQUERYSET) + 2000;
while (1)
{
    ret = WSALookupServiceNext(hLookup, 0, &dwLength, pqs);
    if (ret == SOCKET_ERROR)
    {
        if ((dwErr = WSAGetLastError()) == WSAEFAULT)
        {
            printf("Buffer too small; required size is: %d\n", dwLength);
```

```
         break;
     }
          else if ((dwErr == WSA_E_NO_MORE) || (dwErr = WSAENOMORE))
         break;
     else
     {
         printf("Failed with error: %d\n", dwErr);
         break;
     }
 }
 for (i = 0; i < pqs->dwNumberOfCsAddrs; i++)
 {
     addr = (SOCKADDR *)pqs->lpcsaBuffer[i].RemoteAddr.lpSockaddr;
     if (addr->sa_family == AF_INET)
     {
         SOCKADDR_IN *ipaddr = (SOCKADDR_IN *)addr;
         printf("IP address:port = %s:%d\n",
         inet_ntoa(ipaddr->sin_addr),
         ipaddr->sin_port);
     }
     else if (addr->sa_family == AF_IPX)
     {
         CHAR  IPXString[64];
         DWORD IPXStringSize = sizeof(IPXString);
         if (WSAAddressToString((LPSOCKADDR) addr,
             sizeof(SOCKADDR_IPX), NULL, IPXString,
             &IPXStringSize) == SOCKET_ERROR)
             printf("WSAAddressToString returned error %d\n",
                 WSAGetLastError());
         else
                printf("IPX address: %s\n", IPXString);
        }
    }
}
WSALookupServiceEnd(hLookup);
```

The example code is straightforward, although a bit simplified. Calling *WSALookupServiceNext* requires only a valid handle to a query, the length of the return buffer, and the return buffer. You don't need to specify any control flags because the only valid flag for this function is *LUP_FLUSHPREVIOUS*. If our supplied buffer is too small and this flag is set, the results from this call are discarded. However, in this example we don't use *LUP_FLUSHPREVIOUS*, and if our buffer is too small, the *WSAEFAULT* error is generated. If this occurs, *lpdwBufferLength* is set to the required size. Our example uses a fixed-size buffer equal to the size of the *WSAQUERYSET* structure plus 2000 bytes. Because all you are asking for are service names and addresses, this should be a sufficient size. Of course, in production code, your applications should be prepared to handle the *WSAEFAULT* error.

Once you successfully call WSALookupServiceNext, the buffer is filled with a WSAQUERYSET structure containing the results. In our query, you asked for names and addresses; the fields of *WSAQUERYSET* that are of most interest are *lpszServiceInstanceName* and *lpcsaBuffer*. The former contains the name of the service, the latter is an array of *CSADDR_INFO* structures that contains addressing information for the service. The parameter *dwNumberOfCsAddrs* tells us exactly how many addresses have been returned. In the example code, all we do is simply print out the addresses.

Check for only IPX and IP addresses to print because those are the only address families you requested when you opened the query.

If our query used a wildcard (*) for the service name, each call to *WSALookupServiceNext* would return a particular instance of that service running somewhere on the network—provided, of course, that multiple instances are actually registered and running. Once all instances of the service have been returned, the error *WSA_E_NO_MORE* is generated, and you break out of the loop. The last thing you do is call *WSALookupServiceEnd* on the query handle. This releases any resources allocated for the query.

Querying DNS

Previously we mentioned that the DNS name space is static, which means you cannot dynamically register your service; however, you can still use the Winsock name resolution functions to perform a DNS query. Performing a DNS query is actually a bit more complicated than performing a normal query for a service that you have registered because the DNS name space provider returns the query information as a BLOB. Why does it do this? Remember from the Chapter 3 discussion of *gethostbyname* that a name lookup returns a *HOSTENT* structure that contains not only IP addresses but also aliases. That information doesn't quite fit into the fields of the *WSAQUERYSET* structure.

The tricky aspect about BLOB data is that its data format is not well documented, which makes directly querying DNS challenging. First, let's take a look at how to open the query. The file DNSQUERY.CPP on the companion CD contains the entire example code for querying DNS directly; however, we'll take a look at it piece by piece. The following code illustrates initializing the DNS query:

```
WSAQUERYSET qs;
AFPROTOCOLS afp [2] = {{AF_INET, IPPROTO_UDP},{AF_INET, IPPROTO_TCP}};
GUID        hostnameguid = SVCID_INET_HOSTADDRBYNAME;
DWORD       dwLength = sizeof(WSAQUERYSET) + sizeof(HOSTENT) + 2048;
char        buff[dwLength];
HANDLE       hQuery;
int          ret;

qs = (WSAQUERYSET *)buff;
memset(&qs, 0, sizeof(qs));
qs.dwSize = sizeof(WSAQUERYSET);
qs.lpszServiceInstanceName = argv[1];
qs.lpServiceClassId = &hostnameguid;
qs.dwNameSpace = NS_DNS;
qs.dwNumberOfProtocols = 2;
qs.lpafpProtocols = afp;

ret = WSALookupServiceBegin(&qs, LUP_RETURN_NAME | LUP_RETURN_BLOB,
    &hQuery);
if (ret == SOCKET_ERROR)
    // Error
```

Setting up the query is quite similar to our previous example. The most noticeable change is that we use the predefined GUID *SVCID_INET_ HOSTADDRBYNAME*. This is the GUID that identifies host name queries. The *lpszServiceInstanceName* is the host name that we want to resolve. Because we are resolving host names through DNS, we need to specify only *NS_DNS* for *dwNameSpace*. Finally, *lpafProtocols* is set to an array of two *AFPROTOCOLS* structures, which defines the TCP/IP and UDP/IP protocols as those that our query is interested in.

Once you establish the query, you can call *WSALookupServiceNext* to return data.

```
char          buff[sizeof(WSAQUERYSET) + sizeof(HOSTENT) + 2048];
DWORD         dwLength = sizeof(WSAQUERYSET) + sizeof(HOSTENT) + 2048;
WSAQUERYSET *pqs;
HOSTENT      *hostent;

pqs = (WSAQUERYSET *)buff;
pqs->dwSize = sizeof(WSAQUERYSET);
ret = WSALookupServiceNext(hQuery, 0, &dwLength, pqs);
if (ret == SOCKET_ERROR)
    // Error
WSALookupServiceEnd(hQuery);

hostent = (HOSTENT *) pqs->lpBlob->pBlobData;
```

Because a DNS name space provider returns the host information as a BLOB, you need to supply a sufficiently large buffer. Use a buffer equal in size to a *WSAQUERYSET* structure, plus a *HOSTENT* structure, plus 2048 bytes for good measure. Again, if this were insufficient, the call would fail with *WSAEFAULT*. In a DNS query, the host information is returned within the *HOSTENT* structure, even if a host name is associated with multiple IP addresses. That is why you don't need to call *WSALookupServiceNext* multiple times.

Now comes the tricky part—decoding the *BLOB* structure that the query returns. From Chapter 3, you know the *HOSTENT* structure is defined as

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

When the *HOSTENT* structure is returned as the BLOB data, the pointers within the structure are actually offsets into memory where the data lies. The offsets are from the start of the BLOB data, requiring you to fix up the pointers to reference the absolute memory location before you can access the data. Figure 8-1 shows the *HOSTENT* structure and memory layout returned. The DNS query is performed on the host name *riven*, which has a single IP address and no aliases. Each field in the structure has the offset value. To correct this so the fields reference the right location, you need to add the offset value to the address of the head of the *HOSTENT* structure. This needs to be performed on the *h_name*, *h_aliases*, and *h_addr_list* fields. In addition, the *h_aliases* and *h_addr_list* fields are an array of pointers. Once you obtain the correct pointer to the array of pointers, each 32-bit field in the references location is made up of offsets. If you take a look at the *h_addr_list* field in Figure 8-1, you'll see that the initial offset is 16 bytes, which references the byte after the end of the *HOSTENT* structure. This is the array of pointers to the 4-byte IP address. However, the first pointer in the array is an offset of 28 bytes. To reference the correct location, take the address of the *HOSTENT* structure and add 28 bytes, which points to a 4-byte location with the data 0x9D36B9BA, which is the IP address 157.54.185.186. You then take the 4 bytes after the entry with the offset of 28 bytes, which is 0. This signifies the end of the array of pointers. If multiple IP addresses were associated with this host name, another offset would be present and you would fix the pointer exactly as in the first case. The same procedure is done to fix the *h_aliases* pointer and the array of pointers it references. In this example, there are no aliases for our host. The first entry in the array is 0, which indicates that you don't have to do any further work for that field. The last field is the *h_name* field, which is easy to correct; simply add the offset to the address of the *HOSTENT* structure and it points to the start of a null-terminated string.

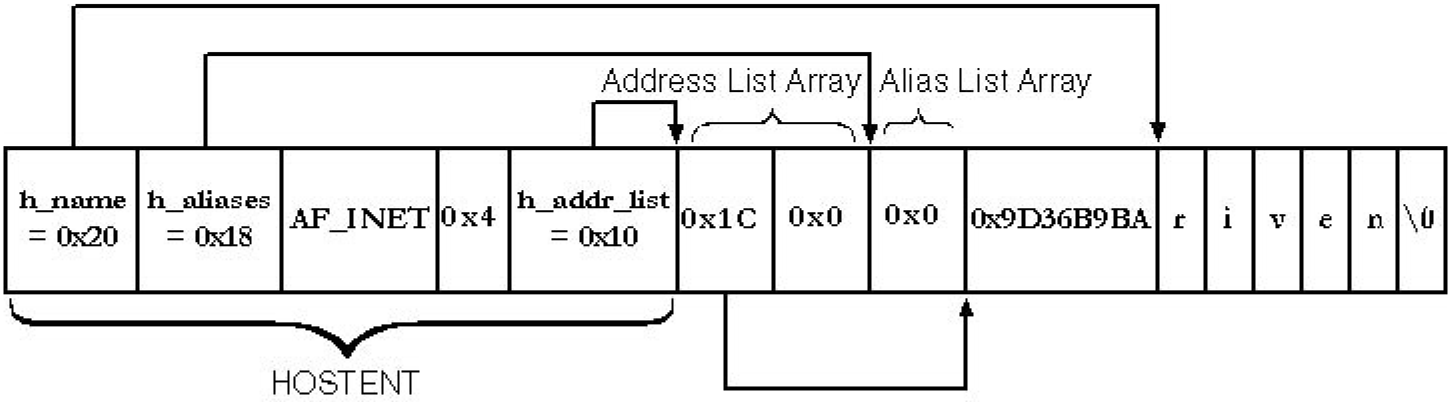


Figure 8-1 *HOSTENT BLOB format*

The code needed to fix these offsets into real addresses is simple, although quite a bit of pointer arithmetic is involved. To fix the *h_name* field, a simple offset adjustment such as the following will do:

```
hostent->h_name = (PCHAR)((DWORD_PTR)hostent->h_name +
(PCHAR)hostent);
```

To fix the array of pointers, as in the *h_aliases* and *h_addr_list* fields, requires a bit more code, but only to traverse the array and fix the references until a null entry is hit. The code looks like this:

```
PCHAR  *addr;

if (hostent->h_aliases)
{
    addr = hostent->h_aliases =
    (PCHAR)((DWORD_PTR)hostent->h_aliases + (PCHAR)hostent);
   while (addr)
   {
        addr = (PCHAR)((DWORD_PTR)addr + (PCHAR *)hostent);
        addr++;
    }
}
```

The code simply steps through each array entry and adds the starting address of the *HOSTENT* structure to the given offset, which becomes the value for that entry. Of course, once you hit an array entry whose value is 0, you stop. The same process needs to be applied to the *h_addr_list* field as well. Once the offsets are fixed, you can use the *HOSTENT* structure as you normally would.

Querying NLA

Windows XP offers a new name service called NLA which can help an application identify network names and connection properties of all networks connected to a machine at any given time or when a change has been made to the available network interfaces. For example, you might be developing an application that must connect to the Internet; NLA can determine if or when an Internet connection path becomes available. This is especially useful in the world of mobile computing, where network interfaces come and go depending on the physical location of a computer, such as a laptop used at home and work. Winsock allows you to query the NLA service through the name space provider interface. Retrieving network name information requires calling *WSALookupServiceBegin*, *WSALookupServiceNext*, and *WSALookupServiceEnd*.

To prepare for NLA queries, call *WSALookupServiceBegin* with the following restrictive *WSAQUERYSET* parameters and guidelines. The *dwSize* field must be set to the size of a *WSAQUERYSET* structure, *dwNameSpace* must be set to *NS_NLA*, and *lpServiceClassId* must be set to the GUID define *NLA_SERVICE_CLASS_GUID*. The remaining fields—except *lpszServiceInstanceName*—are ignored. You can optionally set *lpszServiceInstanceName* to restrict the query to a particular network name. For example, suppose you have two networks available: a home network and an Internet connection network by an ISP. Let's assume your ISP names your Internet connection “MyDSLProvider”. You can limit the NLA query to a single network by specifying the network name like “MyDSLProvider”. The following code fragment demonstrates how to prepare for a query by setting up a restrictive *WSAQUERYSET* structure from our discussion so far.

```
WSAQUERYSET qs;
GUID NLANameSpaceGUID = NLA_SERVICE_CLASS_GUID;
// Required fields to begin a query
qs.dwSize = sizeof(WSAQUERYSET);
qs.dwNameSpace = NS_NLA;
qs.lpServiceClassId = &NLANameSpaceGUID;
// Optional field to restrict the query
qs.lpszServiceInstanceName = "MyDSLProvider";
```

Once you have the restrictive *WSAQUERYSET* established, you have to decide how to control the query from the NLA service with the correct control flags for *WSALookupServiceBegin*. Table 8-8 describes the control flags that are used specifically in NLA queries. If you use either *LUP_RETURN_NAME* or *LUP_RETURN_COMMENT* exclusively you will receive only a unique list of network names. If you also include *LUP_RETURN_BLOB*, you will receive both network names and network characteristics for each network identified. The *LUP_DEEP* flag allows you to receive the maximum amount of information available; however, the queries will take longer to complete. The most common way to start the query is to simply OR the *LUP_RETURN_ALL* and the *LUP_DEEP* flags as follows:

```
LPHANDLE hNLA;
WSALookupServiceBegin(&qs, LUP_RETURN_ALL | LUP_DEEP, &hNLA)
```

Once you successfully retrieve a lookup handle *lphLookup* from *WSALookup-ServiceBegin*, you are ready to begin your NLA queries for available network names.

Table 8-8 *NLA Specific Control Flags*

Flag	Meaning
LUP_RETURN_ALL	Equates to using *LUP_RETURN_NAME*, *LUP_RETURN_COMMENT*, and *LUP_RETURN_BLOB*
LUP_RETURN_NAME	Retrieves the network name in the *WSAQUERYSET* field *lpszServiceInstanceName*
LUP_RETURN_COMMENT	Retrieves a friendly network name in the *WSAQUERYSET* field *lpszComment*
LUP_RETURN_BLOB	Retrieves network characteristic information, such as connection speed, adapter identification, and Internet connectivity.
LUP_DEEP	Tells NLA to query for all characteristics for each network available
LUP_FLUSHPREVIOUS	Used only on calls to *WSALookupServiceNext* to skip over network information that you can't handle/allocate memory for

Because the NLA name space provider is implemented as a service, it is designed to inform your application any time a change has occurred to the available network names. When you first call *WSALookupServiceBegin*, you can immediately query current network information using *WSALookupServiceNext* in which each call will return a network name until the function fails with error *WSA_E_NO_MORE*.

Depending on the control flags established in *WSALookupServiceBegin*, if you pass *LUP_RETURN_BLOB*, you will need to pass a sufficient-sized buffer for the *WSAQUERYSET* passed to *WSALookupServiceNext*; otherwise the function will fail with error *WSAEFAULT*. *LUP_RETURN_BLOB* can allow the query to return with an array of *NLA_BLOB* data structures containing specific network information such as speed and connectivity settings for each network returned from *WSALookupServiceNext*.

An *NLA_BLOB* can consist of an array of zero or more *NLA_BLOB* data structures and is defined as

```
typedef struct _NLA_BLOB {
   struct {
       NLA_BLOB_DATA_TYPE type;
       DWORD dwSize;
       DWORD nextOffset;
   } header;
   union {
       // header.type -> NLA_RAW_DATA
       CHAR rawData[1];
       // header.type -> NLA_INTERFACE
       struct {
           DWORD dwType;
           DWORD dwSpeed;
           CHAR adapterName[1];
       } interfaceData;
       // header.type -> NLA_802_1X_LOCATION
       struct {
            CHAR information[1];
        } locationData;

        // header.type -> NLA_CONNECTIVITY
        struct {
            NLA_CONNECTIVITY_TYPE type;
            NLA_INTERNET internet;
        } connectivity;

        // header.type -> NLA_ICS
        struct {
            struct {
                DWORD speed;
                DWORD type;
                DWORD state;
                WCHAR machineName[256];
                WCHAR sharedAdapterName[256];
            } remote;
        } ICS;

    } data;

} NLA_BLOB, *PNLA_BLOB, * FAR LPNLA_BLOB;
```

After successfully calling *WSALookupServiceNext*, you should first check if the *WSAQUERYSET lpBlob* field is not *NULL*. If it isn't, then you can begin looking at the *NLA_BLOB* structure returned from the call. As seen in the code above, an *NLA_BLOB* has two portions: header and data. The header section describes which NLA type the data portion of the *BLOB* represents and if there are any more *NLA_BLOBs* (through the *nextOffset* field) to be processed on the network name. Your application should cycle through all the data BLOBs identified by the *nextOffset* field in every *NLA_BLOB* until the field equals zero. Each *NLA_BLOB* found can have one of the following data types, which are defined in Table 8-9.

Table 8-9 *NLA Data Types*

NLA_BLOB Data Types	Corresponding *NLA_BLOB* Data Section
NLA_802_1X_LOCATION	Wireless network information stored in the *locationData* structure
NLA_CONNECTIVITY	Basic network connectivity information stored in *connectivity* structure—there are two fields: type and Internet, which refer to *NLA_CONNECTIVITY_TYPE*, *NLA_INTERNET* enumeration types, respectively—see Tables 8-10 and 8-11 for descriptions of these types.
NLA_ICS	An internal network sharing an Internet connection in the *ICS* structure
NLA_INTERFACE	General network information, such as media type, connection speed, and adapter name, stored in the interfaceData structure. Possible media types are defined in Ipifcons.h in the media types section.
NLA_RAW_DATA	Refers to the rawData field and is not used.

Table 8-10 *NLA Connectivity Types*

NLA_CONNECTIVITY_TYPE	Meaning
NLA_NETWORK_AD_HOC	Represents a private network not connected to any other network
NLA_NETWORK_MANAGED	Represents a network that is managed with a domain controller

NLA_NETWORK_UNKNOWN	Represents the network that is the private (non-Internet) side of an ICS connection
NLA_NETWORK_UNMANAGED	The service cannot determine the connection characteristics

Table 8-11 *NLA Internet Types*

NLA_INTERNET **Data Types**	**Meaning**
NLA_INTERNET_NO	Indicates that a path to the Internet is not available
NLA_INTERNET_UNKNOWN	Indicates that the service could not determine if there is a path to the Internet
NLA_INTERNET_YES	Indicates a path to the Internet is available

As you probably can tell, the *NLA_INTERFACE* and *NLA_CONNECTIVITY NLA_BLOB* data types offer some of the most useful information about the available networks. For example, if you receive an *NLA_CONNECTIVITY* data *BLOB*, you will be able to determine if you have a connection to the Internet. The following code fragment demonstrates how to get each network name by calling *WSALookupServiceNext* and cycle through the *NLA_BLOB* data found in each name.

```
char           buff[16384];
DWORD          BufferSize;
while (1)
{
    memset(qs, 0, sizeof(*qs));

    BufferSize = sizeof(buff);
    if (WSALookupServiceNext(hNLA, LUP_RETURN_ALL,
                             &BufferSize, qs) == SOCKET_ERROR)
    {
        int Err = WSAGetLastError();

        if (Err == WSA_E_NO_MORE)
        {
            // There is no more data. Stop asking.
            //
            break;
        }

        printf("WSALookupServiceNext failed with error %d\n",
            WSAGetLastError());

        WSALookupServiceEnd(hNLA);
        return;
    }

    printf("\nNetwork Name: %s\n", qs->lpszServiceInstanceName);
    printf("Network Friendly Name: %s\n", qs->lpszComment);
    if (qs->lpBlob != NULL)
    {
        //
        // Cycle through BLOB data list
        //
        DWORD     Offset = 0;
        PNLA_BLOB pNLA;
```

```
do
{
    pNLA = (PNLA_BLOB) &(qs->lpBlob->pBlobData[Offset]);

    switch (pNLA->header.type)
    {
        case NLA_RAW_DATA:
            printf("\tNLA Data Type: NLA_RAW_DATA\n");
            break;
        case NLA_INTERFACE:
            printf("\tNLA Data Type: NLA_INTERFACE\n");
            printf("\t\tType: %d\n",
                pNLA->data.interfaceData.dwType);
            printf("\t\tSpeed: %d\n",
                pNLA->data.interfaceData.dwSpeed);
            printf("\t\tAdapter Name: %s\n",
                pNLA->data.interfaceData.adapterName);
            break;
        case NLA_802_1X_LOCATION:
            printf("\tNLA Data Type: NLA_802_1X_LOCATION\n");
            printf("\t\tInformation: %s\n",
                pNLA->data.locationData.information);
            break;
        case NLA_CONNECTIVITY:
            printf("\tNLA Data Type: NLA_CONNECTIVITY\n");
            switch(pNLA->data.connectivity.type)
            {
                case NLA_NETWORK_AD_HOC:
                    printf("\t\tNetwork Type: AD HOC\n");
                    break;
                case NLA_NETWORK_MANAGED:
                    printf("\t\tNetwork Type: Managed\n");
                   break;
               case NLA_NETWORK_UNMANAGED:
                   printf("\t\tNetwork Type: Unmanaged\n");
                   break;
               case NLA_NETWORK_UNKNOWN:
                   printf("\t\tNetwork Type: Unknown\n");
           }
           switch(pNLA->data.connectivity.internet)
           {
               case NLA_INTERNET_NO:
                   printf("\t\tInternet connectivity: No\n");
                   break;
               case NLA_INTERNET_YES:
                   printf("\t\tInternet connectivity: Yes\n");
                   break;
               case NLA_INTERNET_UNKNOWN:
                   printf("\t\tInternet connectivity:
                        Unknown\n");
                    break;
            }
            break;
        case NLA_ICS:
            printf("\tNLA Data Type: NLA_ICS\n");
            printf("\t\tSpeed: %d\n",
                 pNLA->data.ICS.remote.speed);
```

```
                    printf("\t\tType: %d\n",
                         pNLA->data.ICS.remote.type);
                    printf("\t\tState: %d\n",
                         pNLA->data.ICS.remote.state);
                    printf("\t\tMachine Name: %S\n",
                        pNLA->data.ICS.remote.machineName);
                    printf("\t\tShared Adapter Name: %S\n",
                      pNLA->data.ICS.remote.sharedAdapterName);
                 break;

              default:
                 printf("\tNLA Data Type: Unknown to this program\n");
                 break;
          }
          Offset = pNLA->header.nextOffset;
      }
      while (Offset != 0);
  }
}
```

Once you have exhausted all the names and *NLA_BLOB* data returned from *WSALookupServiceNext*, the NLA service offers the capability to inform your application of changes made to these interfaces through the *WSANSPIoctrl* API. *WSANSPIoctrl* takes the lookup handle that *WSALookupServiceBegin* returned and associates a completion method for your application to receive notification of changes. *WSANSPIoctl* is defined as

```
int WSAAPI WSANSPIoctl(
  HANDLE hLookup,
  DWORD dwControlCode,
  LPVOID lpvInBuffer,
  DWORD cbInBuffer,
  LPVOID lpvOutBuffer,
  DWORD cbOutBuffer,
  LPDWORD lpcbBytesReturned,
  LPWSACOMPLETION lpCompletion
);
```

The *hLookup* parameter should be set to the handle that was originally returned from *WSALookupServiceBegin*. The *dwControlCode* should be set to *SIO_NSP_NOTIFY_CHANGE*. NLA does not use *lpvInBuffer* and *lpvOutBuffer* and they should be NULL. The *cbInBuffer* and *cbOutBuffer* are also not used and should be set to zero. The *lpcbBytesReturned* is not useful but you have to supply a buffer. The *lpCompletion* field allows you to specify an asynchronous wait method as described in the Overlapped I/O model of Chapter 5. The following code fragment demonstrates how to set up event-based notification for waiting on network changes.

```
WSAOverlap.hEvent = Event;
WSAComplete.Type = NSP_NOTIFY_EVENT;
WSAComplete.Parameters.Event.lpOverlapped = &WSAOverlap;


if (WSANSPIoctl(hNLA, SIO_NSP_NOTIFY_CHANGE, NULL, 0, NULL, 0,
    &BytesReturned, &WSAComplete) == SOCKET_ERROR)
{
    int Ret = WSAGetLastError();
    if (Ret != WSA_IO_PENDING)
    {
        printf("WSANSPIoctrl failed with error %d\n", Ret);
        return;
    }
}

if (WSAWaitForMultipleEvents(1, &Event, TRUE, WSA_INFINITE, FALSE) ==
    WSA_WAIT_FAILED)
{
    printf("WSAWaitForMultipleEvents failed with error %d\n",
        WSAGetLastError());
    return;
}

WSAResetEvent(Event);

// Query for all current network names available using the same
// lookup handle returned originally through WSALookupServiceBegin
```

> On the companion CD, you will find a sample named NLAQUERY.CPP that demonstrate
> how to query for NLA names from our discussion so far.

Finally, NLA will allow you to associate additional comment information with each network name returned from a query using the *WSASetService* API. All you have to do is fill in a *WSAQUERYSET* structure with a valid name in the *lpszServiceInstanceName* field and provide a new friendly name in the *lpszComment* field. Once the fields are set, call *WSASetService* with the first parameter (*lpqsRegInfo*) set to a pointer to your *WSAQUERYSET* structure, the second parameter (*essOperation*) set to the flag *RNRSERVICE_REGISTER*, and the third parameter (*dwControlFlags*) set to the value *NLA_FRIENDLY_NAME*.

Conclusion

The registration and name resolution (RNR) functions might seem overly complicated, but they offer great flexibility in writing client/server applications. The real limitation of name registration lies with the name space. It's rather amazing that even with the popularity of TCP/IP, the only name resolution method available is been DNS, which is not very flexible. With the Windows 2000, Windows XP, and Windows NT domain spaces, a persistent, protocol-independent method of name resolution is available, offering the necessary flexibility to write robust applications. In addition, other name spaces (such as SAP) are available for IPX/SPX-based applications, offering many of the same capabilities of the Windows NT domain space (except for protocol independence).

Chapter 9

Multicasting

Multicasting technology (or point-to-multipoint as it's sometimes called) allows data to be sent from one group member and then replicated to many others without creating a network traffic nightmare. This technology was developed as an alternative to broadcasting, which can negatively impact network bandwidth if used extensively. Multicast data is replicated to a network only if processes running on workstations in that network are interested in that data. Not all protocols support the notion of multicasting—on Windows platforms, only two protocols are multicast capable: IP and ATM. IP multicasting includes both IPv4 and IPv6. In addition, there are different types of IP multicasting. For example, Windows XP supports source multicasting for IPv4 as well as a reliable multicast provider that uses IP multicasting but adds reliability and session-oriented semantics. This chapter presents the information necessary to understand multicasting in general as well as how multicasting applies specifically to these protocols.

First, we will cover the basic semantics of multipoint networking, which includes the various possible types of multicasting as well as basic characteristics of IP and ATM multicasting. Then we will present the specifics of IP and ATM in their own sections. In each section, we will discuss how the Winsock API is used to access each protocol's multicast features.

All Windows platforms support one or more multicasting protocols. IPv4 multicasting is available on all Windows platforms (Windows CE requires version 2.1 or greater). IPv6 multicasting is available in Windows XP as well as in Windows 2000 with the IPv6 technology preview and in Windows NT 4.0 with the Microsoft Research IPv6 stack. Reliable multicasting is available in Windows XP if the Microsoft Message Queue (MSMQ) component is installed. Finally, native ATM multicasting is available in Windows 98, Windows Me, Windows 2000, and Windows XP.

Multicast Semantics

Multicasting has two important properties: the control plane and the data plane. The ***control plane*** defines the way that group membership is organized. The ***data plane*** refers to the way that data is propagated among the members. Either one of these properties can be rooted or non-rooted. In a ***rooted*** control plane, there is a special member of the multicast group known as the ***c_root***. Each of the remaining group members is known as a ***c_leaf***. In most cases, the c_root establishes the multipoint group by initiating connections to any number of c_leafs. In some cases, a c_leaf might request membership to a given multipoint group at a later time. Note that there can be only one root node for a given group. The ATM protocol is an example of a rooted control plane.

A ***non-rooted*** control plane allows anyone to join a group without exception. In this situation, all group members are c_leaf nodes. Each member has the power to join a multipoint group. You can impose your own group membership scheme in a non-rooted control plane (this will in effect make one node a c_root) by implementing your own group membership protocol. However, your group membership scheme is still built upon a non-rooted control plane. All forms of IP multicasting and reliable multicasting are examples of a non-rooted control plane. Figure 9-1 illustrates the difference between rooted and non-rooted control planes. In the rooted control plane on the left side of the figure, the c_root must explicitly ask each c_leaf to join the group, while in the non-rooted scheme on the right anyone can join the group.

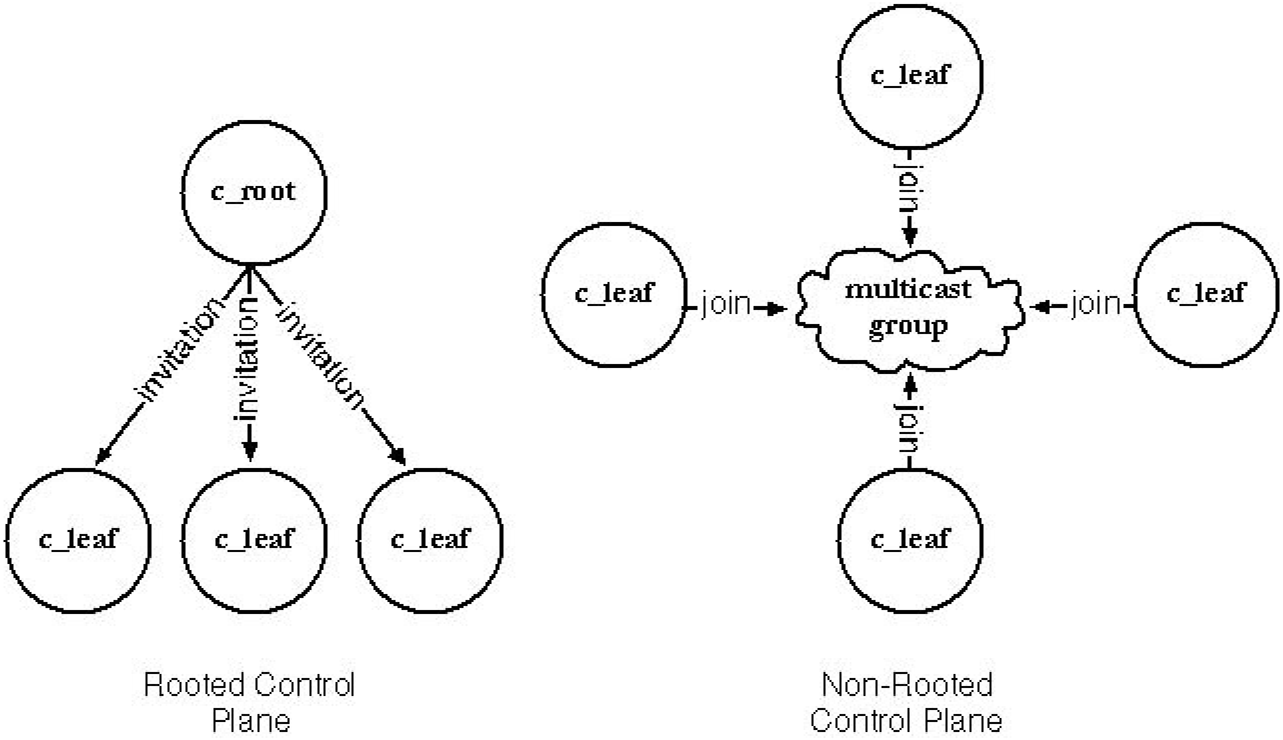
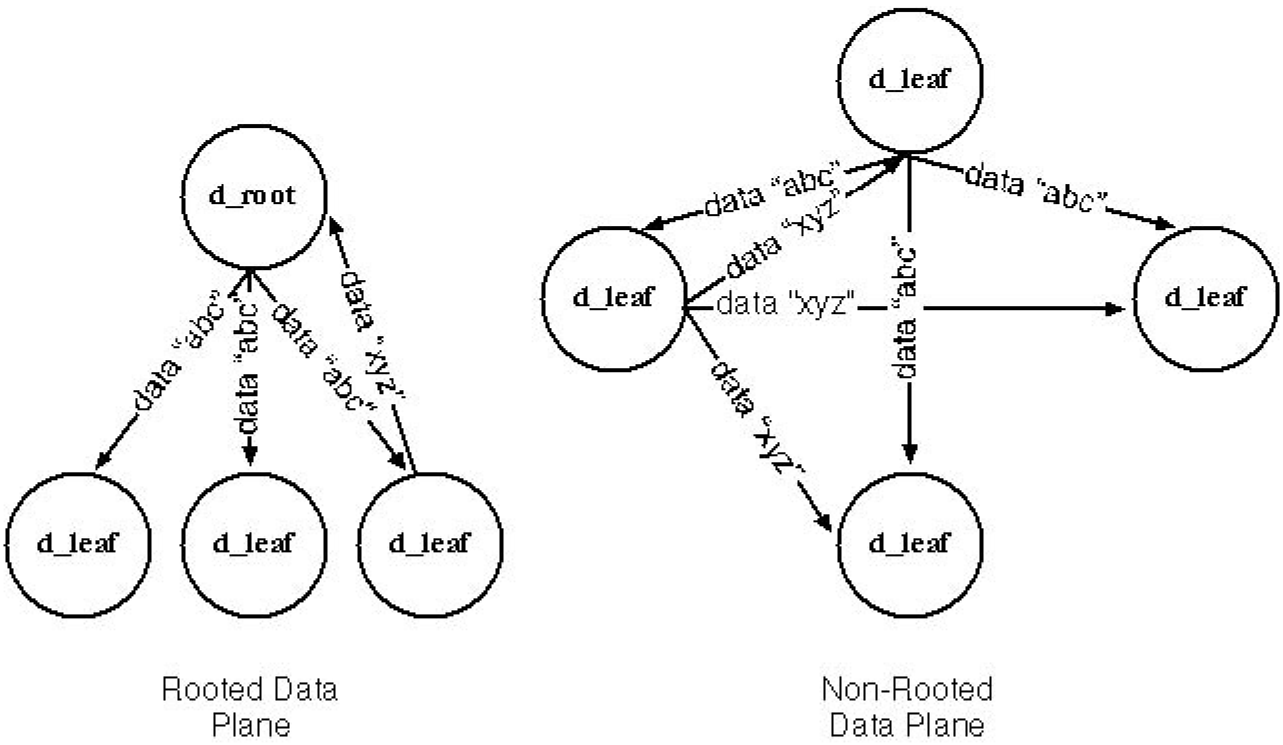


Figure 9-1 *Rooted and non-rooted control planes*

The data plane also can be rooted or non-rooted. A rooted data plane has a participant called the ***d_root***. The transfer of data occurs only between d_root and all other members of the multipoint session, who are each referred to as a ***d_leaf***. The traffic can be either unidirectional or bidirectional, but a rooted data plane implies that data sent from one d_leaf will be received by only the d_root, and data sent from the d_root will be received by each d_leaf. ATM and reliable multicasting are examples of a rooted data plane. Figure 9-2 illustrates the difference between rooted and non-rooted data planes. In the rooted data plane on the left of the figure, data ***abc*** from the d_root is propagated to every d_leaf. Data *xyz* sent from a d_leaf is received by only the d_root. This contrasts with the non-rooted example on the right, in which data *abc* and *xyz* are propagated to every member, no matter who sent the data.



Rooted Data Plane

Non-Rooted Data Plane

Figure 9-2 *Rooted and non-rooted data planes*

Finally, in a non-rooted data plane all group members can send data to all other members of the group. A block of data sent from a group member is delivered to all other members, and all recipients can send data back. There are no restrictions on who can receive or send data. Again, IP multicasting is non-rooted in the data plane.

So we see that ATM multicasting is rooted in the control and data planes, and IP multicasting is non-rooted in both planes. Reliable multicasting is an example of a non-rooted control plane but a rooted data plane.

Finding Multicast Properties

In Chapter 2, we discussed how to enumerate protocol entries and determine their properties. All of the pertinent multipoint information about a protocol is also available from the protocol's entry in the catalog. The *dwServiceFlags1* entry in the *WSAPROTOCOL_INFO* structure returned by *WSAEnumProtocols* contains several bits we're interested in. If the *XP1_SUPPORT_MULTIPOINT* bit is set, the protocol entry supports multicasting. Then, if the *XP1_MULTIPOINT_CONTROL_PLANE* bit is set, the protocol supports a rooted control plane; otherwise, it is non-rooted. If the *XP1_MULTIPOINT_DATA_PLANE* bit is set, the protocol supports a rooted data plane. Likewise, if the bit is 0, the protocol supports only a non-rooted data plane.

IP Multicasting

IP multicasting relies on a special group of addresses known as multicast addresses. It is this group address that names a given group. For example, if five nodes all want to communicate with one another via IP multicast, they all join the same group address. Once they are joined, any data that one node sends is replicated to every member of the group, including the node that sent the data. A multicast IPv4 address is a class D IP address in the range 224.0.0.0 through 239.255.255.255 and IPv6 multicast addresses begin with the prefix FF (1111 1111), as we discussed in Chapter 3. A number of these addresses are reserved for special purposes. For a comprehensive list of reserved addresses, take a look at RFCs 1700 and 2375. The IANA maintains this list. Table 9-1 lists a few of the IPv4 addresses currently marked as reserved. Actually, you can use any address except for the first three reserved multicast addresses because they are used by routers on the network. Refer to RFC 1700 for the exact multicast address assignments.

Table 9-1 *Reserved IPv4 Multicast Addresses*

Multicast Address	Use
224.0.0.0	Base address (reserved)
224.0.0.1	All nodes on this subnet
224.0.0.2	All routers on this subnet
224.0.1.1	Network time protocol
224.0.0.9	RIP version 2 group address
224.0.1.24	WINS server group address

IPv6 multicasting also sets aside certain multicast addresses for specific purposes, some of which are listed in Table 9-2. One difference between IPv4- and IPv6-reserved multicast addresses is that IPv6 offers different addresses depending on the desired scope, such as link-local and site-local.

Table 9-2 *IPv6 Multicast Addresses*

Scope	Multicast Address	Use
Node-local	FF01::1	All nodes address
Node-local	FF01::2	All routers address
Link-local	FF02::1	All nodes address
Link-local	FF02::2	All routers address
Link-local	FF02::9	RIP routers
Site-local	FF05::2	All routers address
Site-local	FF05::1:3	All DHCP servers
Site-local	FF05::1:4	All DHCP relays

Because multicasting had not been envisioned when TCP/IP was developed, a number of accommodations had to be made to allow IP to support it. For example, we have already discovered that IP requires a set of special addresses to be set aside for multicast traffic. In addition, for IPv4, a special protocol was introduced to manage multicast clients and their membership in a group. Imagine if two workstations on separate subnets want to join a single multicast group. How is this implemented over IP? You can't simply broadcast the data to the multicast address everywhere because the network would become flooded with broadcast data in no time. IGMP was developed to signal routers that a machine on the network is interested in data destined for a given group. The latest revision of IGMP (version 3) adds support for limiting what sources data is accepted from. Note that the reliable multicast provider still relies on IGMP to communicate group membership to network elements.

For IPv6, the ICMPv6 protocol rolls up the various support protocols, such as ICMP, ARP, and IGMP, into one. Multicast membership is managed by MLD messages, which are a type of message sent over the ICMPv6 protocol.

Support Protocols

Multicasting hosts use IGMP and MLD to notify routers that a computer on the router's subnet wants to join a particular multicast group. IGMP is the backbone of IPv4 multicasting and MLD is its counterpart for IPv6. For multicasting to work correctly, all routers between two multicasting nodes must support the appropriate multicast support protocol. For example, in the case of IPv4, if machines A and B join the multicast group 224.1.2.3, and there are three routers between the two, all three routers must be IGMP-enabled for successful communication to occur. Any non-multicast-enabled router simply drops received multicast data. When an application joins a multicast group, a “join” (or report) message is sent to the all-routers address on the interface the group was joined. This command notifies the router that it has clients interested in a particular multicast address. Thus, if the router receives data destined for that multicast address, it forwards it to the network with the multicast client.

In addition, when an endpoint joins a multicast group, it specifies a TTL parameter that indicates how many routers the endpoint's multicast application is willing to traverse to send and receive data. For example, if you write an IP multicast application that joins group X with a TTL of 2, a join command is sent to the all-routers group on the local subnet. The routers on that subnet pick up the command, indicating that it should forward multicast data destined for that address. The router decrements the TTL by 1 and passes the join command on to its neighboring networks. The routers on those networks do the same upon receipt of the command. At this point, those routers decrement the TTL again, which now makes the TTL value 0, and the command is no longer propagated. Because of this, TTL limits how far multicast data will be replicated.

Once a router has one or more multicast groups registered by workstations, it periodically sends a “group query” message to the all-hosts address for each multicast address that a join command notified it of. If clients on that network are still using that multicast address, they respond with another message so that the router knows to keep forwarding data related to that address; otherwise, the router stops forwarding any data for that address. Also, both IGMP (version 2 and greater) and MLD support the notion of a host explicitly leaving a group. That is, each interface maintains a reference count of how many applications are joined to a particular multicast group. When the count goes to zero, a leave (or “done”) message is sent. This notifies the routers to stop forwarding data for that group. Note that for IGMPv1, there is no explicit leave message, so the router will continue forwarding data even after the interested application(s) have exited, which can have undesired results. Only when the IGMPv1 router sends a group query will it discover that no one is listening to that particular group.

Windows XP introduces support for IGMP version 3 for IPv4 multicasting, which allows applications to join an IPv4 multicast group and list one or more sources from which to accept or deny data from. If group X is joined, which specifies sources A and B as the only valid sources, then only data originating from A or B will be propagated to the application. Likewise, if group X was joined and sources A and B were ***excluded***, then data sent to the multicast group from everyone except A and B will be propagated to the application. For IGMPv3 to work properly, the routers on the network must also support IGMPv3. If they don't, there is no real gain because the routers will propagate all data for the joined multicast groups, not just from the source list specified. If a Windows XP host is on a network that is not IGMPv3-enabled, it will fall back to the version of IGMP present.

Windows 98, Windows Me, and Windows 2000 natively support IGMP version 2. For Windows 95, the latest Winsock 2 update also includes IGMP version 2. In Windows NT 4.0, Service Pack 4 includes support for IGMP version 2. Previous Service Packs and the base OS supported only version 1. Windows XP supports IGMPv3. If you want to read the complete specifications on IGMP version 1 or version 2, consult RFC 1112 or RFC 2236, respectively. Currently, IGMP version 3 is an IETF draft: *draft-ietf-idmr-igmp-v3-07.txt*.

Multicasting with *Setsockopt*

Originally, the only way to join or leave a multicast group was via the *setsockopt* API. Winsock 2 introduces a protocol-independent method of multicasting with the *WSAJoinLeaf* API (discussed in the next section), but as we will soon see, the *setsockopt* method is much more flexible even though it is more closely tied to the protocol being used.

IPv4

There are two socket options that control joining and leaving groups: *IP_ADD_MEMBERSHIP* and *IP_DROP_MEMBERSHIP*. The socket option level is *IPPROTO_IP*. The input parameter is a *struct ip_mreq* structure, which is defined as

```
struct ip_mreq {
   struct in_addr imr_multiaddr;   /* IP multicast address of group */
   struct in_addr imr_interface;   /* local IP address of interface */
};
```

The *imr_multiaddr* field is the 32-bit IPv4 address of the multicast group in network-byte order and *imr_interface* is the 32-bit IPv4 address of the local interface on which to join the multicast group (also specified in network-byte order). The following code snippet illustrates joining a multicast group.

```
SOCKET                          s;
```

```
SOCKADDR_IN    localif;
struct ip_mreq mreq;

s = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);

localif.sin_family = AF_INET;
localif.sin_port   = htons(5150);
localif.sin_addr.s_addr = htonl(INADDR_ANY);
bind(s, (SOCKADDR *)&localif, sizeof(localif));

mreq.imr_interface.s_addr = inet_addr("157.124.22.104");
mreq.imr_multiaddr.s_addr = inet_addr("234.5.6.7");

setsockopt(s, IPPROTO_IP, IP_ADD_MEMBERSHIP, (char *)&mreq,
sizeof(mreq));
```

Note that the socket should be bound to the wildcard address (INADDR_ANY) before joining the group. In this example, the socket is joined to the multicast group 234.5.6.7 on the local interface 157.124.22.104. Multiple groups may be joined on the same socket on the same or different interface.

Once one or more multicast groups are joined, the *IP_DROP_MEMBERSHIP* option is used to leave a particular group. Again, the *struct ip_mreq* structure is the input parameter. The local interface and multicast group to drop are the arguments of the structure. For example, given the code sample you just saw, the following code drops the multicast group previously joined:

```
// Join the group as shown above

mreq.imr_interface.s_addr = inet_addr("157.124.22.104");
mreq.imr_multiaddr.s_addr = inet_addr("234.5.6.7");

setsockopt(s, IPPROTO_IP, IP_DROP_MEMBERSHIP, (char *)&mreq,
sizeof(mreq));
```

Finally, if the application exits or the socket is closed, any multicast groups joined by that process or socket are cleaned up.

An IPv4 multicasting sample that uses *setsockopt* is provided on the companion CD in th directory IP-SETSOCKOPT.

IPv4 with Multicast Sourcing

IP source multicasting is available on systems that support the IGMPv3 protocol and allows a socket to join a multicast group on an interface while specifying a set of source addresses to accept data from. There are two possible modes in which a socket may join a group. The first is the *INCLUDE* mode, in which a socket joins a group specifying N number of valid source addresses to accept data from. The other mode is *EXCLUDE*, in which a socket joins a group specifying to accept data from anyone ***except*** the N source addresses listed. Depending on which mode is used, the socket options differ.

To join a multicast group while using the *INCLUDE* mode, the socket options are *IP_ADD_SOURCE_MEMBERSHIP* and *IP_DROP_SOURCE_MEMBERSHIP*. The first step is to add one or more sources. Both socket options take a *struct ip_mreq_source* structure, which is defined as

```
struct ip_mreq_source {
   struct in_addr imr_multiaddr;   /* IP multicast address of group */
   struct in_addr imr_sourceaddr;  /* IP address of source          */
   struct in_addr imr_interface;   /* local IP address of interface */
};
```

The *imr_multiaddr* and *imr_interface* fields are the same as in the *struct ip_mreq* structure. The new field *imr_sourceaddr* specifies the 32-bit IP address of the source to accept data from. If there are multiple valid sources, then the *IP_ADD_SOURCE_MEMBERSHIP* is called again with the same multicast address and interface with the other valid source. The following code sample joins a multicast group on a local interface with two valid sources:

```
SOCKET                s;
SOCKADDR_IN           localif;
struct ip_mreq_source mreqsrc;

s = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);

localif.sin_family = AF_INET;
localif.sin_port   = htons(5150);
localif.sin_addr.s_addr = htonl(INADDR_ANY);
bind(s, (SOCKADDR *)&localif, sizeof(localif));

mreqsrc.imr_interface.s_addr = inet_addr("157.124.22.104");
mreqsrc.imr_multiaddr.s_addr = inet_addr("234.5.6.7");
mreqsrc.imr_sourceaddr.s_addr = inet_addr("172.138.104.10");

setsockopt(s, IPPROTO_IP, IP_ADD_SOURCE_MEMBERSHIP,
                (char *)&mreqsrc, sizeof(mreqsrc));

mreqsrc.imr_sourceaddr.s_addr = inet_addr("172.141.87.101");

setsockopt(s, IPPROTO_IP, IP_ADD_SOURCE_MEMBERSHIP,
                (char *)&mreqsrc, sizeof(mreqsrc));
```

To remove a source from the *INCLUDE* set, the *IP_DROP_SOURCE_ MEMBERSHIP* is called with the multicast group, local interface, and source to be removed.

To join a multicast group that excludes one or more sources, the multicast group is joined with *IP_ADD_MEMBERSHIP*. Using *IP_ADD_MEMBERSHIP* to join a group is equivalent to joining a group in the *EXCLUDE* mode except that no one is excluded. Data sent to the joined group is accepted regardless of the source. Once the group is joined, then the *IP_BLOCK_SOURCE* option is called to exclude the given source. Again, the *struct ip_mreq_source* structure is the input parameter that specifies the source to block. The following example joins a group and then excludes a single source:

```
SOCKET                 s;
SOCKADDR_IN            localif;
struct ip_mreq         mreq;
struct ip_mreq_source mreqsrc;

s = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);

localif.sin_family = AF_INET;
localif.sin_port   = htons(5150);
localif.sin_addr.s_addr = htonl(INADDR_ANY);
bind(s, (SOCKADDR *)&localif, sizeof(localif));

// Join a group - the filter is EXCLUDE none
mreq.imr_interface.s_addr = inet_addr("157.124.22.104");
mreq.imr_multiaddr.s_addr = inet_addr("234.5.6.7");

setsockopt(s, IPPROTO_IP, IP_ADD_MEMBERSHIP, (char *)&mreq,
                sizeof(mreq));

mreqsrc.imr_interface = mreq.imr_interface;
mreqsrc.imr_multiaddr = mreq.imr_multiaddr;
mreqsrc.imr_sourceaddr.s_addr = inet_addr("172.138.104.10");

setsockopt(s, IPPROTO_IP, IP_BLOCK_SOURCE, (char *)&mreqsrc,
                sizeof(mreqsrc));
```

If after some point, the application wishes to accept data from a source previously blocked, it may remove that source from the exclude set by calling *setsockopt* with *IP_UNBLOCK_SOURCE*. A *struct ip_mreq_source* is the input parameter that specifies the source to accept data from.

An IPv4 source multicasting sample that uses *setsockopt* is provided on the companion CD in the directory IP-SOURCE.

IPv6

Multicasting with IPv6 is similar to IPv4 multicasting except that the socket options are named differently and take slightly different input parameters. The options for IPv6 are *IPV6_ADD_MEMBERSHIP* and *IPV6_DROP_MEMBERSHIP*. The option level is *IPPROTO_IPV6*. The structure that specifies the multicast group and interface is a *struct ipv6_mreq* that is defined as

```
typedef struct ipv6_mreq {
    struct in6_addr ipv6mr_multiaddr;  /* IPv6 multicast address */
    unsigned int    ipv6mr_interface;  /* Interface index */
} IPV6_MREQ;
```

The structure fields are equivalent to the IPv4 structure *struct ip_mreq* except that the local interface is the interface index instead of the full IPv6 address. The easiest way to find the interface index of a local IPv6 address is via the IP Helper API *GetAdaptersAddresses* covered in Chapter 16. Also, if the link-local address is used as the local interface, the scope-ID of that link is the interface index. The following code sample illustrates joining an IPv6 multicast group when the link-local address is given for the local interface:

```
SOCKET             s;
struct ipv6_mreq   mreq6;
struct addrinfo   *reslocal,
                  *resmulti,
                   hints;

s = socket(AF_INET6, SOCK_DGRAM, IPPROTO_UDP);

// Get the local wildcard address to bind to (i.e. "::")
memset(&hints, 0, sizeof(hints));
hints.ai_family = AF_INET6;
getaddrinfo(NULL, "5150", &hints, &reslocal);
bind(s, reslocal->ai_addr, reslocal->ai_addrlen);
freeaddrinfo(reslocal);

// Resolve the link-local interface
getaddrinfo("fe80::250:4ff:fe7c:7036%6", NULL, NULL, &reslocal);
// Resolve the multicast address
getaddrinfo("ff12::1", NULL, NULL, &resmulti);

// Join the multicast group
mreq6.ipv6mr_multiaddr =
        ((SOCKADDR_IN6 *)resmulti->ai_addr)->sin6_addr;
mreq6.ipv6mr_interface =
        ((SOCKADDR_IN6 *)reslocal->ai_addr)->sin6_scope_id;

setsockopt(s, IPPROTO_IPV6, IPV6_ADD_MEMBERSHIP,
                (char *)&mreq6, sizeof(mreq6));

freeaddrinfo(resmulti);
freeaddrinfo(reslocal);
```

As with IPv4, to leave the multicast group the same *struct ipv6_mreq* structure is passed into *setsockopt* but with the *IPV6_DROP_MEMBERSHIP* option.

An IPv6 multicasting sample that uses *setsockopt* is provided on the companion CD in th directory IP-SETSOCKOPT. This is the same sample as that for IPv4 multicasting because it will figure out which address family is being used from the addresses supplie on the command line (using the *getaddrinfo* API to make the sample IP version independent as discussed in Chapter 3).

Sending Multicast Data with IPv4

The previous sections have shown how to join and leave multicast groups to receive data sent to that group; however, an application need not join the group to send data to it. There is one issue to be aware of: multihomed computers. If an application creates a UDP socket and calls *sendto* with a destination address of “234.5.6.7”, which interface is the data sent on? Basically, the first interface listed in the routing table is the interface the data is sent on. To override this behavior, applications may use the *IP_MULTICAST_IF* socket option to specify the interface for outgoing data. The option value is simply the 32-bit IPv4 address of the local interface. The following code sample sets the outgoing interface on a socket.

```
SOCKET         s;
SOCKADDR_IN    dest;
ULONG          localif;
char           buf[1024];
int            buflen=1024;

s = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);

localif = inet_addr("157.124.22.104");
setsockopt(s, IPPROTO_IP, IP_MULTICAST_IF,
                (char *)&localif, sizeof(localif));

dest.sin_family = AF_INET;
dest.sin_port   = htons(5150);
dest.sin_addr.s_addr = inet_addr("234.5.6.7");

sendto(s, buf, buflen, 0, (SOCKADDR *)&dest, sizeof(dest));
```

The multicasting sample code on the companion CD also illustrates using this socket option.

Sending Multicast Data with IPv6

The same sending issue that applies to IPv4 is also present for IPv6 multicasting. The outgoing interface for data sent to an IPv6 multicast group is specified with the *IPV6_MULTICAST_IF* option. The input parameter is an integer that specifies the adapter interface index that outgoing data is sent on. Also, if *sendto* or *WSASendTo* is used for sending data to the multicast group, the *sin6_scope_id* of the *SOCKADDR_IN6* structure given for the *to* address parameter should be zero.

The multicasting sample code on the companion CD also illustrates using this socket option.

Multicasting with *WSAIoctl*

For IPv4 source multicasting, there is an ioctl that can specify one or more source addresses to include or exclude with a single call. This is *SIO_SET_MULTICAST_FILTER*. The input parameter is a *struct ip_msfilter* structure that is defined as

```
struct ip_msfilter {
   struct in_addr imsf_multiaddr;  /* IP multicast address of group */
   struct in_addr imsf_interface;  /* local IP address of interface */
   u_long         imsf_fmode;      /* filter mode - */
                                   /* INCLUDE or EXCLUDE */
   u_long         imsf_numsrc;    /* number of sources in src_list */
    struct in_addr imsf_slist[1];
};
```

The first two fields are self-explanatory. The third field, *imsf_fmode*, indicates whether the source addresses listed in the *imsf_slist* array are sources that should be included or excluded from the multicast group. To include the sources, the constant *MCAST_INCLUDE* is supplied; otherwise, to exclude these sources *MCAST_EXCLUDE* is used. *imsf_numsrc* indicates the number of sources supplied, and *imsf_slist* is the array of source addresses.

The following code sample illustrates using this ioctl:

```
SOCKET                            s;
char                              buf[1024];
struct ip_msfilter *msfilter;
DWORD                             bytes;
int                               filterlen;

s = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);

// bind the socket to INADDR_ANY
msfilter = (struct ip_msfilter *)buf;
msfilter->imsf_multiaddr.s_addr = inet_addr("234.5.6.7");
msfilter->imsf_interface.s_addr = inet_addr("157.124.22.104");
msfilter->imsf_fmode = MCAST_INCLUDE;
msfilter->imsf_numsrc = 3;
msfilter->imsf_slist[0].s_addr = inet_addr("172.138.104.10");
msfilter->imsf_slist[1].s_addr = inet_addr("172.138.104.11");
msfilter->imsf_slist[2].s_addr = inet_addr("172.138.104.12");

filterlen = sizeof(struct ip_msfilter) +
                ((msfilter->imsf_numsrc - 1) * sizeof(struct in_addr));

WSAIoctl(s, SIO_SET_MULTICAST_FILTER, (void *)msfilter, filterlen,
                NULL, 0, &bytes, NULL, NULL);
```

This sample joins a multicast group and specifies three valid sources to accept data from. To retrieve the multicast filter set, use the ioctl *SIO_GET_MULTICAST_ FILTER*. The input parameter must be a *struct ip_msfilter* structure that contains the local interface and multicast group on which to obtain the multicast state for. Upon success, a *struct ip_msfilter* structure is returned via the output parameter, which contains the full set of sources for the group and interface. Note that you must specify a large enough buffer to hold the filter plus all the sources.

An IPv4 source multicasting sample that uses *WSAIoctl* is provided on the companion C in the directory IP-SOURCE. This is the same sample that uses *setsockopt* for IP source multicasting. The ‘-f' flag indicates to use *SIO_SET_MULTICAST_FILTER* instead of th socket options.

Multicasting with *WSAJoinLeaf*

Winsock 2 introduces the *WSAJoinLeaf* API, which is designed to be protocol independent. The API is defined as

```
SOCKET WSAJoinLeaf(
    IN SOCKET s,
    IN const struct sockaddr FAR * name,
    IN int namelen,
    IN LPWSABUF lpCallerData,
    OUT LPWSABUF lpCalleeData,
    IN LPQOS lpSQOS,
    IN LPQOS lpGQOS,
    IN DWORD dwFlags
    );
```

In most respects, *WSAJoinLeaf* takes the same parameters as *WSAConnect*. The exceptions are that *name* indicates the multicast address to join and *dwFlags* indicates whether the socket will be sending or receiving data on the multicast socket. The valid flag values are *JL_SENDER_ONLY*, *JL_RECEIVER_ONLY*, and *JL_BOTH*.

For IP multicasting, a UDP socket must be created by using the *WSASocket* API, and the flags *WSA_FLAG_MULTIPOINT_C_LEAF* and *WSA_FLAG_MULTIPOINT_D_LEAF* must be specified. The socket should then be bound to the specific interface on which the group is to be joined. Then *WSAJoinLeaf* is invoked with the multicast address of the group to join. There are a couple of differences here. First, instead of binding to the wildcard address, you should bind to the specific local interface. Second, the outgoing (sending) interface does not have to be set. When the multicast group is joined, the outgoing interface is automatically set to the interface the socket is bound to. Finally, only a single multicast group can be joined on a socket. That is, unlike using *setsockopt*, *WSAJoinLeaf* can be invoked only once on a socket and the multicast group is joined until the socket is closed—there is no other way to leave the group.

An IP multicasting sample that uses *WSAJoinLeaf* is provided on the companion CD in the directory IP-WSAJOINLEAF. The sample works over IPv4 and IPv6.

Reliable Multicasting

The reliable multicasting provider is available on Windows XP when the Microsoft Message Queuing (MSMQ) component is installed. Reliable multicasting is a protocol built over IPv4 multicasting. Currently, it does not support IPv6. The reliable multicast implementation on Windows XP is based on the Pragmatic General Multicasting (PGM) specification. The protocol is NAK-based, meaning that all receivers of the data send notice to the sender only when it detects that a packet is missing. If the protocol was acknowledgment-based, like TCP, you would have cases in which hundreds or possibly thousands of receivers sent a packet to the sender indicating they successfully received a packet, which would flood the sender's network. This section provides only an overview of how the protocol is implemented so you can gain a basic understanding of how reliable multicasting is provided. For a complete specification of the PGM protocol, consult the IETF draft: *draft-speakman-pgm-spec-06.txt*.

The reliable multicast protocol works by the sender caching data sent to the receivers for a specified time period, which is known as the send window. This window is periodically moved forward, discarding the oldest data within the window to make room for new data the sender sends. There are three variables that make up the send window: the send rate, the size of the window (in bytes), and the size of the window in seconds (for example, how long the data remains in the window before being purged). The reliable multicast driver establishes certain default values for the window size that can be overridden (because the default values are rather low).

The protocol also contains a sequence number in the data packets so that if a receiver detects that it has missed a particular sequence, it may send a NAK to the sender, who will retransmit the missing data. Note that it is possible that a receiver NAKs for data that was recently purged from the send window. In this case, the receiver's “session” will be reset because of unrecoverable data loss—this is analogous to a TCP session being reset due to an ACK timeout or similar error.

In terms of the Winsock provider for reliable multicasting, there are a couple of important issues. First, there is a stream provider (*SOCK_STREAM*) as well as a reliable datagram provider (*SOCK_RDM*). The protocol value that is passed to the socket creation functions is *IPPROTO_RM*. The socket options specific to this protocol are contained in WSRM.H.

In the following sections we will look at how to create a reliable multicast sender as well as a receiver. A complete code sample is available on the companion CD in the directory IP-RM.

Reliable Sender

Establishing the reliable multicast sender is a simple process. The following list contains the necessary steps.

1. Create a reliable multicast socket.
2. Bind the socket to *INADDR_ANY*.

3. Set the outgoing interface using *RM_SET_SEND_IF*.
4. Connect the socket to the multicast group address that the data transmission is to take place on.

As with IP multicasting, it is necessary to set the outgoing interface in the case of a multihomed computer. Also note that no real connection is taking place between the sender and receiver. The connect that the sender called simply associates the destination multicast address with the socket. The following code example illustrates setting up a reliable multicast sender.

```
SOCKET                          s;
ULONG                           sendif;
SOCKADDR_IN                             localif,
                                multi;
char                            buf[1024];
int                             buflen=1024;

s = socket(AF_INET, SOCK_RDM, IPPROTO_RM);

// Bind to INADDR_ANY
localif.sin_family = AF_INET;
localif.sin_port   = htons(0);
localif.sin_addr.s_addr = htonl(INADDR_ANY);
bind(s, (SOCKADDR *)&localif, sizeof(localif));

// Set the outgoing interface
sendif = inet_addr("157.124.22.104");
setsockopt(s, IPPROTO_RM, RM_SET_SEND_IF, (char *)&sendif,
                sizeof(sendif));

// Connect the socket to the multicast destination
multi.sin_family = AF_INET;
multi.sin_port   = htons(5150);
multi.sin_addr.s_addr = inet_addr("234.5.6.7");
connect(s, (SOCKADDR *)&multi, sizeof(multi));

// Send the data
send(s, buf, buflen, 0);

// Close up the session
closesocket(s);
```

In this example, we create a socket of type *SOCK_RDM*. This is similar to *SOCK_DGRAM* because it provides message-oriented semantics but also implies reliability. Also, the local port supplied to the bind call is not significant. After the session is created, the sender may begin to send data.

Modifying the Window Size

By default, the reliable multicast driver sets up the default window size to buffer sent data. At some interval, the window is advanced by an increment, discarding the oldest data to make room for newer data. This window plays an important role in how reliable the data is. For example, a high data rate sender may receive many NAKs for lost data from receivers on a lower bandwidth or a congested network. As a result, if the send window is fairly small, those receivers may not be able to keep up. To prevent this, the sender may resize the send window to better suit the application's needs. This is done via the *RM_RATE_WINDOW_SIZE* socket option. The input parameter is a *RM_SEND_WINDOW* structure defined as

```
typedef struct _RM_SEND_WINDOW
{
    ULONG   RateKbitsPerSec;
    ULONG   WindowSizeInMSecs;
    ULONG   WindowSizeInBytes;
} RM_SEND_WINDOW;
```

The *RateKbitsPerSec* specifies the data rate that the sender will be sending at. By default, the send rate is 56 Kbps. *WindowSizeInMSecs* specifies how long (in milliseconds) the data is to remain in the window before being purged to make room for new data. The last field, *WindowSizeInBytes*, represents how many bytes can be stored in the window before old data has to be purged to make room for new data. When setting this option, the following equation must hold true: *WindowSizeInBytes* = (*RateKbitsPerSec/8*) x *WindowSizeInMSecs*. This option must be set before connecting the socket to the multicast destination address—all sender socket options must be set before issuing the connect call.

FEC

FEC is a method by which H parity packets are created out of K original data packets (for a total of N packets) so that the receiver can reconstruct the K original packets out of ***any*** K packets out of N received. For example, if for each four original data packets an additional four parity packets were created (for a total of eight packets), then the receiver needs to receive only ***any*** four of the eight to reconstruct the original data.

The algorithm employs packet-level Reed Solomon Erasure correcting techniques. This means that when FEC is enabled, more packets are hitting the wire as a number of parity packets are generated in addition to the data, but receivers typically lose a small number of packets at a time. If the receiver loses one packet out of the N generated, it can still recover the original data packets, thus preventing a NAK for the lost data and wait for the retransmission.

There are two operational modes in which FEC is enabled: pro-active and on-demand. With pro-active, the sender always computes parity packets for all data sent. The drawback is that computing parity packets can be a processor-intensive calculation. The second mode is on-demand, which means the sender sends data normally until a receiver sends a NAK, at which point the repair data is sent as a FEC data. In general, pro-active FEC is used when a large number of receivers are on lossy networks and the expected retransmission of lost data will become too large. On-demand is a trade off between reliability and network overhead. It is useful when several clients on the same network lose different packets belonging to the same FEC group. In this case, each receiver NAKs for the lost packet, at which point the sender will send a single FEC repair packet that will satisfy all of the clients' NAK request.

To enable FEC on a socket, the RM_USE_FEC socket option is used. This option must be set before connecting the socket. The input parameter is an *RM_FEC_INFO* structure that is defined as

```
typedef struct _RM_FEC_INFO
{
    USHORT                FECBlockSize;
    USHORT                FECProActivePackets;
    UCHAR                 FECGroupSize;
    BOOLEAN               fFECOnDemandParityEnabled;
} RM_FEC_INFO;
```

The *FECBlockSize* field indicates the maximum number of packets generated. This field is the N packets discussed earlier (for example, original data packets plus parity packets). However, the current implementation ignores this field and instead relies on the sum of the *FECProActivePackets* and *FECGroupSize* fields to determine the block size. *FECProActivePackets* indicates the number of parity packets to be generated—this is the H parity packets generated. *FECGroupSize* is the number of original data packets used to generate parity information from—the original data packets, K. Finally, *fFECOnDemandParityEnabled* indicates whether on-demand FEC requests should be allowed.

Reliable Receiver

Setting up a reliable multicast receiver consists of the following five steps.

1. Create a reliable multicast socket.
2. Bind the socket specifying the multicast group address to join the session on.
3. If the receiver wishes to listen on specific interfaces (as opposed to listening on all interfaces), call *setsockopt* and *RM_ADD_RECEIVE_IF* to add each interface.
4. Call *listen()*.
5. Wait on an accept function.

There are a couple of important differences to note with the receiver. First, when binding the socket, the local interface is not specified. Instead, the multicast group the receiver wishes to join on is given. If the port number specified in the *SOCKADDR_IN* structure is zero, any reliable multicast session for the specified multicast group is accepted. Otherwise, if a specific port is given, then a reliable multicast session will be joined only if it matches the multicast group ***and*** the port number.

The second difference is that the receiver must add which interfaces to listen for incoming sessions on. Because the bind call is used to specify the multicast session to join, it is necessary to indicate which local interfaces should listen on for incoming sessions. By default, the socket will listen for incoming connections on any local interface. If the application wishes to accept sessions from a specific interface it must call *RM_ADD_RECEIVE_IF* for each local interface. Of course, this is important only on multihomed computers. The argument to the *RM_ADD_RECEIVE_IF* is the 32-bit network-byte order local interface to listen on. This option may be called multiple times. The option *RM_DEL_RECEIVE_IF* can be used to remove an interface from the listening set; however, it may be called only if *RM_ADD_RECEIVE_IF* has been invoked previously.

When a receiver joins a reliable multicast session, it does not have to start receiving at the beginning of the sender's data transmission. As a part of each session, the oldest data that may be requested for repair is advertised. Therefore, if the client joins the session after the sender has started transmitting data, the client may NAK for data back to the advertised sequence number. This is known as a late join. The sender can actually limit how much of its send window may be *NAK*ed by late joiners (via the *RM_LATEJOIN* option). Of course, this is totally transparent to the application. The reliable multicast provider will automatically NAK for all available data when the session is joined. Once the session is joined, if data is lost and cannot be repaired, the session will be aborted. If, on the other hand, the sender closes the session, when all the remaining data has been delivered to the application the next receive operation will fail with the error *WSEDISCONN*.

The following code sample illustrates how to set up a reliable multicast receiver:

```
SOCKET             s,
                   ns;
SOCKADDR_IN        multi,
                   safrom;
ULONG              localif;
char               buf[1024];
int                buflen=1024,
                   fromlen,
                   rc;

s = socket(AF_INET, SOCK_RDM, IPPROTO_RM);

multi.sin_family = AF_INET;
multi.sin_port   = htons(5150);
multi.sin_addr.s_addr = inet_addr("234.5.6.7");
bind(s, (SOCKADDR *)&multi, sizeof(multi));

listen(s, 10);

localif = inet_addr("157.124.22.104");
setsockopt(s, IPPROTO_RM, RM_ADD_RECEIVE_IF,
                (char *)&localif, sizeof(localif));

fromlen = sizeof(safrom);
ns = accept(s, (SOCKADDR *)&safrom, &fromlen);

closesocket(s);  // Don't need to listen anymore

// start receiving data . . .
while (1)  {
    rc = recv(ns, buf, buflen, 0);
        if (rc == SOCKET_ERROR) {
        if (WSAGetLastError() == WSAEDISCON)
            break;
        else {
            // An unexpected error
        }
    }
}
closesocket(ns);
```

ATM Multipoint

Native ATM through Winsock also supports multicasting, which offers significantly different capabilities than IP multicasting. Remember that ATM supports rooted control and data planes, so when a multicast server, or c_root, is established, it has control over who is allowed to join the group as well as how data is transmitted within the group.

One important distinction is that on an ATM network, IP over ATM can be enabled. This configuration allows the ATM network to emulate an IP network by mapping IP addresses to ATM native addresses. With IP over ATM enabled, you have a choice of using IP multicasting, which will be translated appropriately to the ATM layer, or of using the native ATM multicasting capabilities, which we present in this section. The behavior of IP multicasting on ATM when configured for IP over ATM should be the same because it appears that you are on an IP network. The only exception is that IGMP is not present because all multicast calls are translated into ATM native commands. In addition, it is possible to configure an ATM network with one or more LAN emulation (LANE) networks. The purpose of a LANE is to make the ATM appear as a "regular" network capable of multiple protocols such as IPX/SPX, NetBEUI, TCP/IP, IGMP, and ICMP. In this situation, IP multicasting does look every bit like IP multicasting on an Ethernet network, which means IGMP is present as well.

We mentioned that ATM supports rooted control planes and rooted data planes. Therefore, when you create a multicast group, you establish a root node that "invites" leaf nodes to join the multicast group. Currently, only root-initiated joins are supported, meaning that a leaf cannot request to be added to a group. In addition, the root node (as a rooted data plane) sends data in one direction only, from the root to the leaves.

One startling contrast between ATM and IP multicasting is that ATM needs no special addresses. All that is required is that the root has knowledge of the addresses of each leaf that it will invite. Also, only one root node can be in a multicast group. If another ATM endpoint starts inviting the same leaves, this association becomes a separate group.

ATM Multipoint with *WSAJoinLeaf*

The *WSAJoinLeaf* API is the only method of using ATM multicasting. In addition, the semantics of setting up an ATM multipoint session are client-server oriented but conceptually reversed. The server creates a socket and then initiates a multipoint join with *WSAJoinLeaf* to each client it wishes to invite into the multicast group. Each client creates a socket, listens, and waits on an accept call. If the server invites the client, the accept call completes.

The next two sections describe the exact steps required for setting up an ATM leaf (client) node followed by the root (server) node. An ATM multicast sample is provided on the companion CD in the directory ATM.

ATM Leaf Node

Creating a leaf node in a multicast group is straightforward. On an ATM network, the leaf must listen for an invitation from a root to join a group. Here are the four steps needed to do this:

1. Using the *WSASocket* function, create a socket of address family *AF_ATM* including the flags *WSA_FLAG_MULTIPOINT_C_LEAF* and *WSA_FLAG_MULTIPOINT_D_LEAF*.
2. Bind the socket to the local ATM address and port with the *bind* function.
3. Call *listen*.
4. Wait for an invitation by using *accept*, *WSAAccept*, or *AcceptEx*. Depending on the I/O model you use, this will differ. (See Chapter 5 for a more thorough description of the Winsock I/O models.)

Once the connection is established, the leaf node can receive data sent from the root. Remember that with ATM multicasting, the data flow is one way: from the root to the leaves.

> Windows 98, Windows Me, Windows 2000, and Windows XP currently support only a single ATM leaf node on the system at any given time. Only a single process on the entir system can be a leaf member of any ATM point-to-multipoint session.

ATM Root Node

Creating a root node is even easier than creating an ATM leaf. The process includes two basic steps:

1. Using the *WSASocket* function, create a socket of address family *AF_ATM* including the flags *WSA_FLAG_MULTIPOINT_C_ROOT* and *WSA_FLAG_MULTIPOINT_D_ROOT*.
2. Call *WSAJoinLeaf* with the ATM address for each endpoint you want to invite.

A root node can invite as many endpoints as it wants, but it must issue a separate *WSAJoinLeaf* call for each.

Conclusion

Multicasting offers a number of advantages for applications that need to communicate with multiple endpoints without the overhead of broadcasting. In this chapter, we defined multicasting and presented the different multicasting models. We then discussed the different IP multicasting options, including IPv4 and IPv6 multicasting, IPv4 source multicasting, and reliable multicasting. Finally, we covered ATM point-to-multipoint communication.

Chapter 10

Generic Quality of Service

With the variety of multimedia applications available today, as well as the popularity of the Internet, many networks are becoming saturated with the traffic of these bandwidth-hungry applications. This is especially a problem on shared media networks—such as Ethernet—because all traffic is treated equally and a single application can flood the network. QOS is a set of components that allows the differentiation and preferential treatment of data on a network. A QOS-enabled network can be configured to offer programmers the following capabilities:

- Prevent non-adaptive protocols (such as UDP) from abusing network resources
- Partition resources between “best-effort” traffic and higher-priority or lower-priority traffic
- Reserve resources for entitled users
- Prioritize access to resources based on the user

Generic Quality of Service (GQOS) is Microsoft's implementation of QOS. Currently, Microsoft supplies a QOS-enabled TCP/IPv4 and UDP/IPv4 provider that is available in Windows 98, Windows Me, Windows 2000, and Windows XP. However, in Windows XP, the IP QOS provider is scaled down and does not offer full GQOS functionality, as we will describe later in the chapter. Note that QOS is not available over IPv6 providers. Microsoft platforms also can access QOS using the ATM protocol because QOS functionality is available natively in ATM.

This chapter covers QOS and how it is implemented on Windows platforms. We will begin with a discussion of the components that need to be in place to allow preferential treatment of network traffic. Then we'll examine how the Winsock interface exposes the capability to write a network application that can take advantage of these components for time- and bandwidth-critical applications. The majority of this chapter is dedicated to QOS on IP networks. At the end of this chapter, we will discuss QOS on ATM networks, which is slightly different from QOS on IP networks.

> Throughout the chapter, you can assume that when we refer to QOS, we are discussing Microsoft's implementation of it.

Background

QOS requires three components to make it work:

- Devices on the network—such as routers and switches—that are QOS aware
- Local workstations that can prioritize traffic that they place on the network
- The policy component: who is allowed to use the available bandwidth and how much they are allowed to use

Before we begin discussing these components, we need to look at the RSVP, which is the signaling protocol used between QOS senders and QOS receivers. RSVP plays a major role in QOS and the integration of the three major components of QOS.

RSVP

RSVP is the glue that binds the network, application, and policy components into one cohesive unit. RSVP carries resource reservation requests through the network, which can be composed of different media. RSVP propagates a user's QOS requests to all RSVP-aware network devices along the data path, allowing resources to be reserved from all RSVP-enabled devices. As a result, the network nodes can indicate whether the network can meet the desired levels of service.

The RSVP protocol reserves network resources by establishing flows end to end through the network. A flow is a network path associated with one or more senders, one or more receivers, and a specific level of QOS. A sending host wanting to send data that requires a specific level of QOS issues a PATH message toward the intended recipient or recipients. This PATH message contains the bandwidth requirements. The relevant parameters are propagated along the path to the intended recipients.

A receiving host that is interested in this data reserves the resources for the flow (and the entire path from the sender) by sending an RESV (reserve) message back toward the sender. As this occurs, intermediate RSVP-enabled devices decide whether they can accommodate the requested bandwidth requirements and ensure that the user requesting resources has the permission to do so. If the requested bandwidth is available and the user's policy settings indicate the user has the right to the request, each intermediate RSVP-enabled device commits the resources and propagates the RESV message back toward the sender.

When the sender receives the RESV message, QOS data can begin to flow. Periodically, each endpoint within the flow sends out PATH and RESV messages to reaffirm the reservation and to provide network information in case the levels of available bandwidth change. Also, by periodically refreshing PATH and RESV messages, the RSVP protocol remains dynamic. If a better (for example, faster) route becomes available, these refresh messages can discover a new route. When we discuss QOS from Winsock later in this chapter, we'll return to RSVP and how the Winsock API calls invoke it.

Be aware of one important aspect of the session setup and RSVP: It is a one-way reservation. This is the case even if the application requests bandwidth requirements for both sending and receiving. One session is initiated for the sending requirements and another session is started for the receiving requirements. Later in this chapter, we will discuss the criteria required for initiating an RSVP session.

Network Components

For end-to-end QOS to work, the network devices between the two endpoints must also be able to differentiate traffic priorities. This way they can route traffic in a manner that satisfies the QOS guarantee that an application received. In addition, these network devices must be able to determine whether enough bandwidth is available on the network when an application requests it. To support these requirements, the following components have been created:

- 802.1p A standard for prioritizing packets in a subnet by setting three bits within the media access control (MAC) header of packets
- IP Precedence A method to establish priority for IP packets
- Layer 2 signaling A mechanism for mapping RSVP objects to native WAN QOS components in a network's OSI Layer 2
- Subnet Bandwidth Manager (SBM) A component that manages shared media network bandwidth
- RSVP A protocol that carries QOS requests and information to QOS-aware network devices along the path between a sender and one or more receivers

802.1p

A major part of enforcing QOS provisions and avoiding treating all packets equally lies on hubs and switches within the network. Hubs and switches lie within Layer 2 of the OSI reference model and as a result are aware of fields only within the MAC header at the beginning of each packet.

802.1p is a standard that prioritizes network packets by setting a 3-bit precedence value in the MAC header. When a subnet becomes congested on non-802.1p networks, switches and routers are unable to keep up with the amount of traffic and a delay is introduced. On the other hand, switches and routers on 802.1p networks can begin prioritizing incoming traffic based on the precedence bits and give the higher-priority packets preferential treatment.

Implementing 802.1p for QOS requires special hardware capable of recognizing this 3-bit field. Network interface cards (NIC), network drivers, and network switches all must be 802.1p-aware.

IP Precedence

IP Precedence is a method of specifying precedence values at a higher level than 802.1p. This method allows packets passing through OSI Layer 3 devices—such as routers—to have their relative priorities differentiated. IP Precedence is implemented by using the TOS field within the IP header to establish varying levels of priority. Based on these bits, routers can establish priority queues to service the different priority levels, so higher priority traffic receives better service from routers.

As with 802.1p, for IP Precedence to work, all Layer 3 devices on the network must be able to understand the significance of the IP Precedence bits and handle traffic accordingly.

Layer 2 Signaling

Layer 2 signaling is necessary when traffic traverses a WAN. Typically, a WAN links several networks over a variety of communications hardware. Thus, a WAN can manipulate Layer 1, Layer 2, and Layer 3 information as data is transmitted. To guarantee end-to-end QOS, the WAN link must understand the prioritization of QOS traffic. To accomplish this understanding, QOS provides a method for mapping RSVP and other QOS parameters to the WAN's native underlying Layer 2 signaling method—the means by which WAN technologies implement their own native QOS.

SBM

The SBM manages the resources on a given shared media network, such as Ethernet. The SBM is also responsible for handling policy-based admission control for QOS applications. An SBM is necessary on shared media networks because when an endpoint requests QOS for an application, each network device admits or rejects the request based on the allocation of the network device's private resources. The network devices are not aware of the available resources on the shared media. The SBM solves this problem by becoming a broker for these devices. The SBM is also closely tied to the Admission Control Service (ACS) that is a part of the policy component. The SBM must check to ensure that an application (or user) requesting bandwidth has the privileges to do so. Note that the SBM for a network can be a host running Windows 2000 Server.

Application Components

You now have a good idea of network requirements for supporting QOS. We must consider how the local system prioritizes data based on the QOS levels that an application has requested. For the local system to support QOS, the following components are necessary:

- GQOS service provider A service provider that invokes other QOS components
- Traffic Control (TC) module The module that controls the traffic leaving the computer. (This module includes the Generic Packet Classifier (GPC), the Packet Scheduler, and the Packet Shaper.)
- RSVP The protocol that is invoked by the GQOS service provider and that carries the reservation request across the network
- GQOS API The programmatic interface to GQOS, such as Winsock
- TC API The programmatic interface to the TC components regulating traffic on the local host

GQOS Service Provider

The GQOS service provider is the GQOS component that invokes nearly all resulting QOS facilities. The GQOS service provider initiates TC functionality (if appropriate) and implements, maintains, and handles RSVP signaling for all GQOS functionality.

To find the QOS-enabled service provider(s) on your host, you can query the provider catalog with *WSAEnumProtocols*. The flag to check whether a provider is QOS-enabled is within the *WSAPROTOCOL_INFO* structure returned from *WSAEnumProtocols*. The field of interest is *dwServiceFlags1*, and the flag to check for is *XP1_QOS_SUPPORTED*. For more information about *WSAEnumProtocols*, consult Chapter 2.

TC Module

TC plays a significant and central role in QOS. Within TC, packets are prioritized both within and outside the network node on which TC is enabled. The effects of this preferential treatment of packets as they flow through the system and through the network reach across the entire network and therefore directly affect QOS characteristics. The TC module is implemented through three modules: the Generic Packet Classifier, the Packet Scheduler, and the Packet Shaper.

GPC

The duty of the GPC is to classify and prioritize packets within network components. The GPC performs this prioritization for activities such as CPU time or transmission onto the network.

The GPC accomplishes this prioritization by creating lookup tables and classification services within the network stack. This becomes the first step in the prioritization process for network traffic.

Packet Scheduler

Packet scheduling controls the way data transmission is performed, which is a key function of QOS. The Packet Scheduler is the TC module that regulates how much data an application, or flow, is allowed, essentially enforcing QOS parameters set for a particular flow.

The Packet Scheduler takes the prioritization scheme that the GPC provides and offers different levels of service to the various priority levels. For example, data that has been classified by the GPC as high priority receives preferential treatment within the Packet Scheduler.

Packet Shaper

The purpose of the Packet Shaper is to regulate the transmission of data from data flows onto the network. Most applications read and write data in bursts; however, many QOS applications need a particular data rate for sent data. Therefore, the Packet Shaper schedules the transmission of data over a period of time, smoothing out network usage and resulting in a more evenly loaded network.

TC API

The TC API is the interface to the components that regulate network traffic on the local host. This includes methods for manipulating the GPC, the Packet Scheduler, and the Packet Shaper. Some TC functions are implicitly invoked through calls made to Winsock GQOS–enabled functions that are serviced by the GQOS Service Provider. However, applications that need to manipulate the TC components directly can do so with the TC API functions. These functions are beyond the scope of this book. (Consult the Platform Software Development Kit (SDK) for more information.) We will cover the Winsock GQOS API later in this chapter.

Policy Components

Policy, the third and final component of GQOS, controls the allocation of resources to QOS-enabled applications. Policy components are of most interest to system administrators who want to control the allocation of resources based on users or on the class of application requesting bandwidth. Policy components include the following:

- ACS A Windows 2000 Server service that intercepts RSVP PATH and RESV messages to control access of QOS-enabled clients to the various levels of guarantees that QOS offers
- Local Policy Module (LPM) Provides resource-access decisions based on policies configured through ACS for the SBM
- Policy Element (PE) Resides on the client and provides authentication information to facilitate reservation requests

ACS

The ACS regulates network usage for QOS-enabled applications. This is done through the RSVP protocol. The ACS intercepts both PATH and RESV messages to verify that the requesting application has sufficient privileges. Once an RSVP message is intercepted, it is passed to the LPM, which performs the actual authentication.

The ACS resides on a Windows 2000 machine and can be configured by the system administrator, who can set resource limits on users, applications, or groups.

LPM

The LPM is closely related to the ACS in that the ACS intercepts RSVP messages, inserts user information, and passes the messages to the LPM. At this point, the LPM looks the user up in Active Directory to verify policy information. If network resources are available (as determined by the SBM), and if the authentication check succeeds, the RSVP message that the ACS intercepts is sent to the next hop. Of course, if the user does not have the necessary permissions to request a certain level of QOS, an error indicating this is generated and returned within the RSVP message.

For policy checks to succeed, users must be part of a Windows 2000 domain.

Policy Element

This component contains the policy information that the LPM requests. These data structures are not covered in this book because they deal mainly with administration of network resources, which is not the focus of this book.

QOS and Winsock

In the previous section, we discussed the various components required for the success of an end-to-end QOS network. Now we'll turn our attention to Winsock 2, which is the API you use to programmatically access QOS from an application. First, we'll take a look at the top-level QOS structures that the majority of Winsock calls need. Next, we'll cover the Winsock functions capable of invoking QOS on a socket, as well as how to terminate QOS once it has been enabled on a socket. The last thing we'll do is cover the provider-specific objects that can be used to affect the behavior of —or return information from—the QOS service provider.

It might seem a bit out of order to jump from a discussion of the major QOS structures to QOS functions and then back to provider-specific structures. However, we want to give a thorough, high-level overview of how the major structures interact with the Winsock API calls before delving into the details of the provider-specific options.

QOS Structures

The central structure in QOS programming is the *QOS* structure. This structure consists of

- A *FLOWSPEC* structure used to describe the QOS levels that your application will use for sending data
- A *FLOWSPEC* structure used to describe the QOS levels that your application will use to receive data
- A service provider–specific buffer to allow the specification of provider-specific QOS characteristics (We will discuss these characteristics in the provider-specific section.)

QOS

The *QOS* structure specifies the QOS parameters for both sending and receiving traffic. It is defined as

```
typedef struct _QualityOfService
{
    FLOWSPEC   SendingFlowspec;
    FLOWSPEC   ReceivingFlowspec;
    WSABUF     ProviderSpecific;
} QOS, FAR * LPQOS;
```

The *FLOWSPEC* structures define the traffic characteristics and requirements for each traffic direction, while the *ProviderSpecific* field is used to return information and to change the behavior of QOS. These provider-specific options are covered in detail a little later in this chapter.

FLOWSPEC

FLOWSPEC is the basic structure that describes a single flow. Remember that a flow describes data traveling in a single direction. The structure is defined as

```
typedef struct _flowspec
{
    ULONG        TokenRate;
    ULONG        TokenBucketSize;
    ULONG        PeakBandwidth;
    ULONG        Latency;
    ULONG        DelayVariation;
    SERVICETYPE  ServiceType;
    ULONG        MaxSduSize;
    ULONG        MinimumPolicedSize;
} FLOWSPEC, *PFLOWSPEC, FAR *LPFLOWSPEC;
```

Let's take a look at the meaning of each of the *FLOWSPEC* structure fields.

TokenRate

The *TokenRate* field specifies the rate of transmission for data that is given in bytes per second. An application can transmit data at this rate, but if for some reason it transmits data at a lower rate, the application can accrue extra tokens so that more data can be transmitted later. However, the number of tokens that an application can accrue is bound by *PeakBandwidth*. This accumulation of token credits is limited by the *TokenBucketSize* field. By limiting the total number of tokens, we avoid a situation of inactive flows that have accrued many tokens, which could lead to flooding the available bandwidth. Because flows can accrue transmission credits over time (at their *TokenRate* value) ***only*** up to the maximum of their *TokenBucketSize*, and they are limited in “burst transmissions” to their *PeakBandwidth*, TC and network-device resource integrity are maintained. TC is maintained because flows cannot send too much data at once, and network-device resource integrity is maintained because such devices are spared high-traffic bursts.

Because of these limitations, an application can transmit only when sufficient credits have accrued. If the required number of credits are not available, the application must either wait until sufficient credits have accrued to send the data or discard the data altogether. The TC module determines what happens to data queued too long without being sent. Therefore, applications should take care to base their *TokenRate* requests on reasonable amounts.

If an application does not require scheduling of transmission rates, this field can be set to *QOS_NOT_SPECIFIED* (-1).

TokenBucketSize

As we discussed earlier, the *TokenBucketSize* field limits the number of credits that can accrue for a given flow. For example, video applications would set this field to the frame size being transmitted because it is desirable to have single video frames being transmitted one at a time. Applications requiring a constant data rate should set this field to allow for some variation. Like the *TokenRate* field, *TokenBucketSize* is expressed in bytes per second.

PeakBandwidth

PeakBandwidth specifies the maximum amount of data transmitted in a given period of time. In effect, this value specifies the maximum amount of burst data. This is an important value because it prevents applications that have accrued a significant number of transmission tokens from flooding the network all at once. *PeakBandwidth* is expressed in bytes per second.

Latency

The *Latency* field specifies the maximum acceptable delay between the transmission of a bit and its receipt by the intended recipient or recipients. How this value is interpreted depends on the level of service requested in the *ServiceType* field. *Latency* is expressed in microseconds.

DelayVariation

DelayVariation specifies the difference between the minimum and maximum delay that a packet can experience. Typically, an application uses this value to determine the amount of buffer space required to receive the data and still maintain the original data transmission pattern. *DelayVariation* is expressed in microseconds.

ServiceType

The *ServiceType* field specifies the level of service that the data flow requires. The following service types can be specified:

- *SERVICETYPE_NOTRAFFIC* indicates that no data is being transmitted in this direction.
- *SERVICETYPE_BESTEFFORT* indicates that the parameters specified in *FLOWSPEC* are guidelines and that the system will make a reasonable effort to maintain that service level; however, there are no guarantees of packet delivery.
- *SERVICETYPE_CONTROLLEDLOAD* indicates that data transmission will closely approximate transmission quality provided by best-effort service on a network with non-loaded traffic conditions. This really breaks down into two conditions. First, packet loss will approximate the normal error rate of the transmission medium; and second, transmission delay will not greatly exceed the minimum delay that delivered packets experience.
- *SERVICETYPE_GUARANTEED* guarantees data transmission at the rate specified by the *TokenRate* field over the lifetime of the connection. However, if the data transmission rate exceeds *TokenRate*, data might be delayed or discarded (depending on how TC is configured). In addition, if *TokenRate* is not exceeded, *Latency* is also guaranteed.
- *SERVICETYPE_QUALITATIVE* behaves similarly to *SERVICETYPE_ BESTEFFORT*.
- *SERVICETYPE_NETWORK_CONTROL* has the highest priority and is typically used only for controlling the network. In general, you should not use this level.

In addition to these six service types, several other flags provide information that can be returned to an application. These informational flags can be *OR*ed with any valid *ServiceType* flag. Table 10-1 lists these information flags.

Table 10-1 *Service Type Modifier Flags*

Value	Meaning
SERVICETYPE_NETWORK_UNAVAILABLE	Indicates a loss of service in either the sending or the receiving direction.
SERVICETYPE_GENERAL_INFORMATION	Indicates that all service types are supported for a flow.
SERVICETYPE_NOCHANGE	Indicates that there is no change in the requested QOS service level. This flag can be returned from a Winsock call or an application can specify this flag when renegotiating QOS to indicate no change in the QOS levels for the given direction.
SERVICETYPE_NONCONFORMING	This flag is not used.
SERVICE_NO_TRAFFIC_CONTROL	This flag can be *OR*ed with other *ServiceType* flags to disable TC altogether.
SERVICE_NO_QOS_SIGNALING	This flag prevents any RSVP signaling messages from being sent. Local TC will be invoked, but no RSVP Path messages will be sent. This flag can also be used in conjunction with a receiving *FLOWSPEC* structure to suppress the automatic generation of an RESV message. The application receives notification that a PATH message has arrived and then needs to alter the QOS by issuing *WSAIoctl (SIO_SET_QOS)* to unset this flag and thereby cause RESV messages to go out.

MaxSduSize

The *MaxSduSize* field indicates the maximum packet size for data transmitted in the given flow. *MaxSduSize* is expressed in bytes.

MinimumPolicedSize

The *MinimumPolicedSize* field indicates the minimum packet size that can be transmitted in the given flow. *MinimumPolicedSize* is expressed in bytes.

QOS-Invoking Functions

Let's say you want your application to make a request on the network for certain bandwidth requirements. Four functions initiate the process. Once an RSVP session has begun, an application can register for *FD_QOS* events. QOS status information and error codes are conveyed to applications as *FD_QOS* events. Applications can register to receive these events in the usual way: by including the *FD_QOS* flag in the event field of either the *WSAAsyncSelect* function or the *WSAEventSelect* function.

The *FD_QOS* notification is especially relevant if a connection is established that uses *FLOWSPEC* structures that specify default values (*QOS_NOT_SPECIFIED*). Once the application has made the request for QOS, the underlying provider will periodically update the *FLOWSPEC* structure to indicate current network conditions and will notify the application by posting an *FD_QOS* event. With this information, applications can request or modify QOS levels to reflect the amount of available bandwidth. Keep in mind that the updated information is an indication of the locally available bandwidth only and does not necessarily indicate the end-to-end bandwidth.

Once a flow is established, available network bandwidth might change or a single party taking part in an established flow might decide to change the requested QOS service level. A renegotiation of allocated resources generates an *FD_QOS* event to indicate the change to the application. At this point, the application should call *SIO_GET_QOS* to obtain the new resource levels. We'll revisit QOS event signaling and status information in the section on programming QOS later in this chapter.

WSAConnect

A client uses the *WSAConnect* function to initiate a unicast QOS connection to a server. WSAConnect is defined as follows:

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

The requested QOS values are passed as the *lpSQOS* parameters. Currently, group QOS is not supported or implemented; a null value should be passed for *lpGQOS*.

The *WSAConnect* call can be used with connection-oriented or connectionless sockets. With a connection-oriented socket, this function establishes the connection and also generates the appropriate PATH and/or RESV messages. For connectionless sockets, you must associate an endpoint's address with the socket so that the service provider knows where to send PATH and RESV messages. The caveat with using *WSAConnect* on a connectionless socket is that only data sent to that destination address will be shaped by the system according to the QOS levels associated with that socket. In other words, if *WSAConnect* is used to associate an endpoint on a connectionless socket, data can be transferred only between those two endpoints for the lifetime of the socket. If you need to send data with QOS guarantees to multiple endpoints, use *WSAIoctl* and *SIO_SET_QOS* to specify each new endpoint.

WSAAccept

The *WSAAccept* function accepts a client connection that can be QOS-enabled. The prototype for the function is as follows:

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

If you want to supply a conditional function, you must prototype it as

```
int CALLBACK ConditionalFunc(
    LPWSABUF lpCallerId,
    LPWSABUF lpCallerData,
    LPQOS lpSQOS,
    LPQOS lpGQOS,
    LPWSABUF lpCalleeId,
    LPWSABUF lpCalleeData,
    GROUP FAR *g,
    DWORD dwCallbackData
);
```

The drawback is that the QOS service provider does not guarantee to return the actual QOS values that the client requests as the *lpSQOS* parameter, so to enable QOS on the client socket, *WSAIoctl* with *SIO_SET_QOS* must be called before or after *WSAAccept*. If QOS is set on the listening socket, those values will be copied over to the client socket by default.

Unfortunately, the QOS service provider will not pass valid QOS parameters into the conditional function even if a PATH message has already arrived. Basically, don't use the *WSAAccept* condition function.

> There is one issue to be aware of when using *WSAAccept* in Windows 98 and Windows Me. If you use a conditional function with *WSAAccept* and the *lpSQOS* parameter is not null, you must set QOS (using *SIO_SET_QOS*), or *WSAAccept* will fail.

WSAJoinLeaf

WSAJoinLeaf is used for multipoint communications. Chapter 9 discusses multicasting in great detail. The function is defined as

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

For an application to join a multicast session, it must create a socket that has the appropriate flags (*WSA_FLAG_MULTIPOINT_C_ROOT*, *WSA_FLAG_MULTIPOINT_C_LEAF*, *WSA_FLAG_MULTIPOINT_D_ROOT*, and *WSA_FLAG_MULTIPOINT_D_LEAF*). When the application sets up multipoint communications, it specifies QOS parameters in the *lpSQOS* parameter.

When you use *WSAJoinLeaf* to join IP multicast groups, the operation of joining a multicast group is separate from the QOS RSVP session setup. In fact, joining a multicast group is likely to succeed. The function returns without the reservation request completing. At some later time, you will receive an *FD_QOS* event that will notify you of either a success or a failure in allocating the requested resources.

Keep in mind the TTL set on multicast data. If you plan on setting the TTL with either *SIO_MULTICAST_SCOPE* or *IP_MULTICAST_TTL*, it must be set prior to calling *WSAJoinLeaf* or calling the *SIO_SET_QOS* ioctl command to set QOS on the socket. If the scope is set after the QOS is already set, the TTL will not take effect until QOS is renegotiated through *SIO_SET_QOS*. The TTL value set will also be carried by the RSVP request.

Setting the TTL before setting QOS on a socket is important because the multicast TTL set on the socket also affects the TTL of the RSVP messages, which directly affects how many networks your resource reservation request is propagated to. For example, if you want to set up several endpoints in an IP multicast group that spans three networks, you ideally would set the TTL to 3 so that the network traffic you generate is not propagated to networks beyond those interested in the data. If the TTL isn't set before *WSAJoinLeaf* is called, RSVP messages are sent out with a default TTL of 63, which results in the host attempting to reserve resources on far too many networks.

WSAIoctl

The *WSAIoctl* function with the ioctl option *SIO_SET_QOS* can be used either to request QOS for the first time on either a connected or an unconnected socket or to renegotiate QOS requirements after an initial QOS request. The one advantage to using *WSAIoctl* is that if the QOS request fails, more detailed error information is returned via the provider-specific information. Chapter 7 covers the *WSAIoctl* function and how it is called, along with *SIO_SET_QOS* and *SIO_GET_QOS*.

The *SIO_SET_QOS* option is used to set or modify QOS parameters on a socket. One feature of using *WSAIoctl* with *SIO_SET_QOS* is the capability to specify provider-specific objects to further refine QOS's behavior. The next section is dedicated to covering all of the provider-specific objects. In particular, if an application using connectionless sockets does not want to use *WSAConnect*, it can call *WSAIoctl* with *SIO_SET_QOS* and specify the destination address object in the provider-specific buffer to associate an endpoint so that an RSVP session can be established. When setting QOS parameters, pass the QOS structure as *lpvInBuffer*, with *cbInBuffer* indicating the amount of bytes passed in.

The *SIO_GET_QOS* option is used upon receipt of an *FD_QOS* event. When an application receives this event notification, a call to *WSAIoctl* with *SIO_GET_QOS* should be made to investigate the reason. As we mentioned earlier, the *FD_QOS* event can be generated because of a change in the available bandwidth on the network or by renegotiation by the peer. To obtain the QOS values for a socket, pass a sufficiently large buffer as *lpvOutBuffer*, with *cbOutBuffer* indicating the size. The input parameters can be *NULL* and 0. The one tricky part of calling *SIO_GET_QOS* is passing a buffer large enough to hold the *QOS* structure, including the provider-specific objects. The *ProviderSpecific* field—a *WSABUF* structure—is within the *QOS* structure. If the *len* field is set to zero and the *buf* field is null, *len* will be updated with the necessary size upon return from *WSAIoctl*. In addition, if the call fails because the buffer is too small, the *len* field will be updated with the correct size. Querying for the buffer size is supported only in Windows 2000 and Windows XP. For Windows 98 and Windows Me, you must always supply a large enough buffer—simply pick a large buffer size and stick with it.

Another ioctl command can be used with *WSAIoctl*: *SIO_CHK_QOS*. This command can be used to query for the six values described in Table 10-2. When you call this command, the *lpvInBuffer* parameter points to a *DWORD* that is set to one of the three flags. The *lpvOutBuffer* parameter should also point to a *DWORD*, and upon return, the value requested is returned. The most commonly used flag is *ALLOWED_TO_SEND_DATA*. This flag is used by senders who have initiated a PATH message but have not received any RESV messages indicating successful allocation of the QOS level. When senders use the *SIO_CHK_QOS* ioctl command with the *ALLOWED_TO_SEND_DATA* flag, the network is queried to see whether the best-effort traffic currently available is sufficient for sending the kind of data described in the *QOS* structure passed to a QOS-invoking function. For more details, look at the entry for this ioctl command in Chapter 7.

Table 10-2 *SIO_CHK_QOS Flags*

***SIO_CHK_QOS* Flag**	**Description**	**Return Value**
ALLOWED_TO_SEND_DATA	Indicates whether sending data can begin immediately or whether the application should wait for an RESV message	*BOOL*
ABLE_TO_RECV_RSVP	Indicates to senders whether its interface is RSVP-enabled	*BOOL*
LINE_RATE	Returns the bandwidth capacity of the interface	*DWORD*
LOCAL_TRAFFIC_CONTROL	Returns whether TC is installed and available for use	*BOOL*
LOCAL_QOSABILITY	Returns whether QOS is available	*BOOL*
END_TO_END_QOSABILITY	Determines whether end-to-end QOS is available on the network	*BOOL*

The options listed in Table 10-2 that return *BOOL* values actually return 1 or 0 to indicate a yes or a no answer, respectively. The last four options in the table can return the constant *INFO_NOT_AVAILABLE* if the system cannot currently obtain the answer.

Terminating QOS

In the previous section, you learned how to invoke QOS on a socket. Next, we'll examine the termination of QOS guarantees. Each of the following events causes a termination of RSVP and TC processing for a socket.

- Closing a socket via the *closesocket* function
- Shutting down a socket via the *shutdown* function
- Calling *WSAConnect* with a null peer address
- Calling *WSAIoctl* and *SIO_SET_QOS* with the *SERVICETYPE_ NOTRAFFIC* or the *SERVICETYPE_BESTEFFORT* service type

Except for the second item in the list, these events are self-explanatory. Remember that the *shutdown* function can signal the cessation of either sending or receiving data, which will result in the termination of the flow of data for only that direction. In other words, if *shutdown* is called with *SD_SEND*, QOS will still be in effect for data being received.

Provider-Specific Objects

The provider-specific objects covered in this section are passed as part of the *ProviderSpecific* field of the *QOS* structure. Either they return QOS information to your application via the *FD_QOS* event or you can pass them along with the other QOS parameters to a *WSAIoctl* call with the *SIO_SET_QOS* option to refine QOS's behavior.

Every provider-specific object contains a *QOS_OBJECT_HDR* structure as its first member. This structure identifies the type of provider-specific object. This is necessary because these provider objects are most commonly returned within the *QOS* structure after a call to *SIO_GET_QOS*. By using the *QOS_ OBJECT_HDR*, your application can identify each object and decode its significance. The object header is defined as

```
typedef struct
{
    ULONG   ObjectType;
    ULONG   ObjectLength;
} QOS_OBJECT_HDR, *LPQOS_OBJECT_HDR;
```

ObjectType identifies the type of preset provider-specific object, while *ObjectLength* tells how long the entire object is, including the object header and the provider-specific object. An object type can be one of the flags listed in Table 10-3.

Table 10-3 *Object Types*

Provider Object	Object Structure
QOS_OBJECT_SD_MODE	*QOS_SD_MODE*

QOS_OBJECT_SHAPING_RATE	*QOS_SHAPING_RATE*
QOS_OBJECT_DESTADDR	*QOS_DESTADDR*
RSVP_OBJECT_STATUS_INFO	*RSVP_STATUS_INFO*
RSVP_OBJECT_RESERVE_INFO	*RSVP_RESERVE_INFO*
RSVP_OBJECT_ADSPEC	*RSVP_ADSPEC*
RSVP_OBJECT_POLICY_INFO	*RSVP_POLICY_INFO*
QOS_OBJECT_END_OF_LIST	None. No more objects.

QOS Shape Discard Mode

This *QOS* object defines how the Packet Shaper element of TC processes the data of a given flow. This property most often comes into play when dealing with flows that do not conform to the parameters given in *FLOWSPEC*. That is, if an application is sending data at a rate faster than what is specified in the *TokenRate* field of the sending *FLOWSPEC*, it is considered nonconforming. This object defines how the local system handles this occurrence. The *QOS_SD_MODE* structure is defined as

```
typedef struct _QOS_SD_MODE
{
    QOS_OBJECT_HDR   ObjectHdr;
    ULONG            ShapeDiscardMode;
} QOS_SD_MODE, *LPQOS_SD_MODE;
```

The *ShapeDiscardMode* field can be one of the values specified in Table 10-4.

Table 10-4 *QOS Shape DiscardMode Flags*

Flag	Description
TC_NONCONF_BORROW	The flow receives the resources remaining after all higher-priority flows have been serviced. Flows of this type are not subjected to either the Shaper or the Sequencer. If a value for *TokenRate* is specified, packets can be nonconforming and will be demoted to less than best-effort priority.
TC_NONCONF_BORROW_PLUS	Similar to *TC_NONCONF_BORROW*, however, packets will not be marked as nonconforming in the Shaper.
TC_NONCONF_SHAPE	A value for *TokenRate* must be specified. Nonconforming packets will be retained in the Packet Shaper until they become conforming.
TC_NONCONF_DISCARD	A value for *TokenRate* must be specified. Nonconforming packets will be discarded.

You might wonder why you would want to use the *TC_NONCONF_ DISCARD* mode when it might result in dropping data before it even gets sent on the wire. One such use is in sending audio or video data. In most cases, the *FLOWSPEC* structure is set up to reflect sending a packet whose size is equal to one frame of video or a small segment of audio. If for some reason the packet does not conform, is it better for an application to wait until it does conform (as is the case with *TC_NONCONF_SHAPE*), or should the application drop the packet altogether and move on to the next one? For time-critical data such as video, it is often better to drop the frame and move on.

QOS Destination Address

The *QOS_DESTADDR* structure is used to specify the destination address for a connectionless sending socket without using a *WSAConnect* call. No RSVP PATH or RESV messages will be sent until the destination address of a connectionless socket is known. The destination address can be set with the *SIO_SET_QOS* ioctl command. The structure is defined as

```
typedef struct _QOS_DESTADDR
{
    QOS_OBJECT_HDR         ObjectHdr;
    const struct sockaddr *SocketAddress;
    ULONG                  SocketAddressLength;
} QOS_DESTADDR, *LPQOS_DESTADDR;
```

The *SocketAddress* field references the *SOCKADDR* structure that defines the endpoint's address for the given protocol. *SocketAddressLength* is simply the size of the *SOCKADDR* structure.

RSVP Status Info

The RSVP status info object is used to return RSVP-specific error and status information. The structure is defined as

```
typedef struct _RSVP_STATUS_INFO {
    QOS_OBJECT_HDR                              ObjectHdr;
    ULONG            StatusCode;
    ULONG            ExtendedStatus1;
    ULONG            ExtendedStatus2;
} RSVP_STATUS_INFO, *LPRSVP_STATUS_INFO;
```

The *StatusCode* field is the RSVP message returned. The possible codes are described in Table 10-5. The other two fields, *ExtendedStatus1* and *ExtendedStatus2*, are reserved for provider-specific information.

Table 10-5 *RSVP Status Info Codes*

Flag	Meaning
WSA_QOS_RECEIVERS	At least one RESV message has arrived.
WSA_QOS_SENDERS	At least one PATH message has arrived.
WSA_QOS_NO_RECEIVERS	There are no receivers.
WSA_QOS_NO_SENDERS	There are no senders.
WSA_QOS_REQUEST_CONFIRMED	The reserve has been confirmed.
WSA_QOS_ADMISSION_FAILURE	Request failed due to lack of resources.
	Request rejected for administrative reasons or bad

WSA_QOS_POLICY_FAILURE	credentials.
WSA_QOS_BAD_STYLE	Unknown or conflicting style.
WSA_QOS_BAD_OBJECT	There is a problem with some part of the *RSVP_FILTERSPEC* structure or with the provider-specific buffer in general. (This object will be discussed shortly.)
WSA_QOS_TRAFFIC_CTRL_ERROR	There is a problem with some part of the *FLOWSPEC* structure.
WSA_QOS_GENERIC_ERROR	General error.
ERROR_IO_PENDING	Overlapped operation is canceled.

Typically, an application receives an *FD_QOS* event and calls *SIO_ GET_QOS* to obtain a QOS structure containing an *RSVP_STATUS_INFO* object when an RSVP message is received. For example, for a QOS-enabled UDP-based receiver, an *FD_QOS* event containing a *WSA_QOS_SENDERS* message is generated to indicate that someone has requested the QOS service to send data to the receiver.

RSVP Reserve Info

The RSVP reserve info object is used for storing RSVP-specific information for fine-tuning interactions via the Winsock 2 QOS APIs and the provider-specific buffer. An *RSVP_RESERVE_INFO* object overrides the default reservation style and is used by a QOS receiver. The object is defined as

```
typedef struct _RSVP_RESERVE_INFO
{
    QOS_OBJECT_HDR       ObjectHdr;
    ULONG                Style;
    ULONG                ConfirmRequest;
    LPRSVP_POLICY_INFO   PolicyElementList;
    ULONG                NumFlowDesc;
    LPFLOWDESCRIPTOR     FlowDescList;
} RSVP_RESERVE_INFO, *LPRSVP_RESERVE_INFO;
```

The *Style* field specifies the filter type that should be applied to this receiver. Table 10-6 lists the filter types available and the default filter types that different types of receivers use.

Table 10-6 *Default Filter Styles*

Filter Style	Default Users
Fixed filter	Unicast receivers; connected UDP receivers
Wildcard	Multicast receivers; unconnected UDP receivers
Shared explicit	None

Each filter style will be discussed in greater detail shortly. If the *ConfirmRequest* field is nonzero, notification will be sent once the RESV request has been received for receiving applications. *NumPolicyElement* is related to the *PolicyElementList* field. *PolicyElementList* is a list of

RSVP_POLICY objects that we define a little bit later in this chapter. Let's take a look at the different filter styles and the characteristics of each.

RSVP_DEFAULT_STYLE

This flag tells the QOS service provider to use the default style. Table 10-6 lists the default styles for the different possible receivers. Unicast receivers use fixed filter, whereas wildcard is for multicast receivers. UDP receivers that call *WSAConnect* also use fixed filter.

RSVP_FIXED_FILTER_STYLE

Normally, this style establishes a single flow with QOS guarantees between the receiver and a single source. This is the case for a unicast receiver and connected UDP receivers: *NumFlowDesc* is set to 1, and *FlowDescList* contains the sender's address. However, it is also possible to set up a multiple fixed filter style that allows a receiver to reserve mutually exclusive flows from multiple, explicitly identified sources. For example, if your receiver intends to receive data from three senders and needs guaranteed bandwidth of 20 Kbps for each, use the multiple fixed filter style. In this example, *NumFlowDesc* is set to 3, while *FlowDescList* contains three addresses, one for each *FLOWSPEC*. It is also possible to assign varying levels of QOS to each sender; they do not all have to be equal. Note that unicast receivers and connected UDP receivers cannot use multiple fixed filters. Figure 10-1 shows the relationship between *FLOWDESCRIPTOR* and *RSVP_FILTERSPEC* structures.

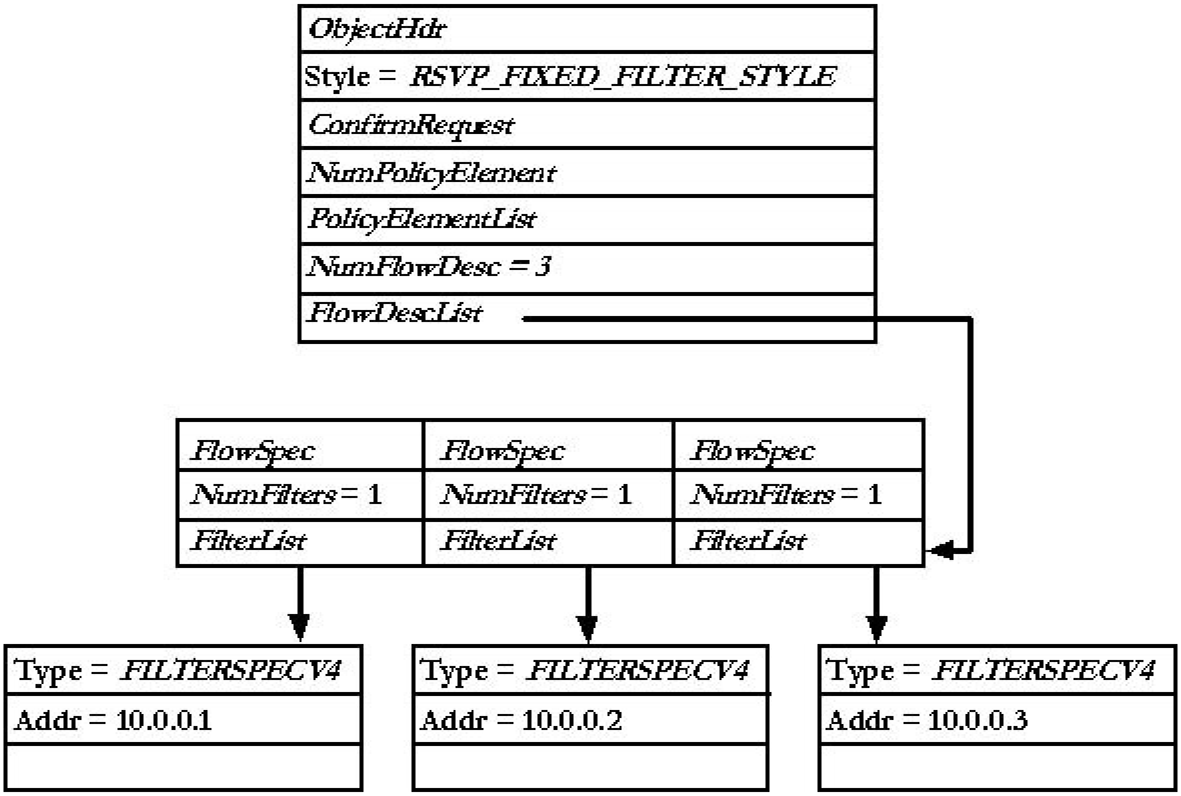


Figure 10-1 *Multiple fixed filter style*

RSVP_WILDCARD_STYLE

Multicast receivers and unconnected UDP receivers use the wildcard style. To use this style for TCP connections or for connected UDP receivers, set *NumFlowDesc* to 0 and *FlowDescList* to *NULL*. This is the default filter style for unconnected UDP receivers and multicast applications because the sender's address is unknown.

RSVP_SHARED_EXPLICIT_STYLE

This style is somewhat similar to multiple fixed filter style except that network resources are shared among all senders instead of being allocated for each sender. In this case, *NumFlowDesc* is 1 and *FlowDescList* contains the list of sender addresses. Figure 10-2 illustrates this style.

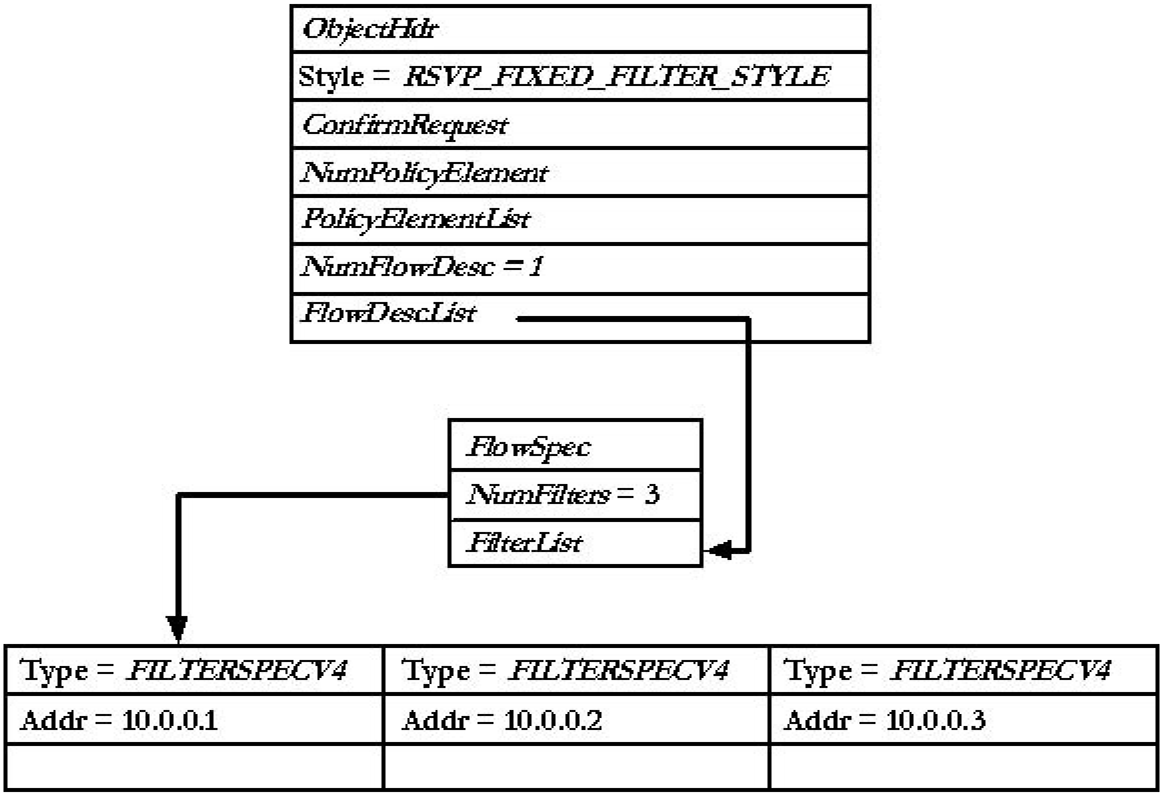


Figure 10-2 *Shared explicit style*

We've introduced the last two fields, *NumFlowDesc* and *FlowDescList*, in our discussion of RSVP styles. How you use these two fields depends on the style. *NumFlowDesc* defines the number of *FLOWDESCRIPTOR* objects in the *FlowDescList* field. This structure is defined as

```
typedef struct _FLOWDESCRIPTOR
{
    FLOWSPEC              FlowSpec;
    ULONG                 NumFilters;
    LPRSVP_FILTERSPEC     FilterList;
} FLOWDESCRIPTOR, *LPFLOWDESCRIPTOR;
```

This object is used to define the types of filters per *FLOWSPEC* given by *FlowSpec*. Again, the *NumFilters* field contains the number of *RSVP_FILTERSPEC* objects present in the *FilterList* array. The *RSVP_FILTERSPEC* object is defined as

```
typedef struct _RSVP_FILTERSPEC {
    FilterType    Type;
    union {
        RSVP_FILTERSPEC_V4      FilterSpecV4;
        RSVP_FILTERSPEC_V6      FilterSpecV6;
        RSVP_FILTERSPEC_V6_FLOW FilterSpecV6Flow;
        RSVP_FILTERSPEC_V4_GPI  FilterSpecV4Gpi;
        RSVP_FILTERSPEC_V6_GPI  FilterSpecV6Gpi;
    };
} RSVP_FILTERSPEC, *LPRSVP_FILTERSPEC;
```

The first field, *Type*, is a simple enumeration of the following values:

```
typedef enum {
        FILTERSPECV4 = 1,
        FILTERSPECV6,
        FILTERSPECV6_FLOW,
        FILTERSPECV4_GPI,
        FILTERSPECV6_GPI,
        FILTERSPEC_END
} FilterType;
```

This enumeration specifies the object present in the union. Each of these filter specs is defined as follows:

```
typedef struct _RSVP_FILTERSPEC_V4 {
    IN_ADDR_IPV4    Address;
    USHORT          Unused;
    USHORT          Port;
} RSVP_FILTERSPEC_V4, *LPRSVP_FILTERSPEC_V4;

typedef struct _RSVP_FILTERSPEC_V6 {
    IN_ADDR_IPV6    Address;
    USHORT          UnUsed;
    USHORT          Port;
} RSVP_FILTERSPEC_V6, *LPRSVP_FILTERSPEC_V6;

typedef struct _RSVP_FILTERSPEC_V6_FLOW {
    IN_ADDR_IPV6    Address;
    UCHAR           UnUsed;
    UCHAR           FlowLabel[3];
} RSVP_FILTERSPEC_V6_FLOW, *LPRSVP_FILTERSPEC_V6_FLOW;

typedef struct _RSVP_FILTERSPEC_V4_GPI {
    IN_ADDR_IPV4    Address;
    ULONG           GeneralPortId;
} RSVP_FILTERSPEC_V4_GPI, *LPRSVP_FILTERSPEC_V4_GPI;

typedef struct _RSVP_FILTERSPEC_V6_GPI {
    IN_ADDR_IPV6    Address;
    ULONG           GeneralPortId;
} RSVP_FILTERSPEC_V6_GPI, *LPRSVP_FILTERSPEC_V6_GPI;
```

RSVP Adspec

The *RSVP_ADSPEC* object defines the information carried in the RSVP Adspec. This RSVP object typically indicates which service types are available (controlled load or guaranteed), whether a non-

RSVP hop has been encountered by the PATH message, and the minimum MTU along the path. The structure is defined as

```
typedef struct _RSVP_ADSPEC
{
    QOS_OBJECT_HDR     ObjectHdr;
    AD_GENERAL_PARAMS  GeneralParams;
    ULONG              NumberOfServices;
    CONTROL_SERVICE    Services[1];
} RSVP_ADSPEC, *LPRSVP_ADSPEC;
```

The first field of interest is *GeneralParams*, which is a structure of type *AD_GENERAL_PARAMS*. This structure is exactly as it sounds—it defines some general characterization parameters. The definition of this object is

```
typedef struct _AD_GENERAL_PARAMS
{
    ULONG       IntServAwareHopCount;
    ULONG       PathBandwidthEstimate;
    ULONG       MinimumLatency;
    ULONG       PathMTU;
    ULONG       Flags;
} AD_GENERAL_PARAMS, *LPAD_GENERAL_PARAMS;
```

The *IntServAwareHopCount* is the number of hops that conform to Integrated Services (IntServ) requirements. *PathBandwidthEstimate* is the minimum bandwidth available from sender to receiver. *MinimumLatency* is the sum of minimum latencies, in microseconds, of the packet forwarding processes in the routers. *PathMTU* is the maximum transmission unit—end-to-end—that will not incur any fragmentation. The *Flags* field is not used anymore.

RSVP Policy Info

The last provider object we'll take a look at is the RSVP policy info. This object is rather nebulous—it contains any number of policy elements from RSVP that are not defined. The structure is defined as

```
typedef struct _RSVP_POLICY_INFO  {
    QOS_OBJECT_HDR     ObjectHdr;
    ULONG              NumPolicyElement;
    RSVP_POLICY        PolicyElement[1];
} RSVP_POLICY_INFO, *LPRSVP_POLICY_INFO;
```

The *NumPolicyElement* field gives the number of *RSVP_POLICY* structures present in the *PolicyElement* array. This structure is defined as

```
typedef struct _RSVP_POLICY {
    USHORT  Len;
    USHORT  Type;
    UCHAR   Info[4];
} RSVP_POLICY, *LPRSVP_POLICY;
```

The *RSVP_POLICY* structure is data transported by RSVP on behalf of the policy component and is not particularly relevant to our needs.

Programming QOS

The programming techniques described in this section apply primarily to the Windows 98, Windows Me, and Windows 2000 platforms. As we mentioned, Windows XP does not feature a fully capable IP QOS service provider. Therefore, the Windows XP platforms no longer feature a supported RSVP signaling and admission control layer. The traffic control IP packet scheduler layer is still available and the technique of setting up a *FLOWSPEC* still applies, but information regarding QOS notifications and RSVP signaling does not apply.

Central to QOS is the initiation of an RSVP session. It's not until the RSVP PATH and RESV messages have been sent and processed that bandwidth is reserved for the process. Knowing when RSVP messages are generated is important to applications. For senders, three parameters must be known before a PATH message is generated:

- Sending *FLOWSPEC* member
- Source IP address and port
- Destination IP address, port, and protocol

The *FLOWSPEC* member is known whenever a QOS-enabled function is called, such as *WSAConnect*, *WSAJoinLeaf*, or *WSAIoctl* (with the *SIO_SET_QOS* option). The source IP address and port will not be known until the socket is bound locally, either implicitly (such as by connecting) or explicitly by bind. Finally, the application needs the data's destination. This information is gathered either through a connect call or, in the case of connectionless UDP, by setting the *QOS_DESTADDR* object in the provider-specific data passed using the *SIO_SET_QOS* ioctl command.

Similarly, for an RSVP RESV message to be generated, three things must be known:

- Receiving *FLOWSPEC* member
- Address and port of each sender
- Local address and port of the receiving socket

The receiving *FLOWSPEC* member is obtained from any of the QOS-enabled Winsock functions. The address and port of each sender depend on the filter style, which can be set manually via the *RSVP_RESERVE_INFO* provider-specific structure, discussed earlier. Otherwise, this information is obtained from a PATH message. Of course, depending on the socket type, it is not always necessary to have already received a PATH statement to get the sender's address to generate RESV messages. The wildcard filter style used in multicasting is an example of this. The RESV message sent applies to all senders in the session. The local address and port are self-explanatory for unicast and UDP receivers but not for multicast receivers. With multicast receivers, the local address and port are the multicast address and its corresponding port number.

In this section, we'll cover the different socket types and their interaction with the QOS service provider and RSVP messages. Then we'll take a look at how the QOS service provider notifies applications of certain events. Understanding these concepts is central to writing successful QOS-enabled applications. Programming such applications is a matter of knowing how to obtain QOS guarantees as well as knowing when those guarantees are put into effect and when and how they can change.

RSVP and Socket Types

You now have a basic understanding of how PATH and RESV RSVP messages are generated. In the following sections, we'll look at the different types of sockets—UDP, TCP, and multicast UDP—and how they interact with the QOS service provider to generate PATH and RESV messages.

Unicast UDP

Because you have the option of using either connected or unconnected UDP sockets, setting QOS on unicast UDP sockets presents quite a few options. With the UDP sender, the sending *FLOWSPEC* is obtained from one of the QOS-invoking functions. The local address and port are obtained either from an explicit bind call or from an implicit bind done by *WSAConnect*. The last piece is the address and port of the receiving application, which can be specified either in *WSAConnect* or via the *QOS_DESTADDR* provider-specific structure passed through the *SIO_SET_QOS* option. Be aware that if *SIO_SET_QOS* is used to set QOS, the socket must be bound beforehand.

For the UDP receiver, *WSAConnect* can be called to limit the receiving application to a single sender. In addition, applications can specify a *QOS_DESTADDR* structure with the *SIO_SET_QOS* ioctl command. Otherwise, the *SIO_SET_QOS* can be called without providing any kind of destination address. In this case, an RESV message will be generated with the wildcard filter style. In fact, specifying the destination address via *WSAConnect* or via the *QOS_DESTADDR* structure should be done only if you want the application to receive data from only one sender who uses the fixed filter style.

The UDP receiver can actually call both *WSAConnect* and the *SIO_SET_QOS* ioctl command in any order. If *SIO_SET_QOS* is called before *WSAConnect*, an RESV message is created with the wildcard filter first. Once the connect call is made, the previous RESV session is torn down and a new one is generated with the fixed filter style. Alternatively, calling *SIO_SET_QOS* after *WSAConnect* and a fixed filter RESV message does not negate the RSVP session and generate a wildcard filter style. Instead, it simply updates the QOS parameters associated with the existing RSVP session.

Unicast TCP

TCP sessions have two possibilities. First, the sender can be the client who connects to the server and sends data. The second possibility is that the server that the client connects to might be the sender. With the client, QOS parameters can be specified directly in the *WSAConnect* call, which will result in PATH messages being sent. The ioctl command *SIO_SET_QOS* can also be called before calling connect, but until one of the connect calls knows the destination address, no PATH messages will be generated.

When the sender is the server, the server calls *WSAAccept* to accept the client connection. This function does not provide a means of setting QOS on the accepted socket. If QOS is set before a call to *WSAAccept* by using *SIO_SET_QOS*, any accepted socket inherits the QOS levels set on the listening socket. Note that if the sender uses the conditional function in *WSAAccept*, the function should pass QOS values set on the connecting client. However, this is not the case. The QOS service provider passes junk, which is the behavior in Windows 98, Windows Me, and Windows 2000. The exception is that if the *lpSQOS* parameter is non-null under Windows 98 and Windows Me, some kind of QOS values must be set via the *SIO_SET_QOS* ioctl command within the conditional function; otherwise, the *WSAAccept* call fails even if *CF_ACCEPT* is returned. QOS can also be set on the client socket after it has been accepted.

Let's look at receiving TCP applications. The first case calls *WSAConnect* with a receiving *FLOWSPEC*. When this occurs, the QOS service provider creates an RESV request. If QOS parameters are not supplied to *WSAConnect*, the *SIO_SET_QOS* ioctl command can be set at a later time (resulting in an RESV message). The last combination is the server being the receiver, which is similar to the sending case. QOS can be set on the listening socket before a *WSAAccept* call, in which case the client socket inherits the same QOS levels. Otherwise, QOS can be set in the conditional function or after the socket has been accepted. In either case, the QOS service provider generates an RESV message as soon as a PATH message arrives.

Multicast

Multicast senders behave the same way as UDP senders except that *WSAJoinLeaf* is used to become a member of the multicast group, as opposed to calling *WSAConnect* with the destination address. QOS can be set with *WSAJoinLeaf* or separately through an *SIO_SET_QOS* call. The multicast session address is used to compose the RSVP session object included in the RSVP PATH message.

With the multicast receiver, no RESV messages will be generated until the multicast address is specified via the *WSAJoinLeaf* function. Because the multicast receiver doesn't specify a peer address, the QOS provider generates RESV messages with the wildcard filter style. The QOS service provider does not prohibit a socket from joining multiple multicast groups. In this case, the service provider sends RESV messages for all groups that have a matching PATH message. The QOS parameters supplied to each *WSAJoinLeaf* will be used in each RESV message, but if *SIO_SET_QOS* is called on the socket after joining multiple groups, the new QOS parameters will be applied to all multicast groups joined.

When a sender sends data to a multicast group, only data sent to the multicast group that the sender joined results in QOS being applied to that data. In other words, if you join one multicast group and use *sendto/WSASendTo* with any other multicast group as the destination, QOS is not applied to that data. In addition, if a socket joins a multicast group specifying a particular direction (for example, using *JL_SENDER_ONLY* or *JL_RECEIVER_ONLY* in the *dwFlags* parameter to *WSAJoinLeaf*), QOS is applied accordingly. A socket set as a receiver only will not gain any QOS benefits for sent data.

QOS Notifications

Thus far, you have learned how to invoke QOS for TCP, UDP, and multicast UDP sockets and the corresponding RSVP events that occur depending on whether you're sending or receiving. However, the completion of these RSVP messages is not strictly tied to the API calls that invoke them. That is, issuing a *WSAConnect* call for a TCP receiving socket generates an RESV message, but the RESV message is independent of that API call because the call returns without any assurances that the reservation is approved and network resources are allocated. Because of this, a new asynchronous event has been added, *FD_QOS*, which is posted to a socket. Typically, an *FD_QOS* event notification will be posted in the following events:

- Notification of the acceptance or rejection of the application's QOS request
- Significant changes in the QOS that is provided by the network (as opposed to previously negotiated values)
- Status regarding whether a QOS peer is ready to send or receive data for a particular flow

Registering for *FD_QOS* Notifications

To take advantage of these notifications, an application must register to be notified when an *FD_QOS* event occurs. You can do this in two different ways. First, you can use either *WSAEventSelect* or *WSAAsyncSelect* and include the *FD_QOS* flag in the bitwise *OR*ing of event flags. However, an application is eligible to receive the *FD_QOS* event only if a call has already been made to one of the QOS–invoking functions. Note that in some cases an application might want to receive the *FD_QOS* event without having to set QOS levels on a socket. This can be accomplished by setting up a *QOS* structure whose sending and receiving *FLOWSPEC* members contain either the *QOS_NOT_SPECIFIED* or the *SERVICETYPE_NOTRAFFIC* flag. The only catch is that the *SERVICE_NO_QOS_SIGNALING* flag must be *OR*ed with the *SERVICETYPE_NOTRAFFIC* flag for the direction of QOS in which you want to receive event notification.

If you need exact information on how to call the two asynchronous select functions, consult Chapter 5, which covers them in great detail. If you use *WSAEventSelect* once the event has been triggered, you should call the *WSAEnumNetworkEvents* function to obtain additional status codes that might be available. Simply pass the socket handle, the event handle, and a *WSANETWORKEVENTS* object into the call, which will return and set event information into the supplied structure.

RSVP Notifications

We mentioned earlier that there are a couple of ways to receive QOS notifications. This information actually ties into this section: obtaining the results of a QOS event. If you have registered to receive *FD_QOS* notifications with either *WSAAsyncSelect* or *WSAEventSelect* and you actually receive an *FD_QOS* event notification, you must perform a call to *WSAIoctl* with the *SIO_GET_QOS* ioctl option to find out what triggered the event. You don't have to register for *FD_QOS* events—you can simply call *WSAIoctl* with the *SIO_GET_QOS* command using overlapped I/O. This also requires that you specify a completion routine, which is invoked once the QOS service provider detects a change in QOS. Once the callback occurs, a QOS structure will be available in the output buffer.

In either case, once a change in QOS has occurred, your application can be notified of this change by registering for *FD_QOS* or by using overlapped I/O and *SIO_GET_QOS*. If you register for *FD_QOS*, call *WSAIoctl* with the *SIO_GET_QOS* ioctl command upon event notification. For both methods, the QOS structure returned contains QOS information for only a single direction. That is, the *FLOWSPEC* structure for the invalid direction has its *ServiceType* field set to *SERVICETYPE_NOCHANGE*. In addition, more than one QOS event might have occurred, in which case you should call *WSAIoctl* and *SIO_GET_QOS* in a loop until *SOCKET_ERROR* is returned and *WSAGetLastError* returns *WSAEWOULDBLOCK*. The final concern when calling *SIO_GET_QOS* is the buffer size. When an *FD_QOS* event has been triggered, it is possible that provider-specific objects will be returned. In fact, the *RSVP_STATUS_INFO* structure will most often be returned, provided the buffer is large enough. See the earlier entry on *WSAIoctl* for information about how to find the right-size buffer.

If your application uses one of the asynchronous event functions, a particularly important issue is that once an *FD_QOS* event occurs, you must ***always*** perform an *SIO_GET_QOS* operation to re-enable *FD_QOS* notifications.

You now know how to receive QOS event notifications and obtain new QOS parameters as a result of these events, but what types of notifications will occur? The first and most obvious reason for a QOS event is a change in the *FLOWSPEC* parameters for a given flow. For example, if you set up a socket with best-effort service, periodically, the QOS service provider will send notification to your application indicating the current conditions on the network. In addition, if you specify controlled load as well as other parameters, the QOS parameters for token bucket size and token rate might change slightly from what you requested once the reservation occurs. Your application should compare the *FLOWSPEC* returned once a QOS notification occurs to what you originally requested to ensure that it is sufficient for your application to continue. Also, remember that throughout the life of a QOS-enabled socket, you can always perform a *SIO_SET_QOS* to change any of the parameters, which will result in a QOS notification for the peer or peers associated with your current RSVP session. A robust application should be able to handle these conditions.

In addition to updating QOS parameters, QOS event notification signals other occurrences, such as notification of senders or receivers. The possible events are listed in Table 10-5. There are two ways to obtain these status codes. The first is as a part of the *RSVP_STATUS_INFO* object. When a QOS event occurs and a call is made to *SIO_GET_QOS*, it is possible that an *RSVP_STATUS_INFO* object will be returned as part of the provider-specific buffer. Second, if you use *WSAEventSelect* to register for events, these codes can be returned in the *WSANETWORKEVENTS* structure returned from *WSAEnumNetworkEvents*. The codes defined in Table 10-5 can be found in the *iErrorCode* array, indexed by *FD_QOS_BIT*. The first five codes listed are not error codes. They return valuable information concerning the status of the QOS connection. The other status codes listed in the table are QOS errors of concern, but they won't prevent you from sending and receiving data—they merely indicate an error in the QOS session. Of course, data sent in this situation will not carry any of the requested QOS guarantees.

WSA_QOS_RECEIVERS and *WSA_QOS_NO_RECEIVERS*

With unicast, after a sender starts up and receives the first RESV message, a *WSA_QOS_RECEIVERS* is passed up to the application. If the receiver performs any steps to disable QOS, the result is an *RESV* teardown message. Once the sender receives this, *WSA_QOS_NO_RECEIVERS* is passed up to the application. Of course, with unicast many receivers simply close the socket, generating both an *FD_CLOSE* event and the *WSA_QOS_NO_RECEIVERS* event. In most cases, an application's response is simply to close the sending socket.

With multicast, the sending application receives *WSA_QOS_RECEIVERS* whenever the number of receivers changes and is nonzero. A single multicast sender receives *WSA_QOS_RECEIVERS* every time a QOS receiver joins the group, as well as every time a receiver drops out of a group—as long as at least one receiver remains.

WSA_QOS_SENDERS and *WSA_QOS_NO_SENDERS*

The senders notification is similar to the receivers event except that it deals with the receipt of the PATH message. For unicast receivers after startup, the receipt of the first PATH message generates *WSA_QOS_SENDERS*, while the PATH teardown message initiates a *WSA_QOS_NO_SENDERS* message.

Likewise, multicast receivers receive the *WSA_QOS_SENDERS* notification whenever the number of senders decrements or increments and is nonzero. Once the number of senders reaches 0, the *WSA_QOS_NO_SENDERS* message is passed to the application.

WSA_QOS_REQUEST_CONFIRMED

This last status message is issued to receiving QOS applications if they ask to be notified when a reservation request has been confirmed. If it is set to nonzero, a field within the *RSVP_STATUS_INFO* structure named *ConfirmRequest* informs the QOS service provider to notify the application when the reservation request has been confirmed. This object is a provider-specific option that can be passed along with a QOS structure to the *SIO_SET_QOS* ioctl command.

QOS Templates

Winsock provides several predefined QOS structures, referred to as templates, that an application can query by name. These templates define the QOS parameters for some common audio and video codecs, such as G711 and H263QCIF. The function *WSAGetQOSByName* is defined as

```
BOOL WSAGetQOSByName(
    SOCKET s,
    LPWSABUF lpQOSName,
    LPQOS lpQOS
);
```

If you don't know the name of the installed templates, you can use this function to first enumerate all template names. To do this, provide a sufficiently large buffer in *lpQOSName* with its first character set to the null character and pass a null pointer for *lpQOS*, as in the following code:

```
WSABUF  wbuf;
char    cbuf[1024];

cbuf[0] = '\0';
wbuf.buf = cbuf;
wbuf.len = 1024;
WSAGetQOSByName(s, &wbuf, NULL);
```

Upon return, the character buffer is filled with an array of strings separated by a null character, and the entire list is terminated by another null character. As a result, the last string entry will have two consecutive null characters. From here you can get the names of all of the installed templates and query for a specific one. The following code looks up the G711 template:

```
QOS     qos;
WSABUF  wbuf;

wbuf.buf = "G711";
wbuf.len = 4;
WSAGetQOSByName(s, &wbuf, &qos);
```

If the requested QOS template does not exist, the lookup returns *FALSE* and the error is *WSAEINVAL*. Upon success, the function returns *TRUE*.

The example TEMPLATE.CPP on the accompanying CD illustrates how to enumerate th installed QOS templates.

In addition, you can install your own QOS template so that other applications can query for it by name. Two functions do this: *WSCInstallQOSTemplate* and *WSCRemoveQOSTemplate*. The first one installs a QOS template, and the second removes it. The prototypes are

```
BOOL WSCInstallQOSTemplate(
    const LPGUID lpProviderId,
    LPWSABUF lpQOSName,
    LPQOS lpQOS
);

BOOL WSCRemoveQOSTemplate(
    const LPGUID lpProviderId,
    LPWSABUF lpQOSName
);
```

These two functions are fairly self-explanatory. To install a template, call *WSCInstallQOSTemplate* with a GUID, the name of the template, and the QOS parameters. The GUID is a unique identifier for this template that utilities such as UUIDGEN.EXE can generate. To remove the template, simply supply the template name along with the same GUID used in the installation process to *WSCRemoveQOSTemplate*. Both functions return *TRUE* when successful.

Examples

In this section, we'll take a look at two programming examples of using QOS over TCP and UDP. The first example, which uses TCP, will demonstrate how to set up a *FLOWSPEC* and manage RSVP signaling on the Windows 98, Windows Me, and Windows 2000 platforms. The second example describes UDP and is primarily designed for Windows XP, which demonstrates only how to set up a *FLOWSPEC* to invoke the IP packet scheduler. The examples rely on a couple of support routines, *PrintQos* and *FindProtocolInfo*, which are defined in the files PRINTQOS.CPP and PROVIDER.CPP, respectively, both of which can be found on the companion CD. The former routine simply prints out the contents of a QOS structure, while the latter finds a protocol from the provider catalog with the required attributes, such as QOS.

TCP

The code for the TCP example can be found in a file on the companion CD called QOSTCP.CPP. The example is a bit long, but not particularly complicated. Most of the code is nothing more than the usual *WSAEventSelect* code that we introduced in Chapter 5. The only exception is what we do with an *FD_QOS* event. The main function doesn't do anything out of the ordinary. The arguments are parsed, a socket is created, and either the *Server* or the *Client* function is called—depending on whether the application is called as the server or the client. Let's examine the client connection first.

The sample has a command line parameter that tells the example to set QOS before connection, during connection, after connection, or after the peer requests QOS to set QOS locally. If QOS is selected to be set before connection (for the client), bind the socket to a random port and then call *SIO_SET_QOS* with a sending QOS *FLOWSPEC*. Note that it isn't really necessary to bind before calling *SIO_SET_QOS* because the peer's address is not known until a connect call is made, and an RSVP session cannot be initiated until the peer's address is known.

If the user elects to set QOS during connection, the example code passes the QOS structure into the *WSAConnect* call. The following code demonstrates how the sample optionally sets up QOS before or during the connection phase:

```
int  iSetQos;
SOCKET s;
char szServerAddr[32];
. . .
const FLOWSPEC flowspec_notraffic = {QOS_NOT_SPECIFIED,
                                      QOS_NOT_SPECIFIED,
                                      QOS_NOT_SPECIFIED,
                                      QOS_NOT_SPECIFIED,
                                      QOS_NOT_SPECIFIED,
                                      SERVICETYPE_NOTRAFFIC,
                                      QOS_NOT_SPECIFIED,
                                      QOS_NOT_SPECIFIED};

const FLOWSPEC flowspec_g711 = {8500,
                                680,
                                17000,
                                QOS_NOT_SPECIFIED,
                                QOS_NOT_SPECIFIED,
                                SERVICETYPE_CONTROLLEDLOAD,
                                340,
                                340};

QOS  clientQos;        // QOS client structure
QOS  *lpqos;
SOCKADDR_IN server,
SOCKADDR_IN local;
DWORD dwBytes;
. . .

// Set up the client's QOS flowspec

clientQos.SendingFlowspec = flowspec_g711;
clientQos.ReceivingFlowspec =  flowspec_notraffic;
clientQos.ProviderSpecific.buf = NULL;
clientQos.ProviderSpecific.len = 0;

if (iSetQos == SET_QOS_BEFORE)
{
   lpqos = NULL;

   // Bind to the local interface and provide a flowspec

   local.sin_family = AF_INET;
   local.sin_port = htons(0);
   local.sin_addr.s_addr = htonl(INADDR_ANY);

   bind(s, (SOCKADDR *)&local, sizeof(local));

   WSAIoctl(s, SIO_SET_QOS, &clientQos, sizeof(clientQos),
            NULL, 0, &dwBytes, NULL, NULL);
}
else if (iSetQos == SET_QOS_DURING)
{
   // Use the QOS structure during connect
   lpqos = &clientQos;
}
else if (iSetQos == SET_QOS_EVENT)
{
   lpqos = NULL;

   // Set QOS later when signaled from a peer

   clientQos.SendingFlowspec.ServiceType |= SERVICE_NO_QOS_SIGNALING;
   clientQos.ReceivingFlowspec.ServiceType |= SERVICE_NO_QOS_SIGNALING;

   WSAIoctl(s, SIO_SET_QOS, &clientQos, sizeof(clientQos), NULL, 0,
            &dwBytes, NULL, NULL);
}

server.sin_family = AF_INET;
server.sin_port = htons(5150);
server.sin_addr.s_addr = inet_addr(szServerAddr);
```

```
printf("Connecting to: %s\n", inet_ntoa(server.sin_addr));

WSAConnect(s, (SOCKADDR *)&server, sizeof(server), NULL, NULL,
           lpqos, NULL);

. . .
```

The *WSAConnect* call initiates an RSVP session and connects the client to the specified server. Otherwise, the user specifies that the sample should wait for the peer to set QOS, and no QOS structure is passed to *WSAConnect*. Instead, the code takes the sending QOS structure, *ORs* in the *SERVICE_NO_QOS_SIGNALING* flag to the *ServiceType* field in the *FLOWSPEC* structures, and calls *WSAIoctl* with the *SIO_SET_FLAG* ioctl command. This tells the QOS service provider not to invoke TC but to still look for RSVP messages.

After QOS is set, the events that the client wants to be notified of are registered, including *FD_QOS*. Notice that QOS must be set on the socket beforehand for the application to request receiving *FD_QOS*. Once this occurs, the client waits in a loop on *WSAWaitForMultipleEvents*, which unblocks when one of the selected events is signaled. Once an event occurs, the events are then enumerated along with any errors in *WSAEnumNetworkEvents*. The following code fragment demonstrates how the sample handles *FD_QOS* events through *WSAEventSelect:*

```
wbuf.buf = databuf;
wbuf.len = DATA_BUFFER_SZ;

memset(databuf, '#', DATA_BUFFER_SZ);
databuf[DATA_BUFFER_SZ-1] = 0;

while (1)
{
   ret = WSAWaitForMultipleEvents(1, &hEvent, FALSE,
            WSA_INFINITE, FALSE);
   if (ret == WSA_WAIT_FAILED)
   {
      printf("WSAWaitForMulipleEvents() failed: %d\n",
             WSAGetLastError());
      return;
   }

   WSAEnumNetworkEvents(s, hEvent, &ne);

   if (ne.lNetworkEvents & FD_READ)
   {
      // Read notification occurred
   }
   if (ne.lNetworkEvents & FD_WRITE)
   {
      if (ne.iErrorCode[FD_WRITE_BIT])
         printf("FD_WRITE error: %d\n", ne.iErrorCode[FD_WRITE_BIT]);
      else
         printf("FD_WRITE\n");

      if (!bWaitToSend)
      {
```

```
      wbuf.buf = databuf;
      wbuf.len = DATA_BUFFER_SZ;
      //
      // If the network can't support the bandwidth don't send
      //
      if (!AbleToSend(s))
      {
         printf("Network is unable to provide "
                "sufficient best effort bandwidth\n");
         printf("before the reservation "
                "request is approved\n");
      }


      WSASend(s, &wbuf, 1, &dwBytesSent, 0, NULL, NULL);
      printf("Sent: %d bytes\n", dwBytesSent);
   }
}

if (ne.lNetworkEvents & FD_CLOSE)
{
   // Close notification occurred
}

if (ne.lNetworkEvents & FD_QOS)
{
   char        buf[QOS_BUFFER_SZ];
   QOS        *lpqos = NULL;
   DWORD       dwBytes;
   BOOL        bRecvRESV = FALSE;

   if (ne.iErrorCode[FD_QOS_BIT])
   {
      printf("FD_QOS error: %d\n", ne.iErrorCode[FD_QOS_BIT]);
      if (ne.iErrorCode[FD_QOS_BIT] == WSA_QOS_RECEIVERS)
          bRecvRESV = TRUE;
   }
   else
      printf("FD_QOS\n");

   lpqos = (QOS *)buf;
   WSAIoctl(s, SIO_GET_QOS, NULL, 0,
            buf, QOS_BUFFER_SZ, &dwBytes, NULL, NULL);
   //
   // Check to see if there is a status object returned
   // in the QOS structure which may also contain the
   // WSA_QOS_RECEIVERS flag
   //
   if (ChkForQosStatus(lpqos, WSA_QOS_RECEIVERS))
      bRecvRESV = TRUE;

   if (iSetQos == SET_QOS_EVENT)
   {
      lpqos->SendingFlowspec.ServiceType =
                 clientQos.SendingFlowspec.ServiceType;
       WSAIoctl(s, SIO_SET_QOS, lpqos, dwBytes,
                NULL, 0, &dwBytes, NULL, NULL);
         //
         // Change iSetQos so we don't set QOS again if we
         // receive another FD_QOS event
         //
         iSetQos = SET_QOS_BEFORE;
      }

      if (bWaitToSend && bRecvRESV)
      {
         wbuf.buf = databuf;
         wbuf.len = DATA_BUFFER_SZ;

         WSASend(s, &wbuf, 1, &dwBytesSent, 0, NULL, NULL);
         printf("Sent: %d bytes\n", dwBytesSent);
      }
   }
}
```

For the most part, QOSTCP.CPP handles the other events, such as *FD_READ*, *FD_WRITE*, and *FD_CLOSE*, the same way as the *WSAEventSelect* example code in Chapter 5. The only item of note is in the *FD_WRITE* event. One of the command line options is to wait until an RSVP PATH message has been received before sending the data. This is especially relevant if the data being transmitted is likely to exceed the best-effort bandwidth available on the network. The *AbleToSend* function calls *SIO_CHK_QOS* to determine if the QOS parameters requested are within the available best-effort limits. If so, it should be OK to start sending data; otherwise, wait for a confirmation to send data.

In our client's case, we want to receive the *WSA_QOS_RECEIVERS* message to indicate the receipt of an *RESV* message. This can be indicated upon receipt of an *FD_QOS* event. At this point, we call the *SIO_CHK_QOS* command to obtain status information. This *WSA_QOS_RECEIVERS* flag can be returned in two ways. First, the flag can be returned in the *iErrorCode* field of the *WSANETWORKEVENTS* structure as the element indexed by *FD_QOS_BIT*. Second, an *RSVP_STATUS_INFO* structure can be returned in the buffer passed to *WSAIoctl* using the *SIO_GET_QOS* ioctl command. This structure also might contain the *WSA_QOS_RECEIVERS* flag in its *StatusCode* field. If the wait to send flag has been specified, we check the error field from *WSANETWORKEVENTS* to see if an *RSVP_STATUS_INFO* structure has been returned. If the flag is present, we send data. That's all! The code necessary to support QOS for the client is straightforward.

The server side of the example is a bit more complicated, but only because it needs to manage zero or more client connections. The listening socket and the client sockets are handled in a single array, *sc*. Array element 0 is the listening socket and the rest are possible client connections. The global variable *nConns* contains the number of current clients. Whenever a client connection finishes, all active sockets are compacted toward the beginning of the socket array. There is also a corresponding array of event handles.

The server first binds the listening socket and sets receiving QOS if the user chooses to set QOS before accepting client connections. Any QOS parameters set on the listening socket are copied to the client connection (unless the server is using *AcceptEx*). The listening socket registers to receive only *FD_ACCEPT* events. The rest of the server routine is a loop that waits for events on the array of socket handles. At first, the only socket in the array is the listening socket, but as more client connections are established there will be more sockets and their corresponding events. If *WSAWaitForMultipleEvents* unblocks as a result of an event and indicates the event handle in array element 0, the event is occurring on the listening socket. If so, the code will call *WSAEnumNetworkEvents* to find out which event is occurring. If the event is occurring on a client socket, the code calls the handler routine *HandleClientEvents*.

On the listening socket, the event of interest is *FD_ACCEPT*. When this event happens, *WSAAccept* is called with a conditional function. Remember that the QOS parameters passed into the conditional function can't be trusted, and if the QOS parameter is non-null in Windows 98 and Windows Me, some sort of QOS must be set. Windows 2000 does not have that limitation—QOS can be set at any time. If the user specifies that QOS be set during the accept call, this occurs within the conditional function. Once the client socket is accepted, a corresponding event handle is created and the appropriate events are registered for the socket.

The function *HandleClientEvents* handles any events occurring on the client sockets. The read and write events are straightforward—the only exception is whether to wait for the reservation confirmation before sending. If the user specifies to wait for the reservation confirmation to arrive before sending data, the client waits for the *WSA_QOS_RECEIVERS* message to be returned in an *FD_QOS* event. If the message returns, the sending of the data doesn't occur until *FD_QOS* is received. Usually, the most significant aspect of this example is setting QOS on the socket and the *FD_QOS* handler.

UDP

The last sample provided, QOSUDP.CPP, demonstrates how to set up QOS over UDP and invoke the IP packet scheduler service without using RSVP signaling. This sample is written primarily for Windows XP (although it will work on other Windows platforms that support QOS) because RSVP signaling is no longer available.

The sample is just a sender that transmits datagrams to a specified receiving host that receives datagrams on port 5150. QOS is set up on the socket before datagrams are transmitted by calling *SIO_SET_QOS* with a *QOS_DESTADDR* object. An important part to note is the flag *SERVICE_NO_QOS_SIGNALING* is *OR*ed in with the *ServiceType FLOWSPEC* field so this application will behave the same on all Windows platforms regardless of whether RSVP signaling is available.

ATM and QOS

Windows 98 (with Service Pack 1), Windows Me, Windows 2000, and Windows XP support ATM programming from Winsock. QOS is natively available on an ATM network, which means that the network, application, and policy components that are necessary for QOS over IP are not required over ATM. This includes the Admission Control Service and the RSVP protocol. Instead, the ATM switch performs bandwidth allocations and prevents over-allocation of bandwidth.

In addition to the differences we've already mentioned, the Winsock API functions behave a bit differently with ATM QOS than they do with QOS over IP. The first major difference is that the QOS bandwidth request is handled as part of the connection request. This differs from QOS over IP in that the RSVP session is established separately from the connection. Also, if the bandwidth request is rejected under ATM, the connection will fail.

This leads to our next point: both of the native ATM providers are connection-oriented. As a result, you don't have the problem of setting QOS levels for a connectionless socket and then having to specify the endpoint for communication. The next major difference is that only one side sets the QOS parameters for a connection. If the client wants to set QOS on a connection, both the sending and receiving *FLOWSPEC* structures are set within the QOS structure passed to *WSAConnect*. These values will then be applied to the connection, in contrast to QOS over IP, in which the sender requests certain QOS levels and the receiver then makes the reservation. In addition, the listening socket might have QOS set using *WSAIoctl* and *SIO_SET_QOS*. These values will be applied to any incoming connections. This also means that QOS ***must*** be set during connection setup. You cannot set QOS on an established connection.

This leads us to our last point: once QOS is set for a connection, you cannot renegotiate it by calling *WSAIoctl* and *SIO_SET_QOS*. When QOS is set on a connection, it remains until the connection is closed.

Keep in mind that RSVP is not present and no signaling occurs. None of the status flags in Table 10-5 are ever generated. QOS is set when establishing the connection, and no further notifications or events occur until the connection is closed.

Conclusion

QOS offers powerful capabilities to applications that require a guaranteed level of network service. Setting up a QOS connection is rather involved, but don't let this scare you. The most important concept is learning how and when RSVP messages are generated so that you can code your application accordingly. Although the future of GQOS on Windows platforms is uncertain, there is still TC functionality in the latest Windows XP platforms.

Chapter 11

Raw Sockets

A raw socket is one that allows access to the underlying transport protocol. This chapter is dedicated to illustrating how raw sockets can be used to simulate IP utilities, such as Traceroute and Ping. Raw sockets can also be used to manipulate IP header information. This chapter is concerned with the IPv4 and IPv6 protocols only; we will not address raw sockets with any other protocol because most protocols (except ATM) do not support raw sockets. All raw sockets are created using the *SOCK_RAW* socket type and are currently supported only under Winsock 2. Therefore, neither Microsoft Windows CE nor Windows 95 (without the Winsock 2 update) can use raw sockets.

In addition, using raw sockets requires substantial knowledge of the underlying protocol structure, which is not the focus of this book. In this chapter, we will discuss ICMP, ICMPv6, and UDP. ICMP (both versions) is used by the Ping utility, which can detect whether a route to a host is valid and whether the host machine is responding. Developers often need a programmatic method of determining whether a machine is alive and reachable. We will also examine UDP in conjunction with the *IP_HDRINCL* socket option to send completely fabricated IP packets. For all of these protocols, we will cover only the aspects necessary to fully explain the code in this chapter and in the example programs. For more detailed information, consult W. Richard Stevens's book on IP, ***TCP/IP Illustrated, Volume 1*** (Addison-Wesley, 1994) or the individual RFCs for each protocol.

Raw Socket Creation

The first step in using raw sockets is creating the socket. You can use either socket or *WSASocket*. Note that for Windows 95, Windows 98, and Windows Me, no catalog entry in Winsock for IP has the *SOCK_RAW* socket type. However, this does not prevent you from creating this type of socket. It just means that you cannot create a raw socket using a *WSAPROTOCOL_INFO* structure. Refer back to Chapter 2 for information about enumerating protocol entries with the *WSAEnumProtocols* function and the *WSAPROTOCOL_INFO* structure. You must specify the *SOCK_RAW* flag yourself in socket creation. The following code snippet illustrates the creation of a raw socket using ICMP as the underlying IP protocol:

```
SOCKET    s;

s = socket(AF_INET, SOCK_RAW, IPPROTO_ICMP);
// Or
s = WSASocket(AF_INET, SOCK_RAW, IPPROTO_ICMP, NULL, 0,
    WSA_FLAG_OVERLAPPED);
if (s == INVALID_SOCKET)
{
    // Socket creation failed
}
```

When creating a raw socket, the protocol parameter of the socket call becomes the protocol value in the IP header. That is, if a raw *AF_INET6* socket is created with the protocol value 66, the IPv6 header for outgoing packets will contain the value 66 in the next header field.

Because raw sockets offer the capability to manipulate the underlying transport, they can be used for malicious purposes and are a security issue in Windows NT. Therefore, only members of the Administrators group can create sockets of type *SOCK_RAW*. Anyone can create a raw socket on Windows NT, but non-Administrators will not be able to do anything with it because the *bind* API will fail with WSAEACCES. Windows 95, Windows 98, and Windows Me do not impose any kind of limitation.

To work around this limitation on Windows NT, you can disable the security check on raw sockets by creating the following registry variable and setting its value to the integer 1 as a *DWORD* type.

```
HKEY_LOCAL_MACHINE\System\CurrentControlSet
    \Services\Afd\Parameters\DisableRawSecurity
```

After the registry change, you need to reboot the machine.

In the socket creation code in the example, we used the ICMP protocol, but you can also use IGMP, UDP, IP, or raw IP using the flags *IPPROTO_IGMP*, *IPPROTO_UDP*, *IPPROTO_IP*, or *IPPROTO_RAW*, respectively. However, be aware that on Windows 95 (with Winsock 2), Windows 98, and Windows NT 4, you can use only IGMP and ICMP when creating raw sockets. The protocol flags *IPPROTO_UDP*, *IPPROTO_IP*, and *IPPROTO_RAW* require the use of the socket option *IP_HDRINCL*, which is not supported on those platforms. Windows Me and Windows 2000 and later versions support *IP_HDRINCL*, so it is possible to manipulate the IP header (*IPPROTO_RAW*), the TCP header (*IPPROTO_TCP*), and the UDP header (*IPPROTO_UDP*).

Once the raw socket is created with the appropriate protocol flags, you can use the socket handle in send and receive calls. When creating raw sockets, the IP header will be included in the data returned upon any receive, regardless of whether the *IP_HDRINCL* option is set. Applications will have to know the layout of the IP header and have to determine the length of the IP header to find the payload data within the received buffer.

ICMP

ICMP is used as a means of messaging between hosts. Also, there are two versions of ICMP. The original ICMP is used with IPv4 to pass informational messages between two hosts, usually relating to communications errors, such as destination unreachable or TTL exceeded. With IPv6, a new version of ICMP was created: ICMPv6. ICMPv6 includes the informational messages but also incorporates ND and MLD. As we discussed in Chapter 3, ND is the IPv6 equivalent to ARP and MLD is equivalent to IGMP. Our discussion of both versions of ICMP is limited to the informational messages.

As we mentioned previously, ICMP uses IPv4 addressing because it is a protocol encapsulated directly within an IPv4 datagram. Figure 11-1 illustrates the layout of an ICMP message. ICMPv6 is encapsulated in an IPv6 datagram and is identical in structure to the ICMP packet (as least in terms of the first four bytes).

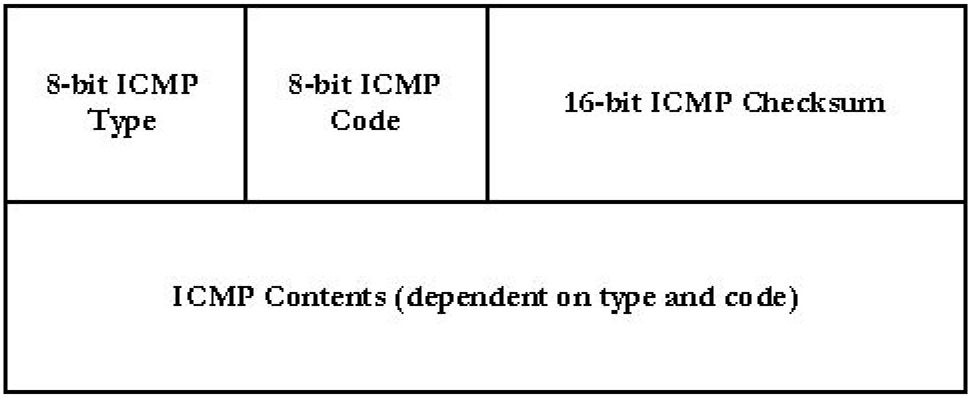


Figure 11-1 *ICMP header*

The first field is the ICMP message type, which is typically classified as either a query or an error. The code field further defines the type of query or message. The checksum field is the 16-bit one's complement sum of the ICMP header. Note that the checksum computation is different for IPv4 and IPv6. For IPv4, the checksum is calculated over the ICMP header and its payload only, and for ICMPv6, the checksum is calculated over the IPv6 pseudo-header followed by the ICMPv6 header and payload. The IPv6 pseudo-header is comprised of the following fields:

- 128-bit IPv6 source address
- 128-bit IPv6 destination address
- 32-bit upper layer protocol packet length
- 24-bit zeroed field
- 8-bit next header protocol value

IPv6 requires this pseudo-header calculation for any checksum calculation by an upper layer protocol that includes addresses from the IP header. This includes both UDP and ICMPv6. If the upper layer protocol contains its own packet length field, that value is used in the pseudo-header computation. Otherwise, the payload length from the IPv6 header is used, minus the size of all IPv6 extension headers present. Figure 11-5, later in this chapter, illustrates the IPv6 pseudo-header along with the UDP header and payload.

Finally, the ICMP contents depend on the ICMP type and code. Table 11-1 lists the most common types and codes for ICMP, and Table 11-2 lists the common types and codes for ICMPv6. The type and code of the ICMP packet dictates what is to follow the ICMP header.

Table 11-1 *ICMP Message Types*

Type	Query/Error (Error Type)	Code	Description
0	Query	0	Echo reply
3	Error: Destination unreachable	0	Network unreachable
		1	Host unreachable
		2	Protocol unreachable
		3	Port unreachable
		4	Fragmentation needed, but the Don't Fragment bit has been set
		5	Source route failed
		6	Destination network unknown
		7	Destination host unknown
		8	Source host isolated (obsolete)
3	Error: Destination unreachable	9	Destination network administratively prohibited
		10	Destination host administratively prohibited
		11	Network unreachable for TOS
		12	Host unreachable for TOS
		13	Communication administratively prohibited by filtering
		14	Host precedence violation
		15	Precedence cutoff in effect
4	Error	0	Source quench
5	Error: Redirect	0	Redirect for network
		1	Redirect for host
		2	Redirect for TOS and network
		3	Redirect for TOS and host
8	Query	0	Echo request
9	Query	0	Router advertisement
10	Query	0	Router solicitation
11	Error: Time exceeded	0	TTL equals 0 during transit
		1	TTL equals 0 during reassembly
12	Error: Parameter problem	0	IP header bad
		1	Required option missing

When an ICMP error message is generated, it always contains as much of the IP header and IP payload that caused the error to occur without exceeding the MTU size. This allows the host receiving the ICMP error to associate the message with one particular protocol and process associated with that error. In our case, Ping relies on the echo request and echo reply ICMP queries rather than on error messages. In the next section, we will discuss how to use ICMP with a raw socket to generate a Ping request by using the echo request and echo reply messages. If you require more information about ICMP errors or the other types of ICMP queries, consult more in-depth sources, such as Stevens's ***TCP/IP Illustrated, Volume 1***. Also, see RFCs 792 and 2463 for more information on ICMP and ICMPv6, respectively.

Table 11-2 *ICMPv6 Message Types*

Type	Query/Error (Error Type)	Code	Description
1	Error: Destination unreachable	0	No route to destination
		1	Communication with destination administratively prohibited
		3	Address unreachable
		4	Port unreachable
2	Error: Packet too big	0	Packet is larger than MTU size and cannot be forwarded
3	Error: Time exceeded	0	Hop limit exceeded in transit
		1	Fragment reassembly time exceeded
4	Error: Parameter problem	0	Erroneous header field encountered
		1	Unrecognized Next Header type encountered
		2	Unrecognized IPv6 option encountered
128	Query: Echo request	0	Request the destination to echo back the ICMP payload
129	Query: Echo reply	0	Reply to an echo request query

Ping Example

Ping is often used to determine whether a particular host is alive and reachable through the network. By generating an ICMP echo request and directing it to the host you are interested in, you can determine whether you can successfully reach that machine. Of course, this does not guarantee that a socket client will be able to connect to a process on that host (for example, a process on the remote server might not be listening); it just means that the network layer of the remote host is responding to network events. Finally, most operating systems offer the capability to turn off responding to ICMP echo requests, which is often the case for machines running firewalls. Essentially, the Ping example performs the following steps.

1. Creates a socket of address family AF_INET, type *SOCK_RAW*, and protocol *IPPROTO_ICMP*. For IPv6, the address family is AF_INET6, type *SOCK_RAW*, and protocol value 58.

2. Creates and initializes the ICMP header.
3. Calls *sendto* or *WSASendTo* to send the ICMP request to the remote host.
4. Calls *recvfrom* or *WSARecvFrom* to receive any ICMP responses.

Initializing the ICMP header is a straightforward task. First, the ICMP header is initialized with the type and code. Remember that the header is the same for ICMP and ICMPv6 (as shown in Figure 11-1). Following the type and code header, the echo request header must be supplied. This header is shown in Figure 11-2.

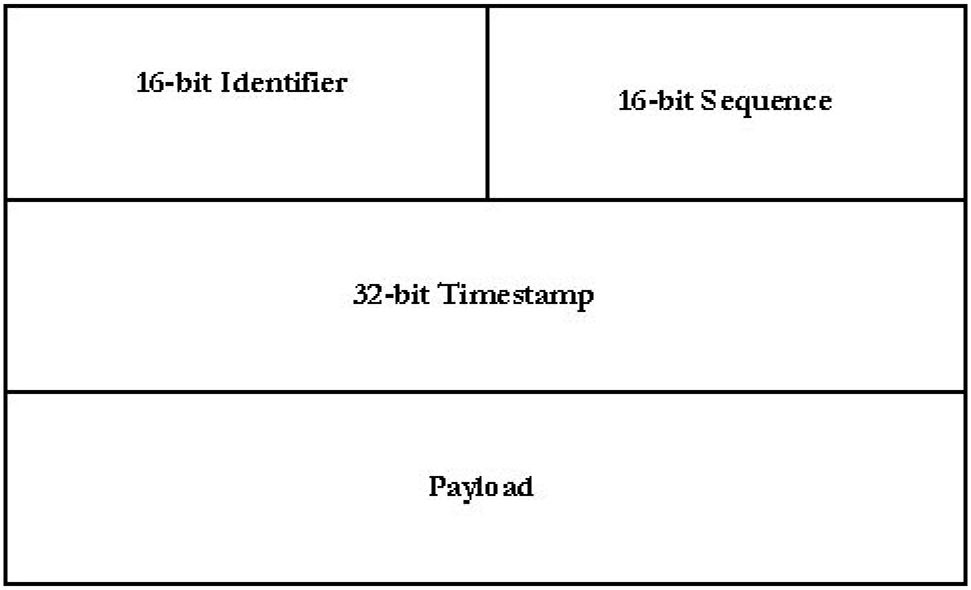


Figure 11-2 *Echo request header*

The first field is a 16-bit identifier, which is used to uniquely identify this request and is used to correlate echo replies received to your request and not some other process's request. Typically, the process identifier for the sending process is used. The next field is the sequence number, which identifies a given request packet from another. The 32-bit timestamp field is present ***only*** for ICMP requests (and not ICMPv6 requests). Following the request header is any payload. The following code sample illustrates initializing and sending an ICMP echo request for IPv4:

```
// Define the ICMP header
typedef struct icmp_hdr
{
    unsigned char   icmp_type;
    unsigned char   icmp_code;
    unsigned short  icmp_checksum;
    unsigned short  icmp_id;
    unsigned short  icmp_sequence;
    unsigned long   icmp_timestamp;
} ICMP_HDR, *PICMP_HDR, FAR *LPICMP_HDR;

ICMP_HDR *icmp=NULL;
SOCKET                   s;
SOCKADDR_STORAGE dest;
char                     buf[sizeof(ICMP_HDR) + 32];

icmp = (ICMP_HDR *)buf;
icmp->icmp_type = 8;                                    // echo
```

```
icmp->icmp_code = 0;
icmp->icmp_id   = GetCurrentProcessId();
icmp->icmp_checksum = 0;
icmp->icmp_sequence = 0;
icmp->icmp_timestamp = GetTickCount();
// Fill in the payload with a random character
memset(&buf[sizeof(ICMP_HDR)], '@', 32);
// Compute the checksum over the ICMP header and payload
//   The checksum() function computes the 16-bit one's
//   complement on the specified buffer. See the Ping
//   code sample on the companion CD for its implementation.
icmp->icmp_checksum = checksum(buf, sizeof(ICMP_HDR)+32);

s = socket(AF_INET, SOCK_RAW, IPPROTO_ICMP);

// Initialize the destination SOCKADDR_STORAGE
((SOCKADDR_IN *)&dest)->sin_family = AF_INET;
((SOCKADDR_IN *)&dest)->sin_port   = htons(0);
// port is ignored for ICMP
((SOCKADDR_IN *)&dest)->sin_addr.s_addr = inet_addr("1.2.3.4");

sendto(s, buf, sizeof(ICMP_HDR)+32, 0, (SOCKADDR *)&dest,
    sizeof(dest));
```

The only other difference between ICMP and ICMPv6 echo requests is computing the checksum contained in the ICMP header. For IPv4, the checksum is computed only over the ICMP header and payload. However, for IPv6 it is more complicated because IPv6 requires that the checksum include the IPv6 pseudo-header before the ICMPv6 header and payload. This means the Ping application must know the IPv6 source and destination address that will be in the IPv6 header to compute the checksum for any outgoing ICMPv6 requests. Because we are not building the IPv6 header by ourselves (as the case would be with the *IPV6_HDRINCL* option), we have no control over what goes into the IPv6 header. However, it is possible to query the transport for which local interface will be used to reach a given destination. This is performed with the *SIO_ROUTING_INTERFACE_QUERY* ioctl (see Chapter 7). Once this query is done, we have all the necessary information to compute the pseudo-header checksum.

When you send the ICMP echo request, the remote machine intercepts it and sends an echo reply message back to you. If for some reason the host is not reachable, the appropriate ICMP error message—such as destination host unreachable—will be returned by a router somewhere along the path to the intended recipient. If the physical network connection to the host is good but the remote host is either down or not responding to network events, you need to perform your own timeout to determine this. Because the timestamp in the echo request is echoed, when the reply is received the elapsed time is easily calculated. The PING.CPP example on the companion CD illustrates how to create a socket capable of sending and receiving ICMP packets, as well as how to use the *IP_OPTIONS* socket option to implement the record route option (supported for IPv4 only).

One noticeable feature of the Ping example is its use of the *IP_OPTIONS* socket option. We use the record route IPv4 option so that when the ICMP packet hits a router, its IPv4 address is added into the IPv4 option header at the location indicated by the offset field in the IPv4 option header. This offset is also incremented by four each time a router adds its address. The increment value is based on the fact that an IPv4 address is 4 bytes long. Once you receive the echo reply, decode the option header and print the IP addresses and host names of the routers visited. See Chapter 7 for more information about the other types of IP options available.

Traceroute

Another valuable IP networking tool is the Traceroute utility. It allows you to determine the IP addresses of the routers that are traversed to reach a certain host on the network. With Ping, using the record route option in the IPv4 option header also allows you to determine the IPv4 addresses of intermediary routers, but Ping is limited to only 9 hops—the maximum space allocated for addresses in the option header. Also, there is no equivalent option for IPv6. A hop occurs whenever an IP datagram must pass through a router to traverse multiple physical networks.

The idea behind Traceroute is to send a UDP packet to the destination and incrementally change the TTL value. Initially, the TTL value is 1, which means the UDP packet will reach the first router, where the TTL will expire. The expiration will cause the router to generate an ICMP time-exceeded packet. Then the initial TTL value increases by 1, so the UDP packet gets one router farther and an ICMP time-exceeded packet is sent from that router. Collecting each of the ICMP messages gives you a clear path of the IP addresses traversed to reach the endpoint. Once the TTL is incremented enough so that packets actually reach the endpoint, an ICMP port-unreachable message is most likely returned because no process on the recipient is waiting for this message.

Traceroute is a useful utility because it gives you a lot of information about the route to a particular host, which is often a concern when you use multicasting or when you experience routing problems. Fewer applications need to perform Traceroute programmatically than Ping, but certain tasks might require Traceroute-like capabilities.

Two methods can be used to implement the Traceroute program. First, you can use UDP packets and send datagrams, incrementally changing the TTL. Each time the TTL expires, an ICMP message will be returned to you. This method requires one socket of UDP to send the messages and another socket of ICMP to read them. The UDP socket is a normal UDP socket, as you saw in Chapter 1. The ICMP socket is a raw socket, which we already discussed how to create. The TTL of the UDP socket needs to be manipulated via the *IP_TTL* or *IPV6_UNICAST_HOPS* socket option. Alternatively, you can create a UDP socket and use the *IP_HDRINCL* option (discussed later in this chapter) to set the TTL manually within the IP header, but this is quite a lot of work.

The other method is simply to send ICMP packets to the destination, also incrementally changing the TTL. This also results in ICMP error messages being returned when the TTL expires. This method resembles the Ping example because it requires only one socket (of ICMP). Under the sample code folder on the companion CD, you will find a Traceroute example using ICMP packets named TRACERT.CPP. The traceroute is similar in structure and code to the Ping sample. The only difference is that the TTL value is incremented with each send.

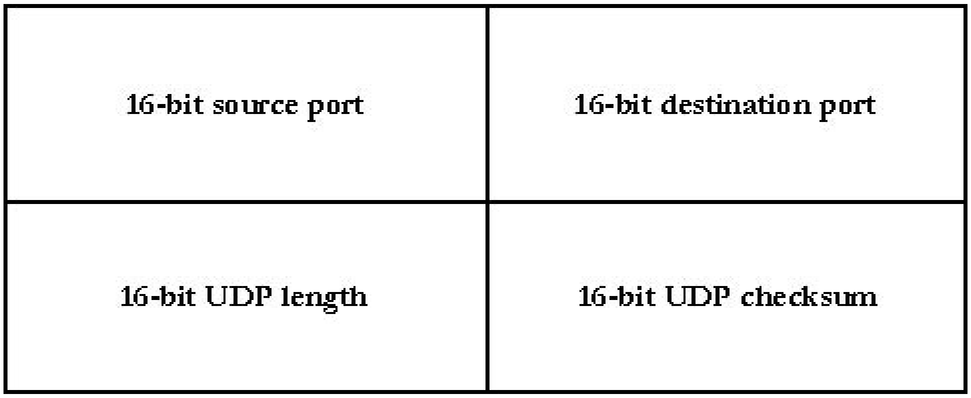
Using IP Header Include Option

The one limitation of raw sockets is that you can work only with certain protocols that are already defined, such as ICMP and IGMP. You cannot create a raw socket with *IPPROTO_UDP* and manipulate the UDP header; likewise with TCP. To manipulate the IP header as well as either the TCP or UDP header (or any other protocol encapsulated in IP), you must use the *IP_HDRINCL* socket option with a raw socket. For IPv6, the option is *IPV6_HDRINCL*. This option allows you to build your own IP header as well as other protocols' headers.

In addition to manipulating well-known protocols such as UDP, using raw sockets with the header include option allows you to implement your own protocol scheme that is encapsulated in IP. This is done by creating a raw socket and using the *IPPROTO_RAW* value as the protocol. This allows you to set the protocol field in the IP header manually and build your own custom protocol header. However, in this section we will take a look at how to build your own UDP packets so that you can gain a good understanding of the steps involved. Once you understand how to manipulate the UDP header, creating your own protocol header or manipulating other protocols encapsulated in IP is fairly trivial.

Before getting into the details of using the header include option, you need to know one important difference between using this option with IPv4 and IPv6. For IPv4, the stack still verifies some fields within the supplied IPv4 header. For example, the IPv4 identification field is set by the stack and the stack will fragment the packet if necessary. That is, if you create a raw IPv4 packet and set *IP_HDRINCL* and send a packet larger than the MTU size, the stack will fragment the data into multiple packets for you. For IPv6, if the *IPV6_HDRINCL* option is set, it is your responsibility to compute all the headers and fields necessary. If you submit a send larger than the MTU size, your application must create the IPv6 fragment headers and compute the offsets correctly; otherwise, the IPv6 stack will drop the packet without sending it.

When you use the header include option, you are required to fill in the IP header yourself for every send call, as well as the headers of any other protocols wrapped within. Both IPv4 and IPv6 headers are shown in Chapter 7 in Figures 7-3 and 7-4. The UDP header is quite a bit simpler than the IP header. It is only 8 bytes long and contains only four fields, as shown in Figure 11-3. The first two fields are the source and destination port numbers. They are 16 bits each. The third field is the UDP length, which is the length, in bytes, of the UDP header and data. The fourth field is the checksum, which we will discuss shortly. The last part of the UDP packet is the data.



16-bit source port	16-bit destination port
16-bit UDP length	16-bit UDP checksum

Figure 11-3 *UDP header format*

Because UDP is an unreliable protocol, calculating the checksum is optional. Unlike the IPv4 checksum, which covers only the IPv4 header, the UDP checksum covers the data and also includes part of the IPv4 header. The additional fields required to calculate the UDP checksum are known as a pseudo-header. The IPv4 UDP pseudo-header is composed of the following items:

- 32-bit source IP address (IP header)
- 32-bit destination IP address (IP header)
- 8-bit field zeroed out
- 8-bit protocol
- 16-bit UDP length

Added to these items are the UDP header and data. The method of calculating the checksum is the 16-bit one's complement sum. Because the data can be an odd number of bytes, it might be necessary to pad a zero byte to the end of the data to calculate the checksum. This pad field is not transmitted as part of the data. Figure 11-4 illustrates all of the fields required for the checksum calculation. The first three 32-bit words make up the UDP pseudo-header. The UDP header and its data follows that. Notice that because the checksum is calculated on 16-bit values, the data might need to be padded with a zero byte.

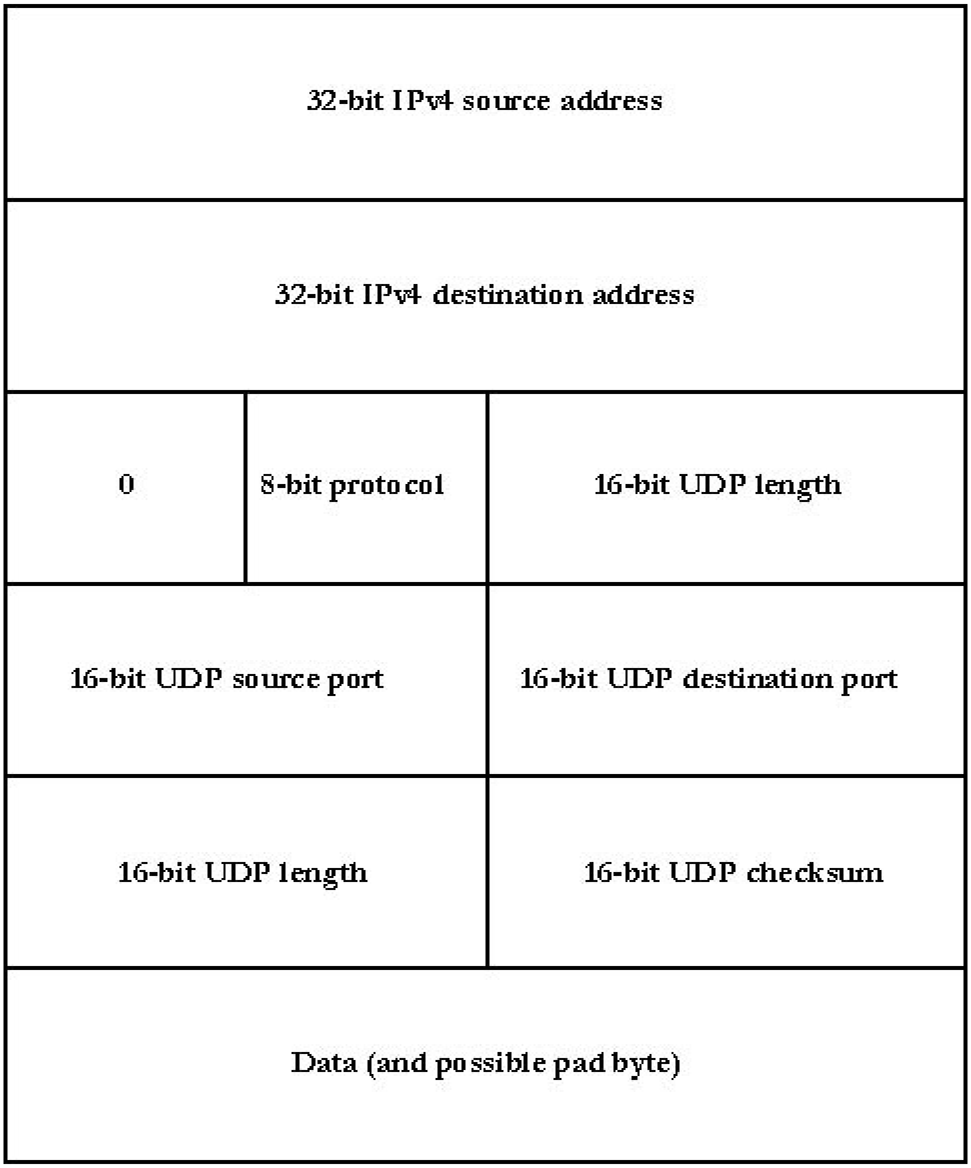

32-bit IPv4 source address
32-bit IPv4 destination address
0
8-bit protocol
16-bit UDP length
16-bit UDP source port
16-bit UDP destination port
16-bit UDP length
16-bit UDP checksum
Data (and possible pad byte)

Figure 11-4 *IPv4 pseudo-header with UDP packet and data*

For IPv6, you have already seen how to calculate the IPv6 pseudo-header as is required to calculate the checksum for ICMPv6 packets. The calculation is the same for UDP with the IPv6 pseudo-header coming first and is followed by the UDP header and payload (zero padded to the next 16-bit boundary if necessary). The IPv6 pseudo-header is shown in Figure 11-5.

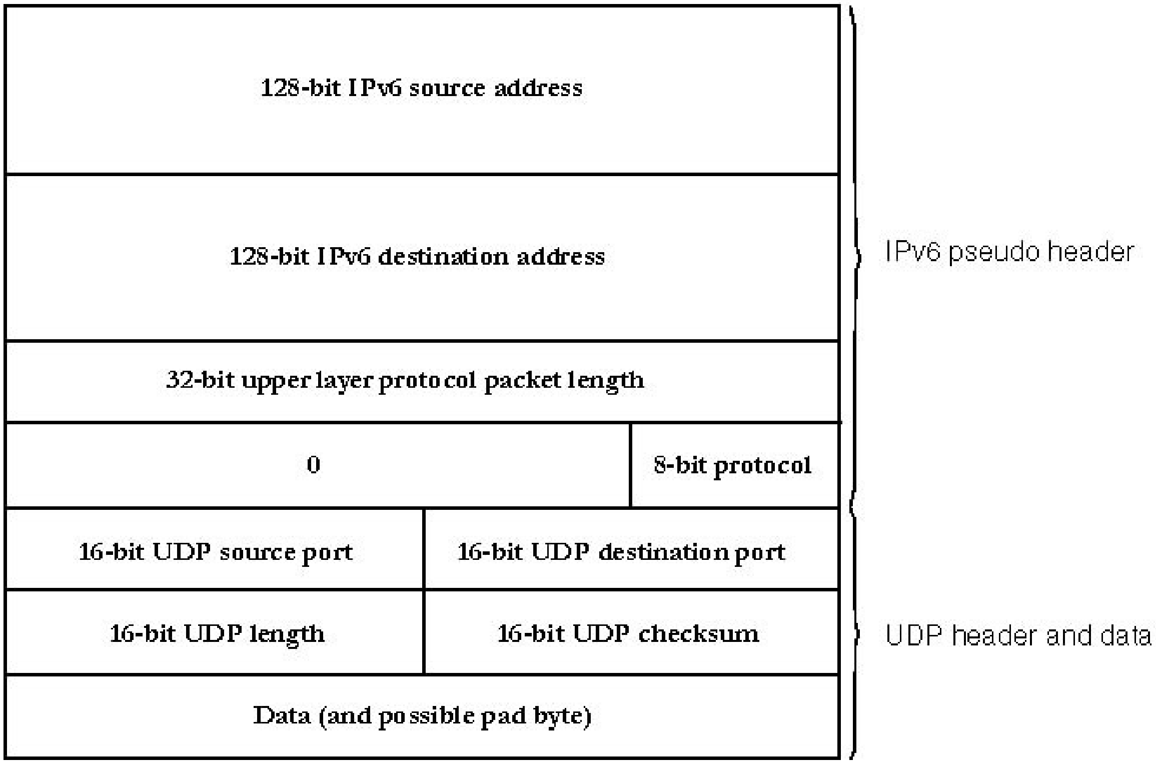

128-bit IPv6 source address
128-bit IPv6 destination address
32-bit upper layer protocol packet length
0
8-bit protocol
16-bit UDP source port
16-bit UDP destination port
16-bit UDP length
16-bit UDP checksum
Data (and possible pad byte)
IPv6 pseudo header
UDP header and data

Figure 11-5 *IPv6 pseudo-header with UDP packet and data*

The directory IPHDRINC on the companion CD contains a fully functional sample that creates raw UDP packets over IPv4 and IPv6.

The following code snippet shows how to build an IPv4 and UDP header:

```
// IPv4 header
typedef struct ip_hdr
{
    unsigned char  ip_verlen;         // 4-bit IPv4 version
                                      // 4-bit header length (in
                                      // 32-bit words)
    unsigned char  ip_tos;            // IP type of service
    unsigned short ip_totallength;    // Total length
    unsigned short ip_id;             // Unique identifier
    unsigned short ip_offset;         // Fragment offset field
    unsigned char  ip_ttl;            // Time to live
    unsigned char  ip_protocol;       // Protocol(TCP,UDP etc)
    unsigned short ip_checksum;       // IP checksum
    unsigned int   ip_srcaddr;        // Source address
    unsigned int   ip_destaddr;       // Source address
} IPV4_HDR, *PIPV4_HDR, FAR * LPIPV4_HDR;

// Define the UDP header
typedef struct udp_hdr
{
    unsigned short src_portno;        // Source port no.
    unsigned short dst_portno;        // Dest. port no.
    unsigned short udp_length;        // Udp packet length
    unsigned short udp_checksum;      // Udp checksum (optional)
} UDP_HDR, *PUDP_HDR;
```

```
SOCKET    s;
char      buf[MAX_BUFFER], // large enough buffer
         *data=NULL;
IPV4_HDR *v4hdr=NULL;
UDP_HDR  *udphdr=NULL;
USHORT   sourceport=5000,
         Destport=5001;
int      payload=512,                                   // size of UDP data
         optval;
SOCKADDR_STORAGE dest;

// Initialize the IPv4 header
v4hdr = (IPV4_HDR *)buf;
v4hdr->ip_verlen = (4 << 4) | (sizeof(IPV4_HDR) / sizeof(ULONG));
v4hdr->ip_tos    = 0;
v4hdr->ip_totallength = htons(sizeof(IPV4_HDR) + sizeof(UDP_HDR) +
    payload);
v4hdr->ip_id     = 0;
v4hdr->ip_offset = 0;
v4hdr->ip_ttl    = 8;                                   // Time-to-live is eight
v4hdr->ip_protocol = IPPROTO_UDP;
v4hdr->ip_checksum = 0;
v4hdr->ip_srcaddr  = inet_addr("1.2.3.4");
v4hdr->ip_destaddr = inet_addr("157.32.159.101");
// Calculate checksum for IPv4 header
//   The checksum() function computes the 16-bit one's
//   complement on the specified buffer. See the IPHDRINC
//   code sample on the companion CD for its implementation.
v4hdr->ip_checksum = checksum(v4hdr, sizeof(IPV4_HDR));

// Initialize the UDP header
udphdr = (UDP_HDR *)&buf[sizeof(IPV4_HDR)];
udphdr->src_portno = htons(sourceport);
udphdr->dst_portno = htons(destport);
udphdr->udp_length = htons(sizeof(UDP_HDR) + payload);
udphdr->udp_checksum = 0;

// Initialize the UDP payload to something
data = &buf[sizeof(IPV4_HDR) + sizeof(UDP_HDR)];
memset(data, '^', payload);

// Calculate the IPv4 and UDP pseudo-header checksum - this routine
// extracts all the necessary fields from the headers and calculates
// the checksum over it. See the iphdrinc sample for the implementation
//    of Ipv4PseudoHeaderChecksum().
udphdr->udp_checksum = Ipv4PseudoHeaderChecksum(v4hdr, udphdr, data,
         sizeof(IPV4_HDR) + sizeof(UDP_HDR) + payload);

// Create the raw UDP socket
s = socket(AF_INET, SOCK_RAW, IPPROTO_UDP);

// Set the header include option
optval = 1;
setsockopt(s, IPPROTO_IP, IP_HDRINCL, (char *)&optval, sizeof(optval));

// Send the data
((SOCKADDR_IN *)&dest)->sin_family = AF_INET;
```

```
((SOCKADDR_IN *)&dest)->sin_port   = htons(destport);
((SOCKADDR_IN *)&dest)->sin_addr.s_addr = inet_addr("157.32.159.101");

sendto(s, buf, sizeof(IPV4_HDR) + sizeof(UDP_HDR) + payload, 0,
         (SOCKADDR *)&dest, sizeof(dest));
```

This code is straightforward and easy to follow. The IPv4 header is initialized with valid entries. In this case, a bogus source IPv4 address is used (1.2.3.4) but a valid destination address is supplied. Also, we set the TTL value to 8. Lastly, the checksum is calculated for the IPv4 header only. After the IPv4 header is the UDP header—as indicated by the *ip_protocol* field of the IPv4 header being set to *IPPROTO_UDP*. For that header, the source and destination ports are set in addition to the length of the UDP header and its payload. The last piece is to compute the pseudo-header checksum, which isn't shown but is an easy computation. The necessary fields are extracted out of the various headers after which the checksum can be computed. The code sample on the CD has a routine to compute the pseudo-header checksum for both IPv4 and IPv6.

As we mentioned previously, using the header include option for IPv4 is easy because the stack will perform any fragmentation necessary. However, for IPv6 the stack will not, which means if your application needs to send raw data with a payload that exceeds the MTU, it will have to fragment the packets manually before sending them. This is accomplished by including the IPv6 fragmentation header after the IPv6 header but before the remaining payload. To do this, the IPv6 header's next header value will indicate the IPv6 fragmentation header (whose value is 44). The next header value of the IPv6 fragmentation header will then indicate *IPPROTO_UDP*. Also note that the UDP header occurs only once. The first fragment will contain the IPv6 header, IPv6 fragmentation header, UDP header, and as much of the payload that will fit into the MTU. The subsequent fragments will contain only the IPv6 header, the IPv6 fragmentation header, and the remaining payload. Figure 11-6 illustrates this example. In this case, the MTU is 1500 bytes but a 2000 byte payload is being sent.

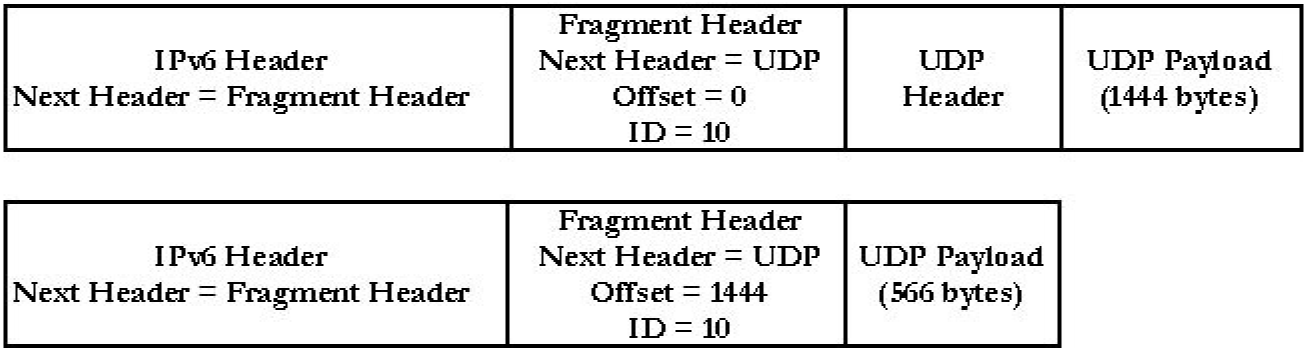


Figure 11-6 *IPv6 UDP packet with fragmentation*

Again, the RAWUDP.CPP sample on the companion CD illustrates sending a fabricated UDP packet for both IPv4 and IPv6. There are two routines of interest: *PacketizeIpv4* and *PacketizeIpv6*. The v4 routine doesn't do anything of interest because we know the stack will fragment the data if required. However, the v6 routine will build the appropriate IPv6 header and fragmentation header for each fragment necessary.

Conclusion

Raw sockets are a powerful mechanism to manipulate the underlying protocol. This chapter illustrated how you can use raw sockets to create ICMP and ICMPv6 applications through Winsock, but raw sockets can be used in a multitude of other applications—too many to discuss in a single chapter. To take full advantage of the capabilities of raw sockets and the header include option (*IP_HDRINCL* and *IPV6_HDRINCL*), you must thoroughly understand the IP protocol as well as any protocols encapsulated in it.

Chapter 12

The Winsock Service Provider Interface

The Winsock 2 Service Provider Interface (SPI) is the complement to the Winsock API we have been discussing. As the name implies, the SPI is a service to applications and is not an application. It is written and exposes itself to applications that can load the service either knowingly or unknowingly. The SPI is a part of the Winsock 2 specification and therefore requires the Winsock 2 update if running in Windows 95. Figure 12-1 illustrates the relationship between Winsock applications and the SPI.

There are two parts to the SPI: transport service providers and name space providers. Each part provides distinctly different functionalities. There are two types of transport service providers: layered and base. A layered service provider installs itself into the Winsock catalog above base providers and possibly between other layered providers and intercepts Winsock API calls from applications. A base provider exposes a Winsock interface that directly implements a protocol such as the Microsoft TCP/IP provider. This chapter discusses only layered service providers. When an application creates a socket that matches the characteristics of the layered provider, that layered service provider is called and can intercept Winsock calls.

A name space provider is similar to a transport service provider except that it intercepts the name resolution Winsock API calls, such as *gethostbyname* and *WSALookupServiceBegin*. A name space provider installs itself within the name space catalog and is invoked when applications perform name resolution searches that match that name space provider.

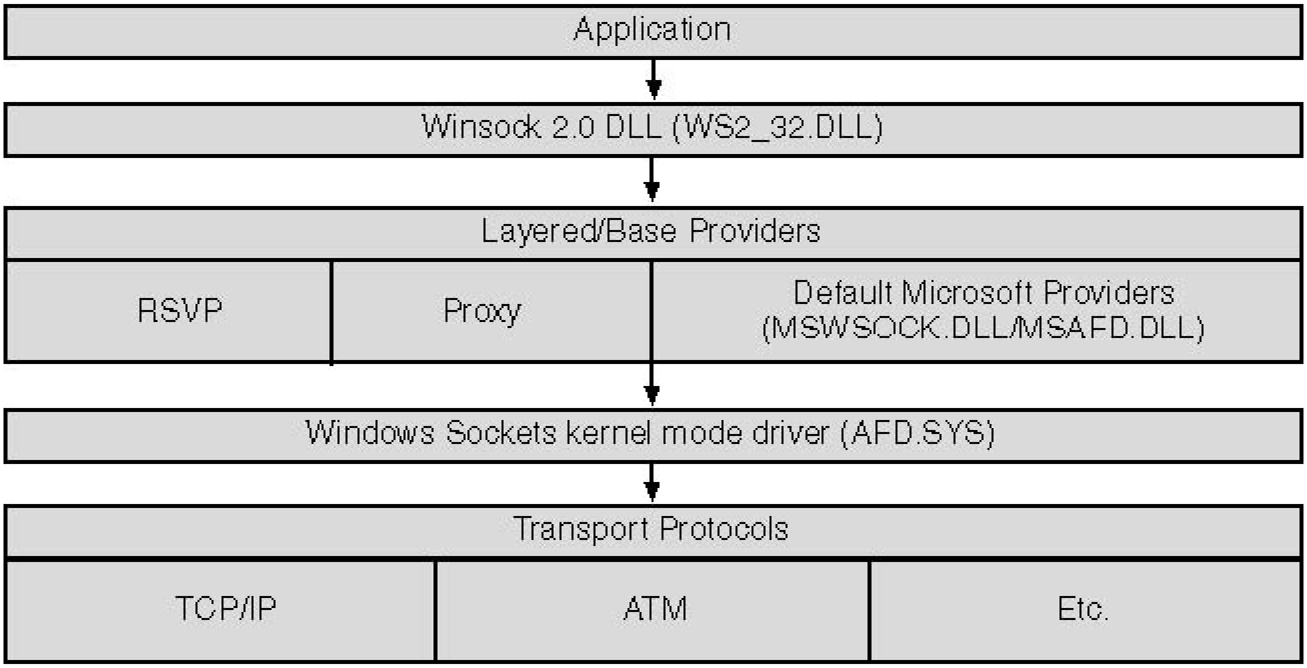

Application
Winsock 2.0 DLL (WS2_32.DLL)
Layered/Base Providers
RSVP
Proxy
Default Microsoft Providers (MSWSOCK.DLL/MSAFD.DLL)
Windows Sockets kernel mode driver (AFD.SYS)
Transport Protocols
TCP/IP
ATM
Etc.

Figure 12-1 *SPI architecture*

Before getting into the specifics, we'll cover some of the basics that pertain to both layered and name space providers. The Winsock Service Provider APIs are contained in the header file WS2SPI.H and SPI applications link with WS2_32.LIB. In addition, there are four types of APIs defined in the SPI.

Table 12-1 lists the prefixes for each type as well as whether they belong to service providers or name space providers.

Table 12-1 *SPI Function Prefixes*

API Prefix	Description
WSC	Installing, removing, or modifying layered and name space providers
WSP	Layered service provider APIs
WPU	Support functions that layered providers use
NSP	Name space providers APIs

In this chapter we'll examine the layered provider interface followed by the name space provider interface.

Layered Service Provider

As we mentioned, a layered service provider (LSP) installs itself into the Winsock catalog so that an application that creates a socket will call into it without necessarily having any awareness of the LSP. This is useful for developing system components that modify or monitor any portion of the Winsock API. For example, a secure socket provider that implements SSL can be implemented as a layered service provider. In this example, the LSP would negotiate the SSL connection when the application issues a connect as well as encrypting data sent via any Winsock send command while decrypting data returned from the receive commands. Other possibilities include Winsock proxy clients and content filtering.

An LSP accomplishes this by installing an entirely new Winsock provider that mimics or extends an existing provider. For example, if you were developing an LSP that filters HTTP requests, you would need to layer your provider over the Microsoft TCP provider because the HTTP protocol runs over TCP. You would want this new provider to be virtually indistinguishable (at least from an application's perspective) from the base Microsoft provider because you want any application that uses TCP to go through your provider first. Of course, it is possible to create an LSP that implements an entirely different protocol with different semantics on top of an existing Winsock provider.

In Chapter 2, you saw how Winsock selects the appropriate provider to load when a socket is created. When an LSP is installed, it is placed in the catalog in a certain order. When an application creates a socket, the catalog is enumerated in order until the best match is found, at which point the system loads that provider. This allows a layered provider to be loaded instead of the default Microsoft provider.

When an application that created a socket from the layered provider makes a Winsock call, the system routes the call into the LSP. At that point, the LSP can perform its necessary tasks. It can also pass the request to the provider below itself if further action is required. For example, in our HTTP content filtering examples, we may want to intercept HTTP requests and modify them before actually making the request. This would require the LSP to perform some action for any of the Winsock APIs that send data. When the application calls any Winsock send function, the call is routed to the LSP, which examines the send buffer and makes the appropriate modifications to it. Of course, the LSP doesn't actually know how to send TCP data; it relies on the underlying TCP provider, which has a kernel mode driver that implements the protocol. The LSP must know where it resides in the protocol chain to pass the modified send request to the provider beneath it. In many cases, this will be the base provider but an LSP can be installed over other LSPs. Eventually, the request will make it to a base provider, which will perform the appropriate action. In the next section you'll see exactly how these protocol chains are implemented because when an LSP is installed, these chains must be built. Figure 12-2 shows the relationship between applications, layered service providers, and base providers.

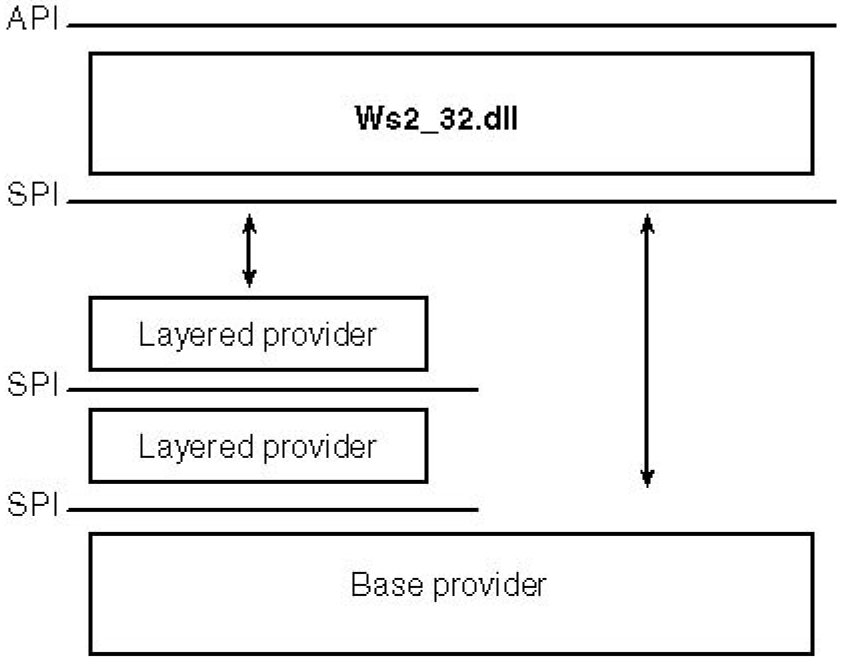


Figure 12-2 *Layered provider architecture*

Winsock LSPs are implemented as a standard Windows dynamic-link library into which you must export a single function entry named *WSPStartup*. When the system invokes the layered provider's *WSPStartup*, it must expose 30 additional SPI functions that make up the LSP via a function dispatch table passed as a parameter. Table 12-2 lists those SPI functions that must be implemented within the DLL.

Table 12-2 *Transport Provider Support Functions*

API Function	Maps to SPI Function
WSAAccept (*accept* also indirectly maps to *WSPAccept*)	*WSPAccept*
WSAAddressToString	*WSPAddressToString*
WSAAsyncSelect	*WSPAsyncSelect*
Bind	*WSPBind*
WSACancelBlockingCall	*WSPCancelBlockingCall*
WSACleanup	*WSPCleanup*
closesocket	*WSPCloseSocket*
WSAConnect (*connect* also indirectly maps to *WSPConnect*)	*WSPConnect*
WSADuplicateSocket	*WSPDuplicateSocket*
WSAEnumNetworkEvents	*WSPEnumNetworkEvents*
WSAEventSelect	*WSPEventSelect*
WSAGetOverlappedResult	*WSPGetOverlappedResult*
getpeername	*WSPGetPeerName*
getsockname	*WSPGetSockName*
getsockopt	*WSPGetSockOpt*
WSAGetQOSByName	*WSPGetQOSByName*
WSAIoctl	*WSPIoctl*
WSAJoinLeaf	*WSPJoinLeaf*

Listen	*WSPListen*
WSARecv (*recv* also indirectly maps to *WSPRecv*)	*WSPRecv*
WSARecvDisconnect	*WSPRecvDisconnect*
WSARecvFrom (*recvfrom* also indirectly maps to *WSPRecvFrom*)	*WSPRecvFrom*
Select	*WSPSelect*
WSASend (*send* also indirectly maps to *WSPSend*)	*WSPSend*
WSASendDisconnect	*WSPSendDisconnect*
WSASendTo (*sendto* also indirectly maps to *WSPSendTo*)	*WSPSendTo*
setsockopt	*WSPSetSockOpt*
shutdown	*WSPShutdown*
WSASocket (*socket* also indirectly maps to *WSPSocket*)	*WSPSocket*
WSAStringToAddress	*WSPStringToAddress*

In most cases, when an application calls a Winsock function, WS2_32.DLL calls a corresponding Winsock SPI function to carry out the request using a specific service provider. For example, *select* maps to *WSPSelect*, *WSAConnect* maps to *WSPConnect*, and *WSAAccept* maps to *WSPAccept*. However, not all Winsock functions have a corresponding SPI function. The following list details these exceptions.

- Support functions such as *htonl*, *htons*, *ntohl*, and *ntohs* are implemented within WS2_32.DLL and aren't passed down to a service provider. The same holds true for the WSA versions of these functions.
- IP conversion functions such as *inet_addr* and *inet_ntoa* are implemented only within WS2_32.DLL.
- All of the IP specific name conversion and resolution functions (i.e., the *WSAGetXbyY* functions) as well as *WSACancelAsyncRequest* and *gethostname* are implemented within WS2_32.DLL.
- Winsock catalog functions and blocking hook-related functions are implemented within WS2_32.DLL. Thus, *WSAEnumProtocols*, *WSA-IsBlocking*, *WSASetBlockingHook*, and *WSAUnhookBlockingHook* do not have SPI equivalent functions.
- Winsock error codes are managed within WS2_32.DLL and as such *WSAGetLastError* and *WSASetLastError* are not mapped to service providers.
- The event object manipulation and wait functions—including *WSACreateEvent*, *WSACloseEvent*, *WSASetEvent*, *WSAResetEvent*, and *WSAWaitForMultipleEvents*—are mapped directly to native Windows operating system calls and aren't present in the service provider.

Also, a sample LSP is included on the companion CD in the directory Lsp. This LSP is a pass-through LSP. It doesn't modify any of the Winsock API calls, it simply passes the call down to the lower layer. Throughout our discussion of layered providers, we'll refer to the sample code to illustrate various points.

Before getting into the details of installing and implementing an LSP, we should discuss error handling. Winsock applications often use *WSAGetLastError* and sometimes *WSASetLastError*.

However, as we have pointed out, there is no SPI equivalent to these functions. Instead, each of the SPI functions an LSP must implement (listed in Table 12-2) are exact mirrors of their API equivalents in terms of parameters except for an additional parameter, *lpErrno*. Those APIs that can be called in an overlapped manner have one additional parameter in addition to *lpErrno* the thread ID for the calling thread (which is discussed in the “Handling I/O” section). This is a pointer to an integer that should be set to the correct error code in case the LSP function fails. To indicate a failure, the LSP function should return *SOCKET_ERROR* and set *lpErrno*. For success, *NO_ERROR* is returned and the *lpErrno* value is ignored. The only exception is *WSPStartup*, which either returns *NO_ERROR* or the actual error code that caused startup to fail.

Installing an LSP

Before we talk about implementing an LSP, the first step is installing the layered provider into the Winsock catalog, which can become very complicated in itself. In Chapter 2, you saw how an application can enumerate the Winsock catalog as well as provide a code sample illustrating that. Installing an LSP consists of installing a *WSAPROTOCOL_INFOW* structure defining the characteristics of the layered provider as well as how the LSP fits into the “chain.” As the name “layered service provider” implies, providers are layered on top of one another to form a protocol chain that is defined as

```
typedef struct _WSAPROTOCOLCHAIN {
    int ChainLen;
    DWORD ChainEntries[MAX_PROTOCOL_CHAIN];
} WSAPROTOCOLCHAIN, FAR * LPWSAPROTOCOLCHAIN;
```

The *ChainLen* field is important because it indicates the type of provider the entry is. Table 12-3 lists the possible values. When *ChainLen* is zero or 1, the data contained in the *ChainEntries* array is meaningless. The value of one indicates a base provider, such as the Microsoft TCP and UDP providers. Typically, a base provider has a kernel mode protocol driver associated with it. For example, the Microsoft TCP and UPD providers require the TCP/IP driver TCPIP.SYS to ultimately function. It is also possible to develop your own base providers, but that is beyond the scope of this book. For more information about base providers, consult the Windows Driver Development Kit (DDK).

Table 12-3 *Chain Length and Type of Provider*

ChainLen Value	Description
0	Layered provider entry
1	Base provider
2 or more	Layered chain entry

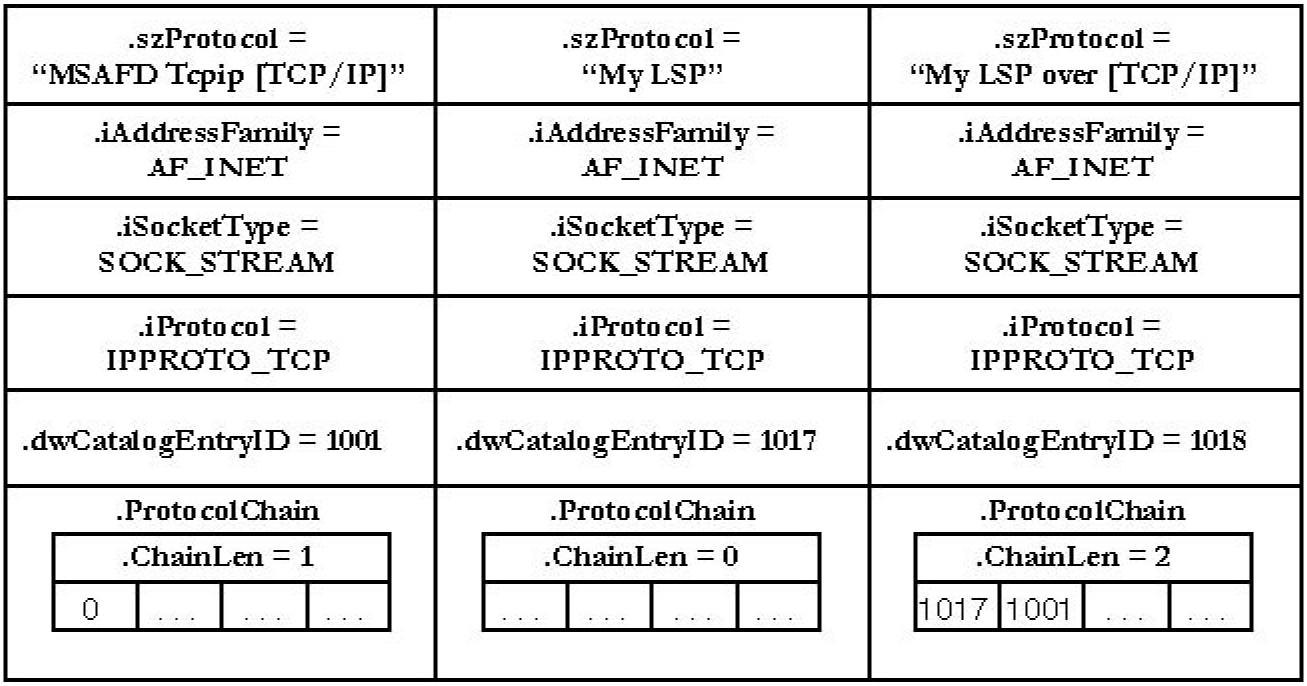
Layered providers use a chain length of zero or greater than 1. Entries whose chain length is zero are special. When a layered provider is installed, the protocol chain must be constructed that describes where the layered provider resides. This is done by filling in the *ChainEntries* array with the catalog IDs for each protocol in the chain. The catalog ID is the *dwCatalogEntryId* contained in the

WSAPROTOCOL_INFOW structure.

Let's look at a quick example before going any further. Say we're developing an LSP that will be layered over the base Microsoft TCP provider. This will require us to install a single provider whose *ChainLen* will be two. The *ChainEntries* array will contain two entries: first is the layered provider catalog ID and second is the Microsoft TCP provider catalog ID. The problem is the value to use for the layered provider's catalog ID. When constructing the *WSAPROTOCOL_INFOW* structure that describes the layered chain for our LSP, the *dwCatalogEntryId* is not initialized and we cannot simply make one up. A catalog ID is assigned ***only*** when a provider is installed via *WSCInstallProvider*. To solve this problem, a dummy provider entry is installed first whose *ChainLen* is zero. Once this dummy provider (also known as the layered provider) is installed, the system assigns the catalog ID, which we can then use to install the actual layered chain entry.

The dummy layered provider's *WSAPROTOCOL_INFOW* structure contains meaningless data (except for the path to the provider's DLL, which will be discussed later). Furthermore, an application that calls *WSAEnumProtocols* will not see any entry with a chain length of zero; only *WSCEnumProtocols* will return these entries (along with all other entries). When writing the install (and remove) code for service provider, you want to use *WSCEnumProtocols* or you'll never see the layered provider dummy entries, only base and layered chain entries.

Getting back to our example LSP, first the dummy LSP entry is installed, after which the catalog is enumerated so we can find the provider ID of the dummy entry. Then we build the *WSAPROTOCOL_INFOW* structure, which describes our layered chain. In this structure the *ChainLen* is 2; *ChainEntries* contains two values. The first value is the catalog entry ID of the dummy entry just installed and the second array index contains the catalog entry ID of the base TCP provider. Figure 12-3 illustrates three *WSAPROTOCOL_INFOW* structures. The structure on the left is the default Microsoft TCP provider. The structure in the middle is the dummy LSP entry, and the structure on the right is the layered chain entry for the LSP provider. Notice that the protocol chain for the LSP provider contains two entries. Also notice that the figure illustrates only the first four protocol chain entries while the *WSAPROTOCOL_INFOW* structure actually contains *MAX_PROTOCOL_CHAIN* entries (which is seven).



.szProtocol = “MSAFD Tcpip [TCP/IP]”	.szProtocol = “My LSP”	.szProtocol = “My LSP over [TCP/IP]”
.iAddressFamily = AF_INET	.iAddressFamily = AF_INET	.iAddressFamily = AF_INET
.iSocketType = SOCK_STREAM	.iSocketType = SOCK_STREAM	.iSocketType = SOCK_STREAM
.iProtocol = IPPROTO_TCP	.iProtocol = IPPROTO_TCP	.iProtocol = IPPROTO_TCP
.dwCatalogEntryID = 1001	.dwCatalogEntryID = 1017	.dwCatalogEntryID = 1018
.ProtocolChain .ChainLen = 1 0 | ... | ... | ...	.ProtocolChain .ChainLen = 0 ... | ... | ... | ...	.ProtocolChain .ChainLen = 2 1017 | 1001 | ... | ...

***Figure 12-3** Example LSP layered over the base Microsoft TCP Provider*

Installing a Provider Entry

Now that we've covered the basics, let's look at the API used to install a Winsock provider, *WSCInstallProvider*. The API is defined as

```
        int WSPAPI
        WSCInstallProvider(
                IN LPGUID lpProviderId,
                IN const WCHAR FAR *lpszProviderDllPath,
                IN const LPWSAPROTOCOL_INFOW lpProtocolInfoList,
                IN DWORD dwNumberOfEntries,
                OUT LPINT lpErrno
);
```

The first thing to notice is this API comes in only a UNICODE version. The parameter list is almost self-explanatory. Each provider installed requires a GUID to uniquely identify that provider entry. A GUID can be generated by the command line utility UUIDGEN.EXE or programmatically by *UuidCreate*. One GUID is required for the dummy layered provider entry and one (or more) is required for the layered chain entry or entries. The *lpszProviderDllPath* parameter is a UNICODE string that contains the path to the DLL that implements the layered provider. The DLL path can contain environment variables such as %SYSTEMROOT%. This provider path should be correct for both the layered provider entries and layered chain entries. Lastly, note that only members of the Administrators group can install (and remove) Winsock catalog entries.

The *lpProtocolInfoList* is an array of *WSAPROTOCOL_INFOW* structures. Each entry in the array is a separate provider entry to be installed. *dwNumberOfEntries* indicates the number of entries in the array. If the provider being installed is layered over multiple providers, they may be installed all at once or one at a time—which is an issue to consider as we will find out later. Of course, the dummy layered provider entry must be installed first by itself to obtain a catalog ID entry used by the layered protocol entries. The last parameter returns the error code in case of a failure, at which point the API returns *SOCKET_ERROR*.

As we have already mentioned, the layered provider entry is meaningless and is installed only to obtain a catalog ID. For layered protocol entries, the *WSAPROTOCOL_INFOW* structure is typically copied from the provider that is to be layered over with two exceptions. First, the *szProtocol* field is modified to contain the name of the new provider. Second, the flag *XP1_IFS_HANDLES* is removed from the *dwServiceFlags1* field if present. When this flag is set, it indicates that the socket handles that this provider returns are true operating system handles and may be passed interchangeably to APIs that don't specifically take SOCKET handles (such as *ReadFile* and *WriteFile*) without taking a performance penalty. For a layered provider to return IFS handles there must be an associated kernel mode component that creates these handles such as what TCPIP.SYS does for the Microsoft TCP and UDP providers. We'll discuss socket handles in more detail later in this chapter.

Of course, if the LSP being developed is a completely new protocol, the install application must set the proper flags and fields within the *WSAPROTOCOL_INFOW* structure to accurately describe the behavior that the provider exposes. See Chapter 2 for a full description of the protocol structure.

Finally, when installing new provider entries, the entries by default appear at the end of the Winsock catalog when enumerated. If your LSP mimics a TCP/IP provider, it will never be called by default because the system will always match socket creation calls to the MSAFD TCP/IP provider that appears before your LSP's entry in the enumeration (see Chapter 2 for more information on how the system finds the appropriate Winsock provider to load). As a result, it may be necessary to reorder the catalog so that the newly install LSP entries appear first. This is done with the API *WSCWriteProviderOrder* defined as

```
int WSPAPI WSCWriteProviderOrder(
                IN LPDWORD lpdwCatalogEntryId,
                IN DWORD dwNumberOfEntries
);
```

The first parameter is an array of *DWORD*, which contains the catalog entry IDs for every provider in the catalog in the order in which they should be written. For example, if there are 20 entries in the Winsock catalog (as returned from *WSCEnumProtocols*), the array should contain 20 entries, each with the catalog ID of an existing provider. After the API is called, the catalog will be reordered in the sequence specified. The array should not contain any duplicates. Note that this API is defined in the header file SPORDER.H and in the library SPORDER.LIB. On recent Platform SDK releases, the definition of this function has been moved into WS2_32.LIB, and SPORDER.LIB simply contains a forward to that definition.

After successfully installing an LSP it is a good idea to reboot the machine. Many of the system services, such as Local Security Authentication Server System (LSASS), create only the majority of their sockets upon bootup and create additional sockets as time goes on. The problem is that after an LSP is installed over the providers they are using, these services now have a mixed set of sockets from multiple providers. This can be problematic if these applications use the select API.

To summarize, installing a provider requires the installation of the “dummy” layered provider entry with a chain length of zero. This is necessary to obtain a catalog ID that the layered chain entries can later reference. After the layered provider is installed, each layered chain is installed that references the catalog ID of the layered dummy provider as its first chain entry. The subsequent chain entries are the catalog IDs of the providers layered under this one. Then, in most cases, the Winsock catalog needs to be reordered so that most applications will call into the LSP rather than into the base providers.

Note that special consideration must be taken when manipulating the Winsock catalog on 64-bit Windows. In order for 32-bit applications to run on 64-bit Windows, two separate Winsock catalogs are maintained—one for 32-bit applications and one for 64-bit native applications. To manipulate the catalog, several new *WSC* functions have been introduced. There is a new API for each *WSC* function that has the string "32" appended to the function name. Note that the parameters remain the same. For example, the function *WSCInstallProvider* has a corresponding function *WSCInstallProvider32*. The "normal" function (i.e., not ending in 32) operates on the Winsock catalog for the platform the install application is compiled for. That is, if our LSP install program is compiled for 64-bit Windows, the *WSCInstallProvider* function installs the LSP into the 64-bit catalog. Likewise, if it was compiled for 32-bit Windows, the LSP would be installed into the 32-bit catalog. The new functions ending in 32 can be used by a native 64-bit application to manipulate the 32-bit catalog.

The only problem is what to do if an LSP needs to be installed into both the 32-bit and 64-bit catalog. When we build the protocol chain, the same catalog ID for the dummy provider is present in the chain. To solve this problem, there is another version of the install function, *WSCInstallProvider64_32*. This function installs the provider into both catalogs so that the same catalog ID is assigned to the 32-bit and 64-bit entries. Also note that when installing into both catalogs, two versions of the LSP's DLL need to be present. The native 64-bit compiled version goes in %SYSTEMROOT%\system32, while the 32-bit version is placed in %SYSTEMROOT%\syswow64. Note that it still requires two separate calls to remove the LSP once installed into both catalogs—i.e., there is no equivalent uninstall routine that operates on both catalogs.

Finally, the Winsock catalog functions (the *WSC* variation) also have a new version ending in 32 that follows the same rules discussed above. If a native 64-bit application needs to enumerate both the 32-bit and 64-bit catalogs, it must call *WSCEnumProtocols* to obtain the 64-bit catalog followed by *WSCEnumProtocols32* to obtain the 32-bit catalog. There is no method for a 32-bit application to obtain the 64-bit catalog, which is true for all 32-bit applications (i.e., a 32-bit application has no way of manipulating the 64-bit catalog).

Removing an LSP

Once an LSP is installed into the Winsock catalog, removing the provider is, in most cases, an easy process. The function *WSCDeinstallProvider* will remove all the catalog entries associated with the given GUID. This API is defined as

```
int WSPAPI WSCDeinstallProvider(
                IN LPGUID lpProviderId,
                OUT LPINT lpErrno
);
```

In the simple case, an LSP will require two GUIDs: one for the dummy entry and one for all of the layered chain entries. To completely remove the LSP in this case, call *WSCDeinstallProvider* once for each GUID. Of course, if the layered chain providers were installed using multiple GUIDs, the uninstall code will have to call *WSCDeinstallProvider* on each one.

Uninstalling an LSP becomes extremely complicated if after your LSP is installed, another LSP is installed over it. The second LSP's chain entries will contain references to your LSP's catalog IDs. If your LSP is blindly uninstalled, the second LSP is broken. If the second LSP is exposing itself as a TCP/IP and UDP/IP provider, most likely the system won't boot or will have no network access.

In this situation, the uninstall code for your LSP must check for any other Winsock entries that reference your LSP's catalog IDs within its protocol chain. If it finds other entries, the uninstaller must copy them, uninstall them, remove your LSP's entry from its protocol chain, and install it back into the catalog. For example, consider the case illustrated in Table 12-4. There are two LSPs in this example. LSP1 is installed over the base TCP/IP and UDP/IP providers, and LSP2 is installed over LSP1's TCP/IP and UDP/IP providers. To remove LSP1, we must also fix LSP2's entries so that they no longer reference catalog ID's 1010 and 1011. To do this, the uninstaller for LSP1 must find all entries that reference any of LSP1's catalog Ids, which in this case are entries 1021 and 1022. These entries should be saved off and uninstalled. Then the saved entries should be modified so that their chain lengths are 2 and any reference to 1010 or 1011 in their protocol chains are removed. Finally, these entries should be installed under the same GUID as before. The entry for 1021 should have a protocol chain of 1020 followed by 1001 and the entry for 1022 should be 1020 and 1002. Note that after re-installing the provider, the entry will appear at the end of the catalog when enumerated. It may be necessary to reorder the catalog to put the LSP2 entries back to their original locations.

Table 12-4 *Winsock Catalog with Multiple LSPs*

Catalog ID	Description	Address Family/Protocol	Chain Length	Protocol Chain
1021	LSP2	TCP/IP	3	1020 —> 1010 —> 1001
1022	LSP2	UDP/IP	3	1020 —> 1011 —> 1002
1020	LSP2 Dummy	N/A	0	N/A
1010	LSP1	TCP/IP	2	1009 —> 1001
1011	LSP1	UDP/IP	2	1009 —> 1002
1009	LSP1 Dummy	N/A	0	N/A
1001	MSAFD TCP/IP	TCP/IP	1	N/A
1002	MSAFD UDP/IP	UDP/IP	1	N/A

Modifying LSP Entries

As you can see, uninstalling a provider when other providers have layered over it is a horrendous task—especially in the complicated cases in which the second provider layered over yours is layered over many other providers and all installed with the same GUID! As a result, Windows XP introduces a new API designed to ease the pain of uninstalling providers. This API allows you to modify an existing provider without uninstalling it. This function is *WSCUpdateProvider* and is defined as

```
int WSPAPI WSCUpdateProvider(
                IN LPGUID lpProviderId,
                IN const WCHAR FAR *lpszProviderDllPath,
                IN const LPWSAPROTOCOL_INFOW lpProtocolInfoList,
                IN DWORD dwNumberOfEntries,
                OUT LPINT lpErrno
);
```

The parameter list is the same as for *WSCInstallProvider* except that instead of installing a new provider in the catalog, this API updates the providers referenced by the supplied GUID. With this API you can update any of the fields within an existing provider's entry except for its provider ID (GUID). So for the example given in Table 12-4, this makes fixing LSP2 almost trivial because all that needs to be modified are the protocol chains and chain length.

Writing the Layered Provider

As we mentioned previously, an LSP is implemented in a DLL. There are three important parts to layered providers: the *WSPStartup* function, tracking socket handles, and handling the various I/O models (such as *select*, *WSAEventSelect*, *WSAAsyncSelect*, overlapped, and completion ports). Of course, this doesn't include the difficulty of implementing the functionality that the LSP provides (such as HTTP proxy, and SSL sockets).

In addition to the three important tasks there is the matter of handling the Microsoft-specific Winsock extensions such as *AcceptEx* and *TransmitFile*. This topic is covered after the main three tasks. Finally, there are a few minor tasks that an LSP must implement to achieve 100 percent compatibility. The last few sections discuss these tasks in detail.

Initializing the Provider

Each LSP must implement and export the *WSPStartup* function. The function is prototyped as

```
int WSPAPI WSPStartup(
                WORD wVersion,
                LPWSPDATA lpWSPData,
                LPWSAPROTOCOL_INFOW lpProtocolInfo,
                WSPUPCALLTABLE UpCallTable,
                LPWSPPROC_TABLE lpProcTable
);
```

The first parameter is the Winsock version that the application requested. The *lpWSPData* parameter is a *WSPDATA* structure, which is a subset of the WSADATA structure seen in Chapter 1. The LSP must fill in the WSPDATA structure provider to indicate the Winsock version supported. The *lpProtocolInfo* structure is a *WSAPROTOCOL_INFOW* structure corresponding to the provider being loaded. With an LSP, this is one of the protocol structures of our LSP. The *UpCallTable* parameter is an input parameter that contains function pointers to various Winsock support routines which we will discuss later. Finally, the *lpProcTable* is a structure of function pointers to those Winsock provider functions that the LSP implemented that it must completely fill in before returning.

Before getting into the specifics of initializing a layered service provider, let's look at what happens when the system invokes an LSP's *WSPStartup* function. When an application calls *WSAStartup*, the system takes no action and it's not until the application actually creates a socket that a provider's *WSPStartup* is called. When the application creates the socket, the system searches the Winsock catalog for a matching entry as described in Chapter 2. When the matching entry is found, the system loads the provider's DLL and invokes its *WSPStartup* function.

The second issue is how many times you can expect your LSP's *WSPStartup* to be invoked, which depends on how the LSP is installed. For example, consider a system with two layered protocol entries, as shown in Figure 12-4. Two layered protocol entries are on the left side of the diagram: one layered over the MSAFD TCP/IP provider and the other layered over the MSAFD IPX provider. Note that both of these entries were installed in the same call to *WSCInstallProvider* because they both contain the same provider GUID. If an application creates a TCP/IP socket, at that point the system loads the LSP and calls its *WSPStartup* function. Afterward, if the application creates an IPX socket, the system will not invoke the *WSPStartup* function again as it has already been invoked for the provider with GUID A. Also, any further TCP/IP or IPX sockets created will not invoke additional calls to *WSPStartup*.

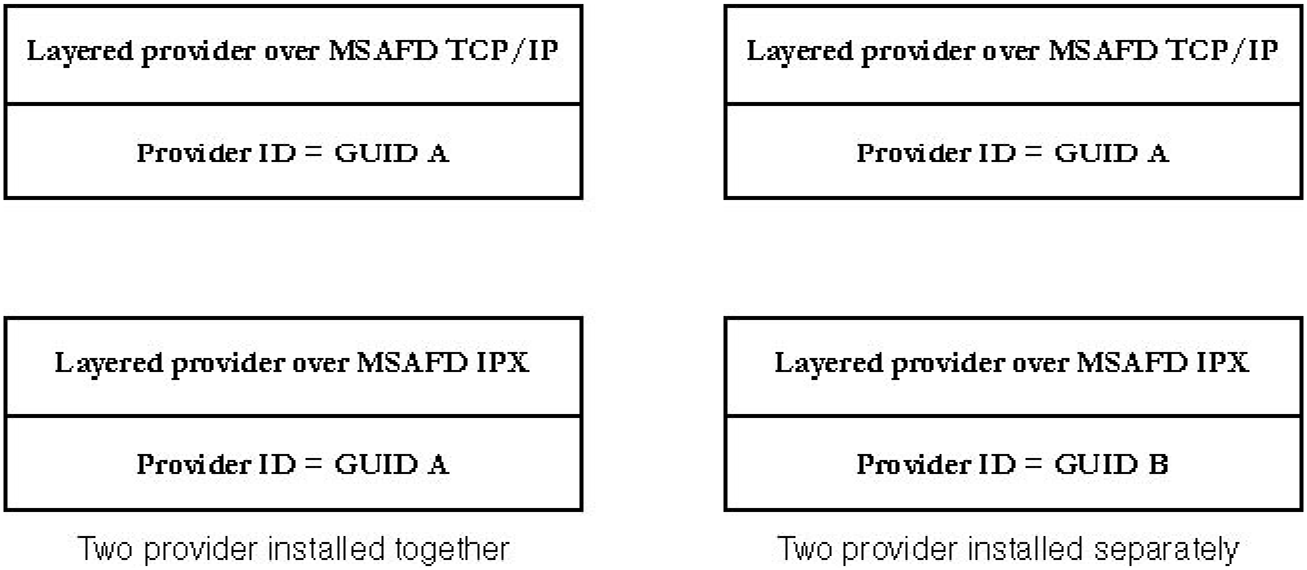

Layered provider over MSAFD TCP/IP
Provider ID = GUID A
Layered provider over MSAFD TCP/IP
Provider ID = GUID A
Layered provider over MSAFD IPX
Provider ID = GUID A
Layered provider over MSAFD IPX
Provider ID = GUID B
Two provider installed together
Two provider installed separately

Figure 12-4 *WSPStartup calling behavior*

However, if the layered protocol entries for TCP/IP and IPX were installed separately with two calls to *WSCInstallProvider* (and therefore two different GUIDs), when the application creates the first TCP/IP socket the LSP's *WSPStartup* is invoked. Then when the application creates an IPX socket the system invokes the *WSPStartup* function once more as the provider corresponding to GUID B has not been initialized yet.

This is an important consideration because it dictates how much initialization overhead is required. For example, consider the LSP that layers over every installed protocol—such as a content filter. In this case, there could be multiple protocols and multiple provider entries for each protocol. If the

LSP's layered protocol entries are installed all at once under the same GUID, the LSP's *WSPStartup* will be called no more than once when the application creates a socket. However, most applications tend to create sockets from a single address family that corresponds to two or three provider entries (for example, IPv4 has three entries: TCP/IP, UDP/IP, and RAW/IP). In this case, the LSP may require setting up multiple internal data structures for each provider: IPv4, IPv6, IPX/SPX, NetBIOS, AppleTalk, and IrDA. However, if the LSP's layered protocol entries are installed in groups corresponding to the different address families, the necessary internal data structure can be allocated only when the calling application decides to use that particular protocol. The decision to install under a single or multiple GUIDs is completely up to the LSP implementer, but it is a good idea to know how often *WSPStartup* will be invoked.

Getting back to the initialization steps, here are the three tasks an LSP must perform within its *WSPStartup* function:

1. Keep track of how many times *WSPStartup* has been invoked.
2. Initialize the *lpWSPData* and *lpProcTable* parameters.
3. Find its location within the protocol chain and initialize the lower layer(s).

The first requirement is fairly simple. The LSP should keep track of how many times *WSPStartup* has been called so that each time it is invoked a simple reference count should be incremented. A provider may need to initialize some internal data structures, which will most likely occur on the first call to *WSPStartup*. Likewise, an LSP must implement *WSPCleanup*, at which point the reference count should be decremented. Once the count reaches zero, any internal data structures and all other dynamically allocated resources should be freed.

The second requirement is also simple. The LSP must initialize the *WSPDATA* and *WSPPROC_TABLE* parameters. For the *WSPDATA* structure, the LSP can either verify that the Winsock version is correct itself or it may pass the *lpWSPData* parameter into the lower provider's *WSPStartup* function when initializing the lower layers (discussed next) if the LSP does not want to validate the parameters. The *WSPPROC_TABLE* is a giant structure containing function pointers for all the functions implemented in the LSP. The layered service provider ***must*** initialize every pointer.

The third requirement is a bit more complex. The layered provider needs to “load” the providers appearing beneath it in the protocol chain. If the LSP is layered above multiple lower layers, load the provider underneath each of the LSP's provider entries. How does the LSP find where it resides within the chain? The *lpProtocolInfo* parameter passed to *WSPStartup* is the provider entry for one of the LSP's protocol chain entries. As we touched on previously, this will match one of the LSP's providers depending on the type of socket the application created first.

As we mentioned, the system passes a *WSAPROTOCOL_INFOW* structure of the layered provider corresponding to the socket that the application is creating from one of the LSP's layered protocol entries. The first entry within the protocol chain array is the catalog ID for the LSP. Given this value, the Winsock catalog can be enumerated and the remaining layered chain providers can be found. The second index of the chain array of each LSP entry contains the catalog ID of the underlying provider that needs to be loaded.

Loading the underlying provider is a simple process. For each underlying provider, call *WSCGetProviderPath* to obtain the location of the DLL implementing that provider. This function is prototyped as

```
int WSCGetProviderPath(
                LPGUID lpProviderId,
                LPWSTR lpszProviderDllPath,
                LPINT lpProviderDllPathLen,
                LPINT lpErrno
);
```

After obtaining the provider's DLL path, *LoadLibrary* is called on it, followed by *GetProcAddress* for that DLL's *WSPStartup* function. Initializing the lower layer is simply calling that DLL's *WSPStartup*. Again, the *lpWSPData* passed to your LSP's *WSPStartup* can be passed to the lower provider's *WSPStartup* call so that it can verify the version requested is correct.

> For Windows 95, Windows 98, and Windows Me, a provider's DLL path is always returned as a UNICODE string so that it must be converted to ANSI and LoadLibraryA must be used.

Each lower provider initialized by calling its *WSPStartup* must follow the same rules that are applied to your LSP's *WSPStartup*. You must pass in a *WSPDATA* structure as well as the *WSAPROTOCOL_INFOW* structure for that provider's entry regardless of whether it is a base provider or another layered provider. For example, if the underlying provider is the MSAFD TCP/IP provider, the *WSAPROTOCOL_INFOW* structure passed is the MSAFD TCP/IP provider and not the LSP's layered protocol entry. Each lower provider will initialize the *WSPPROC_TABLE* passed into it with a list of its function pointers. The LSP must save off this function table. When an application makes a Winsock call into your LSP, your LSP must eventually call the lower provider's corresponding Winsock function (unless the LSP's purpose is to prevent or prohibit that action).

In the sample LSP provided on the companion CD, the following structure is allocated for each layered chain entry that comprises our LSP:

```
typedef struct _PROVIDER {
    WSAPROTOCOL_INFOW NextProvider,
                      LayeredProvider;
    WSPPROC_TABLE     NextProcTable;
    EXT_WSPPROC_TABLE NextProcTableExt;
    WCHAR             ProviderPathW[MAX_PATH],
                      LibraryPathW[MAX_PATH];
    char              ProviderPathA[MAX_PATH],
                      LibraryPathA[MAX_PATH];
    int               ProviderPathLen;
    HINSTANCE         hProvider;
    LPWSPSTARTUP      lpWSPStartup;
    SOCK_INFO        *SocketList;
} PROVIDER;
```

This structure keeps track of a layered chain entries' protocol structure (*LayeredProvider*) as well as the underlying provider's protocol structure (*Next-Provider*). The underlying provider's procedure table is stored in *NextProcTable* and *NextProcTableExt* is our own structure of Microsoft-specific Winsock functions that the lower provider exposes. We'll discuss these entries in detail later. In addition, both the UNICODE and ANSI versions of the provider's path are saved (*ProviderPath*). The *LibraryPath* fields contain the same data as the *ProviderPath* field except that all system variables are expanded via the *ExpandEnvironmentStrings* API. *hProvider* saves off the handle returned from *LoadLibrary* and *lpWSPStartup* contains the address for that DLL's *WSPStartup* function. Lastly, *SocketList* is a linked list of all sockets this provider created. This field will become important later.

Before going any farther, let's summarize the steps involved in initializing an LSP:

1. Verify the correct Winsock version requested.
2. Increment the reference count.
3. Save the *WSPUPCALLTABLE* passed in.
4. Find the Winsock providers layered underneath this LSP's providers. (Note that this may be a subset if this LSP's layered protocol entries were installed under separate GUIDs.)
5. Allocate a *PROVIDER* structure for each layered entry.
6. Load each underlying provider's DLL and invoke its *WSPStartup*.
7. Save the *WSPPROC_TABLE* returned from each underlying provider.

If any one of these steps fails, the LSP's *WSPStartup* should return an error. There are several relevant Winsock error codes usually associated with startup, which are

- *WSAEINVALIDPROCTABLE* Indicates the lower layer returned an invalid proc table (for example, one or more entries are NULL).
- *WSAEPROVIDERFAILEDINIT* Indicates the LSP encountered an error that prevents it from initializing properly.

Of course, if there is a more specific Winsock error code for the type of error encountered, that should be used. For example, if during startup the LSP dynamically allocates memory but that call fails, *WSAENOBUFS* should be returned.

Creating Sockets

The next important task of layered service providers is creating socket handles. The sequence of events covered so far is the application creates a socket whose parameters match an entry of our LSP. Next, the system loads our LSP by calling its *WSPStartup*, and obtains the LSP's function dispatch table, and calls our LSP's *WSPSocket* function to create a socket to return to the application. Because we are dealing with layered providers and not transport providers, the LSP has no way of creating true operating system handles. As a result, the “real” socket handle is obtained by calling the underlying provider's *WSPSocket* function. Remember that we obtained the lower provider's *WSPPROC_TABLE* when it was loaded by our LSP's *WSPStartup*.

Within your LSP's *WSPSocket* function, it must validate the address family, socket type, and protocol parameters, and find which underlying provider should be used—assuming your LSP is installed over multiple entries. Keep in mind that for some transport protocols it is valid for the protocol parameter to the socket creation API call to be zero. For example, if our LSP is layered over MSAFD TCP/IP and MSAFD UDP/IP and the application makes the following socket call: `s = socket(AF_INET, SOCK_STREAM, 0);` our LSP's *WSPSocket* function will be called with those same parameters. The LSP must then determine that this matches the LSP's entry layered over MSAFD TCP/IP. Then we must find the function table returned from our startup call to the DLL implementing MSAFD TCP/IP, at which point its *WSPSocket* can be called. In addition, the calling application may pass in the *WSAPROTOCOL_INFO* structure, which will belong to the LSP. Before creating a socket from the lower provider, its *WSAPROTOCOL_INFO* structure should be substituted.

Once a socket is created from the underlying provider, there are two options. The first is to simply return that socket handle. The problem with this is there will be no way to modify or monitor data sent or received on that socket. In the next section, we will go into detail on why this is the case. The second option is to return a “dummy” handle. The LSP then associates this dummy handle with the handle that the lower provider returned. Now whenever the application calls a Winsock API with our dummy handle, it is routed into our LSP, at which point the LSP finds the lower provider's handle associated with the dummy handle and calls the same Winsock API of the lower provider with the lower provider handle.

These dummy handles are created by calling *WPUCreateSocketHandle*. Note that the LSP cannot call this API directly. Instead, the function pointer to this API is provided in the *WSPUPCALLTABLE* passed into the LSP's *WSPStartup* routine. The prototype is

```
SOCKET WPUCreateSocketHandle(
                DWORD dwCatalogEntryId,
                DWORD_PTR dwContext,
                LPINT lpErrno
);
```

The first parameter is the catalog ID of the LSP's layered protocol chain. The second parameter is any data blob that you wish to associate with the SOCKET handle returned. The last parameter indicates the error code in case this API call fails.

Typically, the LSP creates a socket from the lower provider and then allocates a data structure that contains context information for this socket. In the sample LSP the following context structure is used:

```
typedef struct _SOCK_INFO
{
   SOCKET ProviderSocket;      // Lower provider socket handle
   SOCKET LayeredSocket;       // App's socket handle
   DWORD  dwOutstandingAsync; // Count of outstanding async operations
   BOOL   bClosing;           // Has the app closed the socket?
   volatile LONG  RefCount;  // Reference count
   DWORD  BytesSent;          // Byte counts
   DWORD  BytesRecv;
   HANDLE hIocp;             // Associated with an IOCP?
   HWND   hWnd;              // Window (if any) associated with socket
   UINT   uMsg;              // Message for socket events
   CRITICAL_SECTION  SockCritSec; // Protect access to this object
   struct _PROVIDER  *Provider;// Which provider this belongs to?
   struct _SOCK_INFO *prev,  // Used to link these structures together
                     *next;
} SOCK_INFO;
```

This is a lot of information but it is necessary for a robust LSP. The first field is the socket handle that the underlying provider returned. The second field is the handle returned from *WPUCreateSocketHandle*. When we call *WPUCreateSocketHandle* we pass the address of a *SOCK_INFO* structure as the context information. The majority of the remaining fields deal with handling socket I/O, which is discussed in the next section.

We can now construct the basic outline of the LSP's *WSPSocket* API. It would look like the following example:

```
SOCKET WSPAPI WSPSocket(
    int                 af,
    int                 type,
    int                 protocol,
    LPWSAPROTOCOL_INFOW lpProtocolInfo,
    GROUP               g,
    DWORD               dwFlags,
    LPINT               lpErrno)
{
                PROVIDER  *Provider=NULL;
                SOCK_INFO *Context;
       SOCKET  ProviderSocket,
               LayeredSocket;

       // Validate the arguments first

       // Find the PROVIDER structure for the layered protocol entry
       // that matches the given arguments - set as Provider

       // Substitute lpProtocolInfo with the lower provider's if it
       // is supplied.

       ProviderSocket = Provider->NextProcTable.lpWSPSocket(
               af,
               type,
               protocol,
               lpProtocolInfo,
               g,
               dwFlags,
               pErrno
               );
       if (ProviderSocket != INVALID_SOCKET) {
           Context = AllocateContext(); // Allocate a SOCK_INFO struct

           LayeredSocket = MainUpCallTable.lpWPUCreateSocketHandle(
               Provider->LayeredProvider.ProtocolChain.ChainEntries[0],
                (DWORD_PTR) Context,
                lpErrno
                );
            if (LayeredSocket == INVALID_SOCKET) {
                                // Handle failure
                        }
                        Context->LayeredSocket = LayeredSocket;
                        Context->ProviderSocket = ProviderSocket;
                }
                return LayeredSocket;
}
```

There are a couple of significant fields in the *SOCK_INFO* structure that should be discussed. The *bClosing* field indicates whether the application has called *WSPCloseSocket* on the dummy socket handle. If there are any outstanding I/O operations, then the LSP must not free the socket's context information until all the I/O has completed (most likely with errors). Also, a reference count is maintained (via *RefCount*) so that if the calling application is multi-threaded and one thread is using the socket and another thread closes the socket, the LSP will not free the *SOCK_INFO* structure underneath the first thread (and cause an access violation).

The LSP must implement each of the SPI functions listed in Table 12-2. For those functions that do not create socket handles (for example, everything but *WSPSocket*, *WSPAccept*, and *WSPJoinLeaf*) but take a socket handle as a parameter, it is necessary to translate the application's socket handle into the underlying handle. Remember that a SOCK_INFO structure was associated with each dummy socket handle. This context information can be retrieved by calling *WPUQuerySocketHandleContext*. Again, this function is found in the *WSPUPCALLTABLE* structure. This API is defined as

```
int WSPAPI WPUQuerySocketHandleContext(
                SOCKET s,
                LPDWORD_PTR lpContext,
                LPINT lpErrno
);
```

For example, let's take a look at how an LSP might implement the *WSPGetSockOpt* function.

```
int WSPAPI WSPGetSockOpt(
                SOCKET s,
                int level,
                int optname,
                char FAR *optval,
                LPINT optlen,
                LPINT lpErrno
)
{
                SOCK_INFO *lpContext=NULL;
                int     rc=NO_ERROR;

                rc = MainUpCallTable.lpWPUQuerySocketHandleContext(
                        s,
                        (LPDWORD_PTR)&lpContext,
                        &err
                        );
                if (rc == SOCKET_ERROR) {
                        *lpErrno = WSAENOTSOCK;
                }
                else
                {
                        rc = lpContext->Provider->NextProcTable.lpWSPGetSockOpt(
                                lpContext->ProviderSocket,
                                level,
                                optname,
                                optval,
                                optlen,
                                &lpErrno
                                );
                }
                return rc;
}
```

In this example, we first query for the context information. If it cannot be found, we return the error WSAENOTSOCK. Otherwise, we simply call the underlying provider's *WSPGetSockOpt* function with the correct socket handle. Note that in a real implementation when the socket context is looked up, the reference count would be incremented and before returning from the SPI function the reference count would be decremented. In the sample LSP, two helper routines are defined to do this. They are *FindAndLockSocketContext* and *UnlockSocketContext*, which are listed below.

```
SOCK_INFO *FindAndLockSocketContext(SOCKET s, int *lpErrno)
{
    SOCK_INFO *SocketContext=NULL;
    int        ret;

    EnterCriticalSection(&gCriticalSection);

    ret = MainUpCallTable.lpWPUQuerySocketHandleContext(
            s,
            (PDWORD_PTR) &SocketContext,
            lpErrno
            );
    if (ret == SOCKET_ERROR)
    {
        SocketContext = NULL;
    }
    else
    {
        InterlockedIncrement(&SocketContext->RefCount);
    }
    LeaveCriticalSection(&gCriticalSection);

    return SocketContext;
}

void UnlockSocketContext(SOCK_INFO *context)
{
    EnterCriticalSection(&gCriticalSection);

    InterlockedDecrement(&context->RefCount);

    LeaveCriticalSection(&gCriticalSection);
}
```

In this code sample, *gCriticalSection* is a global *CRITICAL_SECTION* object for the entire DLL. By calling *FindAndLockSocketContext* before using any socket within an SPI function (for any *WSP* function or any support function that needs to query for the socket context), we ensure that multi-threaded applications that close sockets in the middle of other operations will not cause an access violation. Of course, it is important to ensure that a corresponding call to *UnlockSocketContext* occurs before returning from the SPI function.

The last consideration for socket creation comes when the application closes a socket handle. First, query for the socket context of the supplied handle. Note that this action will cause the socket's reference count to be incremented by one (meaning that if the reference count is greater than one, another thread is accessing the structure). The next step is to close the underlying provider's socket handle, which is contained in the *ProviderSocket* field of the *SOCK_INFO* structure. This is necessary so that any outstanding I/O operations will complete with the proper error (discussed in the next section).

However, if the socket context does indicate that there is outstanding asynchronous I/O (indicated via the *dwOutstandingAsync* field of the context information) or if the reference count is greater than zero, then we cannot close the dummy socket handle yet. Instead, we mark the socket context structure as closing (by setting *bClosing* to *TRUE*). If we didn't, then if the application created another socket, the handle value could be re-used, which can cause subtle and hard-to-find problems. For example, consider the case in which two threads are accessing a socket whose handle value is 0x300. If one thread closes the socket and the second thread is about to access it, the socket is closed and the context information removed. Then a third thread creates a new socket and is assigned the handle value 0x300. The thread that was about to access the socket now looks up the context and is returned this new socket's context. At this point, the new socket may be in the wrong state (for example, not connected when it should) or could even be a socket of the wrong protocol. Whatever API uses this socket will most likely fail with a very unexpected error code such as *WSAENOTCONN* or *WSAEOPNOTSUPP*.

Only when the reference count indicates no other thread is accessing the socket context information and the outstanding operation count is zero can the dummy socket be closed and the context information be freed. After asynchronous I/O has completed and when the reference count is decremented, the *bClosing* field of the socket context should be checked. If it is *TRUE*, it indicates that the application has closed the socket and the dummy handle needs to be closed when it is safe to do so.

Once it is determined safe to close the socket, this is done with the *WPUCloseSocketHandle* API, which is another function contained in the *WSPUPCALLTABLE* structure. This function is defined as

```
int WSPAPI WPUCloseSocketHandle(
                IN SOCKET s,
                OUT LPINT lpErrno
);
```

Finally, remember that the functions *WSPAccept* and *WSPJoinLeaf* also return socket handles and the same steps just described for *WSPSocket* apply. Once a new socket handle is returned from the lower provider, an application socket is created with *WPUCreateSocketHandle*, context information is associated with it, and this application socket is returned to the caller. However, in some instances the *WSPJoinLeaf* SPI function does not create a new socket (more on this later).

Handling I/O

The last major task to creating an LSP is handling the various types of I/O an application might initiate on a socket. Remember from Chapter 5 that there are a number of I/O models an application may use: blocking sockets, *select*, *WSAAsyncSelect*, *WSAEventSelect*, overlapped I/O, and completion ports. As we mentioned previously, if an LSP wishes to modify or monitor data send or received, it must create its own socket handles with *WPUCreateSocketHandle* and must handle all possible types of I/O that may occur on the socket. In this section we'll look at each of the I/O models and discuss what steps must be made for each to work.

Before getting into each of the I/O models, it is worthwhile to mention some common rules that apply to all types of I/O. First, if an LSP needs to modify the send buffers, it should not modify the data within the application's buffer—it should make its own copy. Also, LSPs should not behave contrary to the type of I/O initiated. If an application has put a socket into non-blocking mode, the LSP should not block when handling operations that would normally fail with *WSAEWOULDBLOCK*.

Blocking and Non-blocking

For the most basic I/O blocking and non-blocking sockets there really isn't much to do. For those SPI functions that send and receive data, all the LSP needs to do is translate the socket handle to the provider's socket handle and call the lower provider's function. For example, the *WSPSend* function for an LSP would look like the following code:

```
int WSPAPI WSPSend(
    SOCKET          s,
    LPWSABUF        lpBuffers,
    DWORD           dwBufferCount,
    LPDWORD         lpNumberOfBytesSent,
    DWORD           dwFlags,
    LPWSAOVERLAPPED lpOverlapped,
    LPWSAOVERLAPPED_COMPLETION_ROUTINE lpCompletionRoutine,
    LPWSATHREADID   lpThreadId,
    LPINT           lpErrno
    )
{
                SOCK_INFO *SocketContext=NULL;
                int          ret;

                // Get the context info
                SocketContext = FindAndLockSocketContext(s, lpErrno);

                if (lpOverlapped == NULL) // Make sure this is not overlapped
{
                SetBlockingProvider(SocketContext->Provider);
                ret = SocketContext->Provider->NextProcTable.lpWSPSend(
                                SocketContext->ProviderSocket,
                                lpBuffers,
                                dwBufferCount,
                                lpNumberOfBytesSent,
                                dwFlags,
                                NULL,
                                NULL,
                                lpThreadId,
                                lpErrno
                                );
                        SetBlockingProvider(NULL);
                }
                UnlockSocketContext(SocketContext);

                return ret;
}
```

This is a very straightforward process: find the socket context and call the lower provider. For compatibility with 16-bit Winsock, we do have to keep track of which provider is blocking in case the application calls *WSACancelBlockingCall*, which is what the *SetBlockingProvider* function does. It saves off the address of our *PROVIDER* structure, which is currently issuing a blocking call for that thread. If the application calls *WSACancelBlockingCall*, all we have to do is call the blocking lower layer's *WSPCancelBlockingCall*. The *SetBlockingProvider* routine uses thread local storage to save the pointer to the current *PROVIDER* issuing a blocking call.

Select and *WSPSelect*

When an application uses the *select* API to wait for events on a set of sockets, things get a bit complicated. The *select* API will map to the SPI function *WSPSelect* and requires some work before passing the call down to the lower provider. There are three *FD_SET* structures passed in that reference the layered sockets and not the underlying provider's sockets. Because of this, the socket context for each socket contained in the *FD_SET*s must be obtained and a new *FD_SET* built that contains the lower provider's sockets.

The following code shows how to translate the *fdread FD_SET* passed into *WSPSelect*.

```
int WSPAPI WSPSelect(
    int          nfds,
    fd_set FAR * readfds,
    fd_set FAR * writefds,
    fd_set FAR * exceptfds,
    const struct timeval FAR * timeout,
    LPINT        lpErrno)
{
FD_SET ReadFds, WriteFds, ExceptFds,
int ret, HandleCount, count;

// Simple structure to quickly map LSP sockets to provider sockets
struct {
                        SOCKET LayeredSocket;
                        SOCKET ProviderSocket;
} Read[FD_SETSIZE], Write[FD_SETSIZE], Except[FD_SETSIZE];

// Translate LSP handles into provider handles
if (readfds) {
```

```
                        FD_ZERO(&ReadFds);
                        for(i=0; i < readfds->fd_count ;i++) {
                                SocketContext = FindAndLockSocketContext(
                                        (Read[i].LayeredSocket = readfds->fd_arra
                                        lpErrno
                                        );
                                Read[i].ProviderSocket = SocketContext->ProviderS
                                FD_SET(Read[i].ProviderSocket, &ReadFds);
                                UnlockSocketContext(SocketContext);
                        }
                }
                // Do the same for writefds

                // Do the same for exceptfds

                SetBlockingProvider(SocketContext->Provider);
                ret = SocketContext->Provider->NextProcTable.lpWSPSelect(
                        nfds,
                        (readfds ? &ReadFds : NULL),
                        (writefds ? &WriteFds : NULL),
                        (exceptfds ? &ExceptFds : NULL),
                        timeout,
                        lpErrno
                        );
                SetBlockingProvider(NULL);
                HandleCount = ret;
                // Map the signaled provider handles back to the LSP handles
if (readfds) {
                        count = readfds->fd_count;
                        FD_ZERO(&readfds);
                        for(i=0; (i < count) && HandleCount ;i++) {
                        5       if (MainUpCallTable.lpWPUFDIsSet(
                                                Read[i].ProviderSocket,
                                                &ReadFds))  {
                                                FD_SET(Read[i].LayeredSocket, rea
                                                HandleCount--;
                                }
                        }
                }
                // Do the same for writefds

                // Do the same for exceptfds
}
```

For this to work, a mapping is maintained between the layered sockets passed into *select* and its corresponding provider socket. This is necessary because after the lower provider's *WSPSelect* is called, the LSP has to return only those layered provider sockets that were signaled.

The first step is to go through each handle in the *FD_SET* and find its context information. The mapping of the provider socket to the layered socket is maintained in the *Read* array. We then have a second *FD_SET*, *ReadFds*, which contains the underlying provider's socket handles, which we then pass into the lower provider's *WSPSelect* function. Upon return, we know how many handles were signaled with the return value. Then it is a process of seeing which provider handles passed were signaled. This is done by calling the helper function *WPUFDIsSet* function for each provider socket passed in. If it is set, we take the associated layered socket and set it into the *readfds FD_SET* passed into the function so that upon return from the LSP's *WSPSelect* the application has the correct handles signaled. This same process has to be performed for *writefds* and *exceptfds*. Of course, the sample does not perform error handling, nor does it handle the case when a timeout value is supplied. See the sample LSP for the full implementation.

The *WPUFDIsSet* function is another helper function passed to the LSP in the *WSPUPCALLTABLE* structure. The function is defined as

```
int WSPAPI WPUFDIsSet(
                IN SOCKET s,
                IN fd_set FAR *fdset
);
```

This function behaves exactly as the *FD_ISSET* macro seen in Chapter 5.

There is one major issue frequently encountered when implementing an LSP's *WSPSelect*: what to do if one of the event handles passed in an *FD_SET* is unknown. If the LSP queries for the socket context of a given handle and it fails, should the LSP indicate an error (such as *WSAENOTSOCK*) or simply pass that event handle to the lower provider unmodified? The unique problem with the *WSPSelect* API is that it is the only Winsock function that can take multiple socket handles in a single call. For all other Winsock functions, the system knows exactly which provider should handle that API call because there is just one socket handle passed as a parameter.

Even though the Winsock specification explicitly states that only sockets from the same provider may be passed into *select*, many applications ignore this (including Microsoft Internet Explorer) and frequently pass down both TCP and UDP handles together. The base Microsoft providers do not verify that all socket handles are from the same provider. In addition, the Microsoft providers will correctly handle sockets from multiple providers in a single *select* call. This is a problem with LSPs because an LSP may be layered over just a single entry such as TCP. In this case, the LSP's *WSPSelect* is invoked with *FD_SET*s that contain their own sockets plus sockets from other providers (such as a UDP socket from the Microsoft provider). When the LSP is translating the socket handles and comes upon the UDP handle, the context query will fail. At this point, it may return an error (*WSAENOTSOCK*) or pass the socket down unmodified. If an error is returned, then for the case of an LSP layered only over UDP/IPv4 (or TCP/IPv4), Internet Explorer will no longer function. A workaround is to always install the LSP over all providers for a given address family (such as for IPv4, install over TCP/IPv4, UDP/IPv4, and RAW/IPv4). No Microsoft application or service currently passes socket handles from multiple address families into a single *select* call, although LSASS on Windows NT 4.0 used to pass IPX and IPv4 sockets together (this has been fixed in the latest service packs for Windows NT 4.0).

WSAAsyncSelect

Handling sockets that register for event notification via *WSAAsyncSelect* also require some additional help. As you recall from Chapter 5, an application registers for notification on certain events that will be posted to the given window. The problem here is that the *WPARAM* parameter posted to the application's window contains the socket handle. This is bad because the LSP will translate the handle passed into its *WSPAsyncSelect* and call the lower provider's function with the translated socket and the remaining parameters. As a result, when an event is posted, it is posted directly to the application's window handler and the *WPARAM* parameter contains the lower provider's socket and not the LSP-created socket.

To handle this case correctly, the LSP must create its own hidden window on which to receive events from the lower provider. Then within the LSP's window handler, the socket can be translated back to the application socket and posted to the application's window handler. The LSP's *WSPAsyncSelect* must save the window handle and message that is associated with the application socket. This information is saved in the socket context (for example, the *SOCK_INFO* structure of the sample LSP). The following code shows how this is handled:

```
int WSPAPI WSPAsyncSelect (
    SOCKET       s,
    HWND         hWnd,
    unsigned int wMsg,
    long         lEvent,
   LPINT        lpErrno)
{
   SOCK_INFO *SocketContext;
   int ret;

   SocketContext = FindAndLockSocketContext(s, lpErrno);
   if (SocketContext != NULL)
   {
      SocketContext->hWnd = hWnd;
      SocketContext->uMsg = wMsg;

      // Get the handle to our hidden window
      if ((hWorkerWindow = GetWorkerWindow()) != NULL)
      {
         SetBlockingProvider(SocketContext->Provider);
         ret = SocketContext->Provider->NextProcTable.lpWSPAsyncSelect(
                  SocketContext->ProviderSocket,
                  hWorkerWindow,
                  WM_SOCKET,
                  lEvent,
                  lpErrno);
          SetBlockingProvider(NULL);
      }
   }
   UnlockSocketContext(SocketContext);
   return ret;
}
```

In this code, the socket context is found and the window handle and message is saved. Then the hidden asynchronous window that the LSP created is returned via the *GetWorkerWindow* call. This routine simply creates the window and thread to handle the events (see ASYNCSELECT.CPP for the full implementation). Then the lower provider is called with the lower provider socket, except that we supply the window handle of our LSP helper window instead.

The code for our hidden window handler is simple:

```
static LRESULT CALLBACK AsyncWndProc(
    HWND hWnd,
    UINT uMsg,
    WPARAM wParam,
    LPARAM lParam)
{
    SOCK_INFO *si;

    if (uMsg == WM_SOCKET)
    {
        if (si = GetCallerSocket(NULL, wParam))
        {
            MainUpCallTable.lpWPUPostMessage(
                si->hWnd,
                si->uMsg,
                si->LayeredSocket,
                lParam);
            return 0;
        }
    }

    return DefWindowProc(hWnd, uMsg, wParam, lParam);
}
```

Here we look for only the socket notifications. The only challenge is to map the provider socket (indicated as *wParam*) back to the LSP-created socket, which is what the *GetCallerSocket* function does (defined in SOCKINFO.CPP). As you recall, the *PROVIDER* structure contains a linked list of all the *SOCK_INFO* each provider created. The *GetCallerSocket* searches all the linked lists in search of the *SOCK_INFO* that contains the given lower provider socket handle. This is necessary because there is no other convenient way of mapping provider sockets back to LSP sockets.

Once that is found, the helper function *WPUPostMessage* is called to post the event to the application's window with the correct socket handle. Remember, the window handle and message were saved earlier when the application called *WSAAsyncSelect* on the handle. This function is located in the *WSPUPCALLTABLE* and is defined as

```
BOOL WSPAPI WPUPostMessage(
    IN HWND hWnd,
    IN UINT Msg,
    IN WPARAM wParam,
    IN LPARAM lParam
    );
```

WSAEventSelect

This socket model requires no work on the LSP's part. When the select events are signaled, a simple event handle is used—no socket handles are returned, so no extra socket translation is needed. For example, the application calls *WSAEventSelect* with a socket, event, and event mask. Within the LSP's *WSPEventSelect*, the handle is translated and passed to the lower provider with the same event and event mask. When a requested event occurs on the socket, the lower provider sets the application's event to be signaled, after which the application calls a send or receive function as described previously in the section “Blocking and Non-blocking.”

Unless the LSP is required to intercept these event notifications (as determined by the LSP's actual purpose), there is no need to substitute our own event handle to wait for notification from the lower layer. If you did, once your substituted event was signaled, the LSP would perform the necessary computation and then signal the application's event (which must be saved in the socket context) with *WSASetEvent*.

Overlapped I/O

Handling overlapped I/O is another complicated issue that depends on what platforms the LSP is to be installed on. The easiest and most elegant method is to handle all application-initiated overlapped I/O by using a I/O completion port regardless of whether the application is using events, callbacks, or completion ports. However, if the LSP is to be run on Windows 95, Windows 98, or Windows Me, this is impossible. The sample LSP provider handles both cases.

In the case of Windows NT and I/O completion ports, the LSP creates a completion port and a worker thread that services the completion notifications by calling *GetQueuedCompletionStatus*. When an application makes an overlapped I/O call, the LSP first checks to see if the lower provider handle has been associated with the LSP's completion port yet. This information is contained in the socket context information as the *hIocp* field. If the lower provider socket has been associated, this field is non-NULL; otherwise, it contains the handle of the LSP's completion port.

Once the provider socket is associated with the LSP's completion port, the I/O is posted on the lower provider's socket handle. Once it has completed, the LSP worker thread will receive the notification and the LSP can complete the application's request. After the LSP receives completion notification, the application's overlapped I/O is completed so that the application will receive notification either via event, asynchronous procedure call, or its own completion port.

Each Winsock SPI function that can be made in an overlapped fashion (including Microsoft extension functions) requires special handling. First, the LSP must keep track of the *WSAOVERLAPPED* structure the application passed into the function. It maintains useful information such as indicating I/O in progress, error codes, and bytes transferred. To perform this function, the LSP defines its own *WSAOVERLAPPED* structure to maintain information about each overlapped I/O operation posted to the lower provider's socket. This structure is defined as

```
typedef struct _WSAOVERLAPPEDPLUS
{
    WSAOVERLAPPED  ProviderOverlapped;  // passed to lower provider
    PROVIDER      *Provider;            // lower provider info
    SOCK_INFO     *SockInfo;            // socket info for this op
    SOCKET         CallerSocket;        // app (LSP) socket
    SOCKET         ProviderSocket;      // lower provider socket
    HANDLE         Iocp;                // LSP completion port
    int            Error;               // error code?
    union                               // Arguments to operation
    {
        ACCEPTEXARGS         AcceptExArgs;
        TRANSMITFILEARGS     TransmitFileArgs;
        CONNECTEXARGS        ConnectExArgs;
        TRANSMITPACKETSARGS  TransmitPacketsArgs;
        DISCONNECTEXARGS    DisconnectExArgs;
        WSARECVMSGARGS      WSARecvMsgArgs;
        RECVARGS            RecvArgs;
        RECVFROMARGS        RecvFromArgs;
        SENDARGS            SendArgs;
        SENDTOARGS          SendToArgs;
        IOCTLARGS           IoctlArgs;
    };
#define LSP_OP_IOCTL                1     // WSPIoctl
#define LSP_OP_RECV                 2     // WSPRecv
#define LSP_OP_RECVFROM             3     // WSPRecvFrom
#define LSP_OP_SEND                 4     // WSPSend
#define LSP_OP_SENDTO               5     // WSPSendTo
#define LSP_OP_TRANSMITFILE         6     // TransmitFile
#define LSP_OP_ACCEPTEX             7     // AcceptEx
#define LSP_OP_CONNECTEX            8     // ConnectEx
#define LSP_OP_DISCONNECTEX         9     // DisconnectEx
#define LSP_OP_TRANSMITPACKETS     10     // TransmitPackets
#define LSP_OP_WSARECVMSG          11     // WSARecvMsg
    int             Operation;            // Type of operation this is
    LPWSATHREADID   lpCallerThreadId;     // Caller thread
    LPWSAOVERLAPPED lpCallerOverlapped;  // App's WSAOVERLAPPED struct
    LPWSAOVERLAPPED_COMPLETION_ROUTINE   lpCallerCompletionRoutine;
    _WSAOVERLAPPEDPLUS                *next;
} WSAOVERLAPPEDPLUS, * LPWSAOVERLAPPEDPLUS;
```

As you can see, there is a lot of information maintained for each overlapped operation that the application initiates. We won't go into detail about all of these fields because many of them are self explanatory. Instead, let's walk through what needs to occur when handling an overlapped call. The steps are:

1. Allocate an LSP-overlapped context structure (for example, the *WSA-OVERLAPPEDPLUS* object shown in the last listing).
2. Save the caller's *WSAOVERLAPPED* pointer in the *lpCallerOverlapped* structure.
3. If a completion routine is supplied, save it as *lpCallerCompletionRoutine*.
4. Mark the caller's *WSAOVERLAPPED* structure as pending by setting the *Internal* field to *WSS_OPERATION_IN_PROGRESS* (defined in WS2SPI.H).
5. Make sure the lower provider's socket is associated with the LSP's completion port.
6. Call the same SPI function in the lower provider with the lower provider's socket and the *ProviderOverlapped* field of the *WSAOVERLAPPEDPLUS* structure for this operation.
7. Return *SOCKET_ERROR* and set the error code to *WSA_IO_PENDING*.

This lists the minimum steps required. The sample LSP does a few extra steps. First, it saves the caller's parameters, such as buffer pointers and flags. These are saved in the unnamed union within the *WSAOVERLAPPEDPLUS* structure. The union contains a structure for each overlapped enabled Winsock function—each containing fields corresponding to their respective Winsock functions' parameter lists. The sample LSP doesn't use the saved parameters but it may be necessary to do so for an LSP that performs a specific task. One important item to note is that some of the pointer parameters supplied can be stack-based and therefore the LSP ***cannot*** just capture the pointer values. For example, *WSASend*, *WSASendTo*, *WSARecv*, and *WSARecvFrom* take an array of *WSABUF* structures that contain the send or receive buffers. This array can be stack-based, which means as soon as the application calls the Winsock function it may return from the calling function or free that memory (if dynamically allocated). The LSP must copy the buffer pointers into its own allocated *WSABUF* structures.

Once the overlapped I/O has been posted to the lower provider, it's simply a matter of waiting for the LSP's completion thread to receive notification for that operation. When the completion thread receives notification, the pointer to the *WSAOVERLAPPED* structure for the operation returned from *GetQueuedCompletionStatus* is actually our *WSAOVERLAPPEDPLUS* structure. The following three steps need to be performed to finish this operation.

1. Call the lower provider's *WSPGetOverlappedResult* to get bytes transferred, flags, and the appropriate error code in case of a failure.
2. Update the caller's *WSAOVERLAPPED* structure with *Offset* equal to the error (if any), *OffsetHigh* to the flags returned (if any), and *InternalHigh* to the bytes transferred.
3. Complete the application's overlapped request using either *WPUQueueApc* or *WPUCompleteOverlappedRequest* depending on whether a completion function was supplied.

The last step is what notifies the application that its I/O operation has completed. If a completion routine was supplied, the LSP needs to execute that function. This is performed by the *WPUQueueApc* function, which is a field of the *WSPUCALLTABLE* structure and is defined as

```
int WSPAPI WPUQueueApc(
    IN LPWSATHREADID lpThreadId,
    IN LPWSAUSERAPC lpfnUserApc,
    IN DWORD_PTR dwContext,
    OUT LPINT lpErrno
    );
```

The first parameter is the thread ID of the application's thread that initiated this I/O because the completion routine must fire within the context of that thread. If you recall, this is one of the parameters saved in the *WSAOVERLAPPEDPLUS* structure when the application initiated the I/O. The second parameter is the application's completion function to call. The *dwContext* is the caller's original *WSAOVERLAPPED* structure, and *lpErrno* returns an error if *WPUQueueApc* fails.

If the application did not specify a completion routine and supplied only a *WSAOVERLAPPED* structure, the LSP completes the I/O with *WPUComplete-OverlappedRequest*. It's curious to note that this function is not a member of the *WSPUPCALLTABLE*. Instead it is contained in WS2_32.DLL and is called normally. This function is defined as

```
int WSPAPI WPUCompleteOverlappedRequest (
    SOCKET s,
    LPWSAOVERLAPPED lpOverlapped,
    DWORD dwError,
    DWORD cbTransferred,
    LPINT lpErrno
);
```

The parameter list is easy. The *SOCKET* parameter is the application's socket and *lpOverlapped* is its *WSAOVERLAPPED* structure. *dwError* is the error if the call failed (otherwise, it should be *NO_ERROR*). *cbTransferred* is the number of bytes transferred in the operation. The *lpErrno* parameter returns the error code if the *WPUCompleteOverlappedRequest* call fails.

You will notice that an overlapped operation that the LSP handles automatically fails with *WSA_IO_PENDING* even though it is possible that when the LSP makes the call to the lower provider, that overlapped operation could succeed immediately. The LSP does not do this because regardless of whether the operation succeeds immediately, notification will ***always*** be posted to the completion queue. The code is a bit cleaner by always processing completion notifications in the worker thread in addition to being perfectly legal according to the Winsock specification. Care must be taken to ensure that the calling application receives only one notification per I/O operation. The sample LSP provided always returns pending and waits for the completion thread to receive the notification before completing the request.

Handling overlapped I/O on Windows 95, Windows 98, and Windows Me is a bit more challenging. There are two possible approaches. First, the LSP can issue the overlapped I/O to the lower layer and use events for completion notification. The drawback to this, as we saw in Chapter 5, is a single thread can only wait on *MAXIMUM_WAIT_OBJECTS* event handles (which is currently 64). The other method is to use completion functions, which is easier to implement.

When the calling application issues an overlapped request, the LSP builds a *WSAOVERLAPPEDPLUS* structure as we described earlier and then this object is placed in a queue. For this model, we still use a worker thread whose purpose is to wait for overlapped requests to be placed in the queue. Once the worker thread is notified of available work items, it removes an item from the queue and actually makes the requested overlapped operation (by calling the lower provider). It is important that these overlapped operations are executed in the context of the LSP thread and not an application thread. The calling thread must be in an alertable wait state for the completion functions to execute. Because the calling application should not have to be aware if the Winsock provider is layered, it most likely will not go into an alertable wait unless the application is using completion functions (which it may). As a result, the LSP's worker thread executes the overlapped requested and when not servicing work items, it remains in an alertable wait state.

Note that when the LSP issues the overlapped I/O with a completion function, the completion function supplied is the LSP's, not the application's. Once the LSP's completion function fires, the LSP will post the completion to the application via whatever notification mechanism the application supplied (such as signaling the event or firing the completion function).

Winsock Extension Functions

For LSPs that create their own sockets, they must also handle the Winsock-specific extension functions that take socket handles as parameters. This includes *AcceptEx*, *TransmitFile*, *ConnectEx*, *DisconnectEx*, *TransmitPackets*, and *WSARecvMsg*. This is done within the LSP's *WSPIoctl* function. When an application loads a Microsoft-specific function, it will call *WSAIoctl* with the ioctl code *SIO_GET_EXTENSION_FUNCTION_POINTER*. The LSP simply has to determine which function is being loaded via the *InBuffer* parameter, which contains the GUID for the requested function. Once that is done, the LSP returns the address of its own extension function. This extension function will then translate all the socket handles and load the extension function of the lower layer, which will be invoked by the LSP. This works even if the application uses the *TransmitFile* and *AcceptEx* functions exported directly from MSWSOCK.DLL because those functions simply end up calling *WSAIoctl* with *SIO_GET_EXTENSION_FUNCTION_POINTER*.

The sample LSP will then implement its own extension functions in EXTENSION.CPP. The implementation of these functions is the same that it is for the other SPI functions. The LSP must translate the handle, validate arguments as necessary, and handle the possibility of overlapped I/O. The code for *WSPIoctl* is contained in SPI.CPP and you'll notice that the first step done is check to see if the ioctl code is *SIO_GET_EXTENSION_FUNCTION_POINTER*.

Miscellaneous Requirements

This section is devoted to the miscellaneous tasks that an LSP must perform to behave properly. In this section, we'll cover each service provider API in which an LSP must perform a special action.

WSPGetSockOpt

When the calling application calls the LSP's *WSPGetSockOpt* with *SO_PROTOCOL_INFOA* or *SO_PROTOCOL_INFOW*, the LSP should return its own protocol info structure and not translate the handle to pass to the lower provider. If that were the case, the call would return the lower provider's *WSAPROTOCOL_INFO* structure instead of the LSP's. Note that both the ANSI and UNICODE versions must be supported, so the LSP may have to perform the appropriate string conversions on the returned structure.

WSPSetSockOpt

After an application calls *AcceptEx*, it typically calls *setsockopt* with *SO_UPDATE_ACCEPT_CONTEXT*. The argument passed to *WSPSetSockOpt* is the socket handle of the accepted socket. The LSP must translate that handle as well as the listening socket handle before passing the call to the lower provider.

WSPIoctl

There are a couple of ioctl codes that an LSP must handle differently. We've already mentioned that if an LSP is implementing its own extension functions (which it must if returning its own handles), it must capture the *SIO_GET_EXTENSION_FUNCTION_POINTER* command. In addition, it must capture the *SIO_QUERY_TARGET_PNP_HANDLE*. The handles *WPUCreateSocketHandle* created are not true plug-and-play handles and cannot receive notifications. As a result, applications can use *SIO_QUERY_TARGET_PNP_ HANDLE* to obtain the base provider's socket handle. The LSP should return the lower provider's socket handle in the return buffer.

WSPJoinLeaf

The *WSAJoinLeaf* function is a bit odd. Depending on the protocol, the return value is either a new socket handle (as in ATM) or the same handle passed in as the *s* parameter (as in IP multicasting). See Chapter 9 for more information about *WSAJoinLeaf* and its behavior with the various multicast enabled protocols. Currently, Ipv4 and Ipv6 are the only protocols that do not create new socket handles when *WSAJoinLeaf* is called. If your LSP is to be layered over IP, it should take this into account. Otherwise, if it did create new handles including the context information, these structures would be leaked because the calling application will call *closesocket* on just one of the handles.

WSPAddressToString and *WSPStringToAddress*

These functions are unique because they do not take a socket parameter. Instead, the *WSAPROTOCOL_ INFOW* structure of the LSP entry that matches the given address is passed in. The LSP should find the provider layered beneath the supplied *WSAPROTOCOL_INFOW* structure and call that provider's corresponding SPI function. The only rule is if the underlying provider is ***not*** a base provider, the *WSAPROTOCOL_INFOW* structure should be passed unmodified. Otherwise, if the underlying provider is a base provider, the LSP should substitute the base provider's *WSAPROTOCOL_INFOW* structure.

Debugging an LSP

Developing an LSP is a complicated task in which one mistake will probably break all applications accessing Winsock for the protocols the LSP is layered over. In the event of IP, critical services such as LSASS will fail. If this does happen, booting into safe mode and uninstalling the LSP will return the Winsock catalog back to normal. Also, it is a good idea to smoke test the LSP before rebooting the system. Internet Explorer is always a good test application (when the LSP is layered over IP). Otherwise, it may be necessary to write a small suite of test applications to verify the LSP's functionality.

For tracking down minor problems with applications, printing debug messages to the debugger can be invaluable. The sample LSP we've provided uses *OutputDebugString* in several places; it also has the ability to turn on verbose debugging by defining *DEBUG* and *DEBUGSPEW* for the project. Using message boxes for debug messages is a bad idea because during the boot process several system services can load the DLL before the user interface subsystem is fully initialized, which will cause the LSP DLL to fail during load.

For especially difficult problems, it is often necessary to use a debugger to determine the point of failure. For interactive user applications, the Visual Studio debugger, as well as the NT Symbolic Debugger (NTSD)—a text-mode debugger available with the Platform SDK—are both excellent choices. In general tracing the steps of socket creation through the various APIs called on that socket will track down the problem. For NTSD, this is accomplished by enabling “break on load” (for example, the NTSD command is *sxeld*) for each DLL loaded until the LSP DLL is loaded. At this point, breakpoints may be set for the LSP's functions of interest (such as *WSPStartup*, *WSPSocket*, and *WSPConnect*).

If problems occur with system services such as LSASS during boot, debugging is much more complicated. This requires a kernel mode debugger to be attached to the machine running the LSP. Then it is possible to attach NTSD to the failing system service and pipe the NTSD console to the kernel debugger running on the second machine. For information about using and setting up the various types of debuggers, consult the Microsoft Developer Network (MSDN) online at http://msdn.microsoft.com.

LSP Sample

Throughout this discussion we have referred to the sample LSP on the CD in the directory LSP. In this section, we'll briefly describe each file of the project as well as how to install the LSP. The following is a list of files and what they implement.

- ASYNCSELECT.CPP Implements helper routines used for handling *WSAAsyncSelect*. This includes creating the hidden window for receiving events from the lower provider as well as the window procedure that services those notifications.
- EXTENSION.CPP Implements all of the Microsoft-specific Winsock extensions available, such as *AcceptEx*, *TransmitFile*, *TransmitPackets*, *ConnectEx*, *DisconnectEx*, and *WSARecvMsg*.
- INSTLSP.CPP Implements the installation and removal code. This file is compiled into an .EXE that will install and/or remove the LSP from the Winsock catalog.
- OVERLAP.CPP Implements handling overlapped I/O for the LSP. For Windows NT, this includes creating the completion port as well as the worker thread for handling completion notifications. For Windows 95, Windows 98, and Windows Me, this includes establishing a work item queue and a worker thread that services I/O placed within the queue.
- PROVIDER.CPP Implements common routines for enumerating the Winsock catalog as well as defining the GUID under which the LSP is installed. These routines are used by both the LSP DLL and the installation utility (INSTLSP.CPP).
- SOCKINFO.CPP Implements common routines for looking up associated socket context structures for sockets that the LSP creates. This file also contains functions for allocating and freeing *SOCK_INFO* structures in addition to inserting and deleting them from the PROVIDER structures (which maintain a list of all sockets that provider created).
- SPI.CPP This is the “guts” of the LSP. It defines all of the WSP* functions, including *WSPStartup*.

Name Space Providers

In Chapter 8, we showed you how an application can register and resolve services within a name space, which is an especially powerful feature for services that might be dynamically created on the network. Unfortunately, the existing name spaces available are limited in their usefulness. The Winsock 2 specification, however, provides a method for creating your own name spaces in which you can handle name registration and resolution in whatever manner you prefer.

This is accomplished by creating a DLL that implements the nine name space functions. These functions all begin with the NSP prefix and are companions to the RNR functions from Chapter 8. For example, the name space function equivalent to *WSASetService* is *NSPSetService*. After the DLL is created, it is then installed into the system catalog with a GUID that identifies the name space. Once this is done, applications can register and query services in your name space.

With Windows XP, a new registration and name resolution function was added: *WSANSPIoctl*. This new function allows for applications to initiate a lookup via the *WSALookupService* APIs and then use the returned handle in a call to *WSANSPIoctl* to receive information about that request. Name space providers are not required to implement their own *NSPIoctl* but can do so if they wish. Throughout our discussion, we will focus on the nine NSP functions that must be implemented but will cover *NSPIoctl* later.

In this section, we'll first present how to install a name space provider, and then we'll describe the functions a name space provider must implement. Finally, we'll present a sample name space provider as well as a sample application that registers and resolves services.

Installing a Name Space Provider

A name space provider is simply a DLL that implements the name space provider functions. Before applications can use a name space, you must make the system aware of it via the *WSCInstallNameSpace* function. Conversely, once a provider is installed, you can either disable it or remove it altogether from the system catalog using the functions *WSAEnableNSProvider* and *WSAUnInstall-NameSpace*, respectively. We will describe each of these functions next.

WSCInstallNameSpace

This function is used to install a provider into the system catalog and is declared as

```
int WSCInstallNameSpace (
    LPWSTR lpszIdentifier,
    LPWSTR lpszPathName,
    DWORD dwNameSpace,
    DWORD dwVersion,
    LPGUID lpProviderId
);
```

The first thing that you will notice is that all string parameters are wide character strings. Actually, all name space providers are implemented using wide character strings. We'll talk more about this later. The *lpszIdentifier* parameter is the name of the name space provider. This is the name that is enumerated when you call *WSAEnumNameSpaceProviders*, which we saw in Chapter 8. The *lpszPathName* parameter is the location of the DLL. The string can include environment variables, such as %SystemRoot%. The *dwNameSpace* parameter is a numeric identifier for the name space. For example, the header file NSPAPI.H defines constants for other well-known name spaces, such as *NS_SAP* for IPX SAP. The *dwVersion* parameter sets the version number for the name space. Finally, *lpProviderId* is a GUID that identifies the name space provider.

Upon success, *WSCInstallNameSpace* returns 0; upon failure, the function returns *SOCKET_ERROR*. The most common failures are *WSAEINVAL*, which indicates that a name space with that GUID already exists; and *WSAEACCESS*, which indicates that the calling process does not have sufficient privilege. Only Administrators group users can install a name space.

WSCEnableNSProvider

This function is used to modify the state of a name space provider. It can be used to enable or disable the provider. The function is declared as

```
int WSCEnableNSProvider (
    LPGUID lpProviderId,
    BOOL fEnable
);
```

The *lpProviderId* parameter is the GUID identifier for the name space that you want to modify. The *fEnable* parameter is a Boolean value indicating that you should either enable or disable the provider. A disabled provider is unable to process queries or registrations.

Upon success, *WSCEnableNSProvider* returns 0; upon failure, the function returns *SOCKET_ERROR*. If the provider GUID is invalid, *WSAEINVAL* is returned.

WSCUnInstallNameSpace

This function removes a name space provider from the catalog. The function is defined as

```
int WSCUnInstallNameSpace ( LPGUID lpProviderId );
```

The *lpProviderId* parameter is the GUID for the name space to remove. If the GUID is invalid, the function fails with *WSAEINVAL*.

Implementing a Name Space

A name space must implement all nine name space functions that map to the RNR functions covered in Chapter 8. In addition to implementing these functions, you must also develop a method for persisting the data. That is, you must maintain the data beyond the instance of the DLL. Every process that loads the DLL receives its own data segment, which means that data stored within the DLL cannot be shared between instances. (Actually, it is possible to share information between multiple applications that have loaded the DLL, but this practice is discouraged.) For more information about DLLs, see ***Programming Applications for Microsoft Windows, 4th Edition***, by Jeffrey Richter (Microsoft Press, 1999). Remember from Chapter 8 that there are three types of name spaces: dynamic, persistent, and static. Obviously, implementing a static name space might not be a good idea because it disallows programmatic registration of services. Later in this chapter, we'll present some ideas about how to maintain the data that the name space needs to persist.

You must also understand the importance of using wide character strings in all name space provider functions. This not only includes string parameters to functions but also strings within the RNR structures, such as *WSAQUERYSET* and *WSASERVICECLASSINFO*. You might be wondering how this is possible because when an application registers or resolves a name it can use either the normal (ASCII) or the wide character (UNICODE) version of the RNR functions and structure. Either version works because all ASCII calls go through an intermediate layer that converts all strings to wide character strings. This is true on function call and return. That is, if *WSAQUERYSET* is returned to the calling application—as with *WSALookupServiceNext*—any data that the name space provider returns is originally UNICODE and is converted to ASCII before returning from the function call. You can see that if your application uses RNR functions, calling the wide character versions will be faster because no conversions are required.

Of the nine functions that a name space provider must implement, only seven map to Winsock 2 RNR functions, as shown in Table 12-5. The remaining two functions are for initialization and cleanup. Once the name space is installed into the system, applications can use it by specifying either the GUID under which the name space was installed or the name space identifier that is also specified during installation. An application then makes calls to the standard Winsock 2 RNR function, as described in Chapter 8. When one of these functions is called, the equivalent name space provider function is invoked. For example, when an application calls *WSAInstallServiceClass*, which references the GUID for a custom name space, the function *NSPInstallServiceClass* for that provider is invoked. In the next section, we'll cover each of the name space functions.

Table 12-5 *Mapping RNR Functions to NSP Functions*

Winsock Function	Equivalent Name Space Provider Function
WSAInstallServiceClass	*NSPInstallServiceClass*
WSARemoveServiceClass	*NSPRemoveServiceClass*
WSAGetServiceClassInfo	*NSPGetServiceClassInfo*
WSASetService	*NSPSetService*
WSALookupServiceBegin	*NSPLookupServiceBegin*
WSALookupServiceNext	*NSPLookupServiceNext*
WSALookupServiceEnd	*NSPLookupServiceEnd*
WSANSPIoctl	*NSPIoctl*

NSPStartup

The *NSPStartup* function is called whenever the name space provider DLL is loaded. Your name space implementation ***must*** include this function, and it must be exported from the DLL. Any per-DLL data structures required for the provider to operate can be allocated when this function is called. *NSPStartup* is prototyped as

```
int NSPStartup (
    LPGUID lpProviderId,
    LPNSP_ROUTINE lpnspRoutines
);
```

The first parameter, *lpProviderId*, is the GUID for this name space provider. The *lpnspRoutines* parameter is an *NSP_ROUTINE* structure that your implementation of this function must fill out. This structure provides function pointers to the other eight name space functions that belong to your provider. The *NSP_ROUTINE* object is defined as

```
typedef struct _NSP_ROUTINE
{
    DWORD                        cbSize;
    DWORD                        dwMajorVersion;
    DWORD                        dwMinorVersion;
    LPNSPCLEANUP                 NSPCleanup;
    LPNSPLOOKUPSERVICEBEGIN      NSPLookupServiceBegin;
    LPNSPLOOKUPSERVICENEXT       NSPLookupServiceNext;
    LPNSPLOOKUPSERVICEEND        NSPLookupServiceEnd;
    LPNSPSETSERVICE              NSPSetService;
    LPNSPINSTALLSERVICECLASS     NSPInstallServiceClass;
    LPNSPREMOVESERVICECLASS      NSPRemoveServiceClass;
    LPNSPGETSERVICECLASSINFO     NSPGetServiceClassInfo;
    // NSPIoctl is a new API added in Windows XP
    LPNSPIOCTL                   NSPIoctl;
} NSP_ROUTINE, FAR * LPNSP_ROUTINE;
```

The first field, *cbSize*, indicates the size of the *NSP_ROUTINE* structure. The next two fields, *dwMajorVersion* and *dwMinorVersion*, are included for versioning your provider. The versioning is arbitrary and serves no other purpose. The provider sets the rest of the entries to their respective function pointers. For example, the provider assigns its *NSPSetService* function address (no matter what it is named) to the *NSPSetService* field. The names of your provider functions can be arbitrary, but their parameters and return types must match the provider definition.

The only action required of an *NSPStartup* implementation is filling in the *NSP_ROUTINE* structure. Once the provider completes this and any other initialization routines of its own, it returns *NO_ERROR* if everything is successful. If an error occurs, the *NSPStartup* implementation returns *SOCKET_ERROR* and sets the Winsock error code. For example, if a provider attempts and fails to allocate memory, it calls *WSASetLastError* with *WSAENOBUFS* as the parameter and then returns *SOCKET_ERROR*.

This might be a good time to discuss error handling in your provider's DLL. All of the functions you must implement for a provider return *NO_ERROR* upon success and *SOCKET_ERROR* upon failure. If you determine that the call fails, set the appropriate Winsock error code before returning. If you fail to do this, any application attempting to register or query services using your name space provider will report the failure of an RNR function but *WSAGetLastError* will return 0. This will cause tremendous trouble for applications that attempt to handle errors gracefully; 0 is certainly not an expected return value upon error.

NSPCleanup

This routine is called when the provider's DLL is unloaded. Within this function, you can free any memory allocated in the *NSPStartup* routine. This routine is defined as

```
int NSPCleanup ( LPGUID lpProviderId );
```

The only parameter is your name space provider's GUID. Other than cleaning up any dynamically allocated memory, you're not required to do anything in this function.

NSPInstallServiceClass

The *NSPInstallServiceClass* function maps to *WSAInstallServiceClass* and is responsible for registering a service class. *NSPInstallServiceClass* is defined as

```
int NSPInstallServiceClass (
    LPGUID lpProviderId,
    LPWSASERVICECLASSINFOW lpServiceClassInfo
);
```

The first parameter is the provider's GUID. The *lpServiceClassInfo* parameter is the *WSASERVICECLASSINFOW* structure that is being registered. Your provider has to maintain a list of service classes and has to ensure that a service class doesn't already exist using the same GUID within the *WSASERVICECLASSINFOW* structure. If the GUID is already in use, the provider must return the error *WSAEALREADY*. Otherwise, the provider should maintain this service class so that other RNR operations can refer to it.

The majority of the remaining name space provider functions refer to an installed service class.

NSPRemoveServiceClass

This function is the complement of the *NSPInstallServiceClass* function and removes the specified service class. This name space function maps to *WSARemoveServiceClass*. The function is declared as

```
int NSPRemoveServiceClass (
    LPGUID lpProviderId,
    LPGUID lpServiceClassId
);
```

As in the previous function, the first parameter is the provider's GUID. The second parameter, *lpServiceClassId*, is the service class GUID that is to be removed. The provider must remove the given service class from its storage. If the service class specified by *lpServiceClassId* is not found, the provider must generate the error *WSATYPE_NOT_FOUND*.

NSPGetServiceClassInfo

The *NSPGetServiceClassInfo* function maps to the *WSAGetServiceClassInfo* function. It retrieves the *WSANAMESPACE_INFOW* structure associated with a GUID. The function is defined as

```
int NSPGetServiceClassInfo (
    LPGUID lpProviderId,
    LPDWORD lpdwBufSize,
    LPWSASERVICECLASSINFOW lpServiceClassInfo
);
```

Again, the first parameter is the provider's GUID. The *lpdwBufSize* parameter indicates the number of bytes contained in the third parameter, *lpServiceClassInfo*. On input, the third parameter is a *WSASERVICECLASSINFOW* structure that contains the search criteria specifying which service class to return. This structure can contain either a service class name or the GUID for the service class to return. If the provider finds a match, it must return the *WSASERVICECLASSINFOW* structure in *lpServiceClassInfo* and should update *lpdwBufSize* to indicate the number of bytes being returned.

If, given the criteria, no service classes are found, the call should fail and set *WSATYPE_NOT_FOUND* as the error. In addition, if a service class does match but the supplied buffer is too small, the provider should update the *lpdwBufSize* parameter to indicate the correct number of bytes required and the error *WSAEFAULT* should be set.

NSPSetService

The *NSPSetService* function maps to *WSASetService* and either registers or removes services from the name space. The function is defined as

```
int NSPSetService (
    LPGUID lpProviderId,
    LPWSASERVICECLASSINFOW lpServiceClassInfo,
    LPWSAQUERYSETW lpqsRegInfo,
    WSAESETSERVICEOP essOperation,
    DWORD dwControlFlags
);
```

The first parameter is the provider's GUID. The *lpServiceClassInfo* parameter is the *WSASERVICECLASSINFOW* structure to which this service belongs. The *lpqsRegInfo* parameter is the service to either register or delete depending on the operation specified in the fourth parameter, *essOperation*. The last parameter, *dwControlFlags*, might specify the flag *SERVICE_MULTIPLE* that can modify the specified operation. See Tables 8-4 and 8-5 in Chapter 8 for a description of the possible *essOperation* and *dwControlFlags* values.

The name space provider first verifies that the supplied service class does exist. Then, depending on which operation is specified, appropriate action is taken. For a full description of valid *essOperation* values as well as the effect of *dwControlFlags* on them, see the section on service registration in

Chapter 8, which discusses *WSASetService*. Your provider's *NSPSetService* function handles these flags accordingly.

If your service provider is updating or deleting a service that cannot be found, the error *WSASERVICE_NOT_FOUND* is set. If the provider is registering a service and the *WSAQUERYSETW* structure is invalid or incomplete, the provider generates the *WSAEINVAL* error.

This function is one of the most complicated name space provider functions to implement (next to *NSPLookupServiceNext*). The provider must maintain a scheme for persisting the services that can be registered and must allow the *NSPSetService* function to update this data.

NSPLookupServiceBegin

The *NSPLookupServiceBegin* function is associated with the *NSPLookupServiceNext* and *NSPLookupServiceEnd* functions and is used to initiate a query of your name space. This function maps to *WSALookupServiceBegin* and establishes the criteria for your search. This function is prototyped as

```
int NSPLookupServiceBegin (
    LPGUID lpProviderId,
        (continued)     LPWSAQUERYSETW lpqsRestrictions,
    LPWSASERVICECLASSINFOW lpServiceClassInfo,
    DWORD dwControlFlags,
    LPHANDLE lphLookup
);
```

As with previous functions in this section, the first parameter is the provider's GUID. The *lpqsRestrictions* parameter is the *WSAQUERYSETW* structure that defines the parameters for the query. The third parameter, *lpServiceClassInfo*, is the *WSASERVICECLASSINFOW* structure containing the schema information for the service class in which the query is to take place. The *dwControlFlags* parameter takes zero or more flags that affect how the query is performed. Again, for information on *WSALookupServiceBegin* and the different flags that can be used, refer to Chapter 8. Note that not all of the flags make sense for every provider. For example, if your name space does not support the notion of container objects, you don't have to worry about those flags dealing with containers. (A container is simply a way of conceptually organizing the services—what constitutes a container is open to interpretation.) Finally, *lphLookup* is an output parameter, which is a handle that defines this particular query. The handle is used in the subsequent calls to *WSALookupServiceNext* and *WSALookupServiceEnd*.

When implementing *NSPLookupServiceBegin*, keep in mind that the operation cannot be canceled, and it should complete as quickly as possible. Therefore, if you need to initiate a network query, a response should not be required to return successfully.

The provider should save the query parameters and associate a unique handle with the query for later reference. In addition to saving the handle and the query, the provider should maintain state information. We'll explore the significance of this in our discussion of the next function, *NSPLookupServiceNext*.

NSPLookupServiceNext

Once a query has been initiated with *WSALookupServiceBegin*, an application calls *WSALookupServiceNext*, which in turn calls the name space provider function *NSPLookupServiceNext*. This call is what actually searches for results that match the search criteria registered for this query. The function is defined as

```
int NSPAPI WSALookupServiceNext (
    HANDLE hLookup,
    DWORD dwControlFlags,
    LPDWORD lpdwBufferLength,
    LPWSAQUERYSET lpqsResults
);
```

The first parameter, *hLookup*, is the query handle returned from *WSALookupServiceBegin*. The *dwControlFlags* parameter can be the flag *LUP_ FLUSHPREVIOUS*, which indicates that the provider should discard the last result set and move to the next one. Typically, an application requests that the last result set be discarded when the application cannot supply a large enough buffer for the results. The next parameter, *lpdwBufferLength*, indicates the size of the buffer passed as the last parameter, *lpqsResults*.

When *NSPLookupServiceNext* is triggered, the provider should look up the query parameters identified by the handle *hLookup*. Once the query parameters are retrieved, a search should be initiated for all registered services within the service class specified by the query that match the supplied criteria. As we mentioned in the section on *NSPLookupServiceBegin*, the state of the query should be saved. If there are multiple matching entries, a calling process calls *WSALookupServiceNext* multiple times, and with each call your provider needs to return a data set. When there are no more matches, the provider returns the error *WSA_E_NO_MORE*. It is also possible to cancel a query in progress if the application makes a call to *WSALookupServiceEnd* from another thread while a call to *WSALookupServiceNext* is in progress. In this event, *NSPLookupService-Next* should fail with the error *WSA_E_CANCELLED*.

NSPLookupServiceEnd

After a query has been completed, *NSPLookupServiceEnd* is called to end the query and release any underlying resources. This function is defined as

```
int NSPLookupServiceEnd ( HANDLE hLookup );
```

The single parameter to the function is *hLookup*, which is the handle to the query that is to be closed. If the given handle cannot be found (for example, if it's an invalid handle), the call must fail with the error *WSA_INVALID_HANDLE*.

NSPIoctl

The last function is *NSPIoctl*, which is not required to develop a name space provider. If a NSP decides not to implement this function, the *cbSize* field of the *NSP_ROUTINE* should be set to the size of the structure without the *LPNSPIOCTL* pointer. This can be done with the following code:

```
lpnspRoutine->cbSize = FIELD_OFFSET(NSP_ROUTINE, NSPIoctl));
```

As we saw in Chapter 8, the *WSANSPIoctl* API is new to Windows XP and is currently used only by the NLA name space, which provides notification when information about the current network changes.

The function definition for the NSP equivalent *NSPIoctl* is

```
INT NSPIoctl(
    HANDLE          hLookup,
    DWORD           dwControlCode,
    LPVOID          lpvInBuffer,
    DWORD           cbInBuffer,
    LPVOID          lpvOutBuffer,
    DWORD           cbOutBuffer,
    LPDWORD         lpcbBytesReturned,
    LPWSACOMPLETION lpCompletion,
    LPWSATHREADID   lpThreadId
    );
```

This function provides a method for exposing miscellaneous commands from the name space, and it is completely up to the name space developer to determine how the input and output parameters work. For example, if an NSP wanted to expose some statistics, such as number of entities registered or number of queries performed, it could do this by implementing *NSPIoctl* and allowing applications to query it via *WSANSPIoctl* with its own defined ioctl code and output buffer structure.

The other benefit to this function is that it allows asynchronous notification through the *lpCompletion* parameter. This structure allows the calling application to specify an overlapped structure, completion routine, window, or completion port to receive notification of completion. Again, it's up to the NSP developer to determine what information is being registered for notification, but this function could allow asynchronous name resolution—something that cannot be done via the *WSALookupService* functions. In Chapter 8, you saw how an application can register to receive notifications when the local network information changes.

If a name space chooses to implement this function, it must save the input and output buffers as well as the *WSACOMPLETION* information. Then when the ioctl completes (either immediately or at some later point), the output information should be copied to the supplied buffer (or an error returned if it's not large enough), and the notification routine (if supplied) should be signaled.

Depending on how the service is implemented, there are several ways to process these asynchronous completion events. If the NSP is notifying the application from the DLL, it is simple. To send a message to a window, use *PostMessage*. To queue an APC, *QueueUserApc* is used. To signal an event, *SetEvent* is called. Lastly, to notify a completion port, *PostQueuedCompletionStatus* is used. These functions are regular Windows APIs and more information about them can be found in the Platform SDK.

The situation becomes more difficult if the application is notified from a service or separate process that persists the name space data. Services often operate under a different user group and may not have access to certain resources. Writing Windows services is beyond the scope of this book. For more information, consult the Platform SDK or ***Programming Server-Side Applications for Windows 2000*** by Jeffrey Richter and Jason D. Clark (Microsoft Press, 2000). In the next section, we'll discuss the sample NSP, which is implemented as a separate process but does not expose the *NSPIoctl* function.

Name Space Provider Example

In the previous sections, we covered the steps for creating your own name space and touched on some of the important name space creation issues, such as methods for data persistence. However, developing an entire name space provider can be complicated, and the rest of this chapter will be devoted to our example name space provider. Although the example is not the fastest or most optimized code, it illustrates the topics that require the most attention. In addition, we kept it simple so it's easy to follow and understand.

The example provider is located on the companion CD in the NSP directory in the files MYNSP.H, MYNSP.CPP, and MYNSP.DEF. These three files make up the name space DLL. In addition to the DLL, you'll find the name space service that is a Winsock server responsible for handling requests from the DLL. This server, which maintains the service registration data, is found in the file MYNSPSVC.CPP. Two additional files, NSPSVC.CPP and PRINTOBJ.CPP, are used by both the DLL and the service and contain support routines for marshaling and demarshaling data sent on a socket between the service and the DLL. Marshaling and demarshaling data will be explained later in this section. In addition to these two files, you'll find their accompanying header files, NSPSVC.H and PRINTOBJ.H, which contain the function prototypes for the support routines. Finally, the file RNRCS.C is a modified sample from Chapter 8 that registers and looks up services in our custom name space. Note that when we refer to the name space provider as a service, we are not implying that the sample code is a true Windows service.

In the following sections, we will discuss how our name space is implemented. First, we'll give an overview of the method we chose to persist the data. This overview will be followed by an examination of how the actual name space DLL is structured as well as how to install the name space. Then we'll cover the implementation of the name space service. Finally, we'll look at how an application performs service registrations and queries to our custom name space.

Data Persistence

For our name space, we chose a separate Winsock application to maintain the name space information. In each of the name space functions implemented in the DLL, a connection is made to this process and data is transacted to complete the operation. For simplicity, this process runs locally (the service listens on the loopback address 127.0.0.1). In an actual implementation, our name space service's IP address would be accessible via the Registry or some other means so that when an application invoked the name space, it could connect to the service wherever it was running. For example, with DNS, the IP address of the DNS server is either set statically or obtained during a DHCP request.

Of course, writing a service to maintain the information is not the only option available. You could maintain a file on the network that keeps the necessary information; however, this is probably not the best option because performance is then bound by disk operations. One performance limitation of our sample name space is that it establishes TCP connections to the service. A production-quality implementation is more likely to use a connectionless datagram protocol such as UDP to improve performance. Of course, this would involve additional programming requirements—such as ensuring that dropped packets are retransmitted—but the overall performance gains would be considerable.

Name Space DLL

Before we look at how the name space service is implemented, let's take a look at the name space DLL. Each name space provider requires a unique GUID, and ours is defined in MYNSP.H. In addition to the unique identifier, we need a simple integer identifier for our name space. This identifier can be used in the *dwNameSpace* field of the *WSAQUERYSET* structure, as you saw in Chapter 8. The GUID and name space identifier are

```
GUID MY_NAMESPACE_GUID = {0x55a2bd9e,0xbb30,0x11d2,
                          {0x91,0x66,0x00,0xa0,0xc9,0xa7,0x86,0xe8}
                         };
#define NS_MYNSP                                           66
```

These values are important because applications that want to use this name space must specify these values in their Winsock calls. Of course, an application's developer can specify these values directly or retrieve them with a *WSAEnumNameSpaceProviders* call. (See Chapter 8 for more information.) Also, be aware that if an application performs an operation specifying the *NS_ALL* name provider, the operation occurs on all installed name providers. You should keep this in mind because several Windows applications, such as Internet Explorer, perform queries on all installed name providers. Be very careful, therefore, when testing a name provider. A poorly written name provider can cause system-wide problems. In addition, the GUID and name space identifier values are important because they are required to install the name provider.

Now let's take a look at the *NSP* functions implemented in MYNSP.CPP. For the most part, these functions are quite similar except for the startup and cleanup functions, *NSPStartup* and *NSPCleanup*. The startup function simply initializes the *NSP_ROUTINE* structure with our custom name space functions. The cleanup routine does nothing because no cleanup is necessary.

The rest of the functions require interaction with our service to either query or register data. When communication with the service is necessary, follow these steps:

1. Connect to the service (via the *MyNspConnect* function).
2. Write a 1-byte action code. This indicates to the service which action is about to take place (such as service registration, service deletion, and query).

3. Marshal parameters and send them to the service. The type of parameters sent will depend on the operation. For example, *NSPLookupServiceNext* sends the query handle to the service so that it can resume the query, whereas *NSPSetService* sends an entire *WSAQUERYSET* structure.
4. Read the return code. Once the service has the necessary parameters to perform the requested operation, the return code (success or failure) of the operation is returned. The file MYNSP.H defines two constants for this purpose: *MYNSP_SUCCESS* and *MYNSP_ERROR*.
5. If the requested operation was a query and the return code was success, read and demarshal the results. For example, *NSPLookup- ServiceNext* returns a *WSAQUERYSET* structure if a matching service is found.

As you can see, implementing the DLL is not overly complicated. The *NSP* functions must take the parameters and process them, which in our case is to pass this information to the name space service. After this, it is up to the service to perform the requested operation. However, we have glossed over one difficult operation that must be performed: sending data over a socket. Normally, there aren't any special requirements for sending data, but when sending entire data structures, there are. Most of the name space functions take either a *WSAQUERYSET* or a *WSASERVICECLASSINFO* structure as a parameter. This object must be sent or received on the socket connection to the service. This presents some difficulty because these structures aren't contiguous blocks of memory. They contain pointers to strings and other structures that can be located anywhere in memory, as illustrated in Figure 12-5. You need to take all of these pieces of memory—wherever they are—and copy them into a single buffer one after another. This is known as ***marshaling data***. On the receiving end, this process has to be reversed. The data read needs to be reassembled into the original structure, and the pointers have to be “fixed” so that they point to valid memory locations on the recipient's machine.

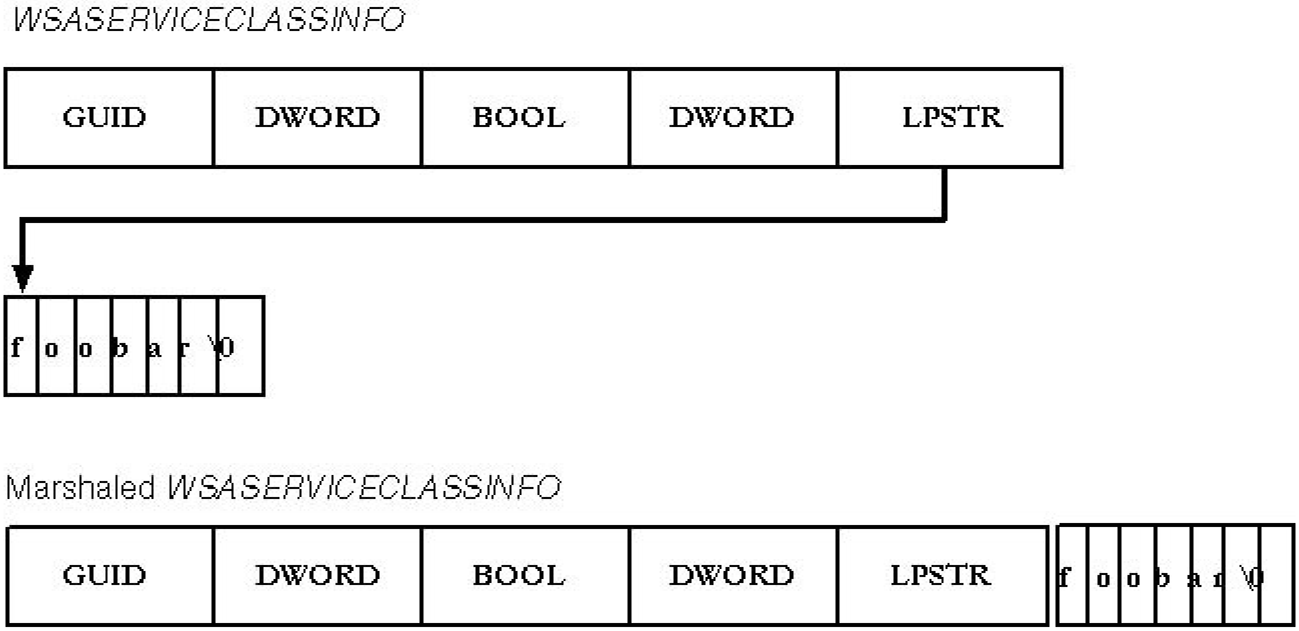


Figure 12-5 *Marshaling data*

For our name space provider, we provide functions to marshal and demarshal both the *WSANAMESPACEINFO* and *WSAQUERYSET* structures. These functions are located in NSPSVC.CPP and are used by both the name space DLL and the name space service (because both sides need the capability to marshal and demarshal these structures). All four functions are self-explanatory—we won't cover them in depth here.

Installing the Name Space

Installing a name space provider is the most trivial step in the entire process. The file NSPINSTALL.C is a simple installation program. The following code installs our provider:

```
ret = WSCInstallNameSpace(L"Custom Name Space Provider",
    L"%SystemRoot%\\System32\\Mynsp.dll", NS_MYNSP, 1,
    &MY_NAMESPACE_GUID);
if (ret == SOCKET_ERROR)
{
    printf("Failed to install name space provider: %d\n",
        WSAGetLastError());
}
```

The only parameters to the call are the provider's name, the DLL's location, the integer identifier, the version, and the GUID. After installation, the only requirement is to make sure that the name space DLL is located where you say it is. The only error that's a real possibility is trying to install a name provider with a GUID that's already in use by another provider.

Removing a name space provider is even easier. The following code snippet from our installation program removes our provider:

```
ret = WSCUnInstallNameSpace(&MY_NAMESPACE_GUID);
if (ret == SOCKET_ERROR)
{
    printf("Failed to remove provider: %d\n", WSAGetLastError());
}
```

Name Space Service

The name space service is the real guts of the name provider. This service keeps track of all registered service classes and service instances. When a user's application triggers the name space DLL, it connects to the name space service to perform the operation. The service is simple. Within the main function, a listening socket is established. Then, within a loop, connections are accepted from instances of the name space DLL. For simplicity, only a single connection is handled at a time. This also prevents you from having to synchronize access to the data structures that maintain the name space information. Again, a real provider would not do this because it degrades performance, but it does make the example easier to understand. Once a connection is accepted, the service reads a single byte from the name space DLL that identifies the action to follow.

Within the loop, the action is decoded and parameters are passed from the name space DLL to the service. From there, the requested actions are performed. These actions aren't complicated. The code is easy to follow, and by examining the steps for each possible action you can determine how the service works—we don't need to go into detail here. However, we will examine the structures that maintain the information. There are only two data types that name space providers are concerned about: the *WSASERVICECLASSINFO* and *WSAQUERYSET* structures. As you have seen, the majority of the RNR functions reference one or the other of these two structures in their parameters. As a result, we maintain two global arrays—one for each structure type—along with a counter for each.

When the DLL requests to install a service class, the name service provider's main function first calls *LookupServiceClass*, a support routine defined in MYNSPSVC.CPP. This function iterates through the array of *WSASERVICECLASSINFO* structures, *ServiceClasses*. If a service class is found with the same GUID, it returns an error (which the DLL translates as *WSAEALREADY*). Otherwise, the new service class is added to the end of the array and the *dwNumServiceClasses* counter is incremented.

During the deletion of a service class (as when installing a service class), the main function calls *LookupServiceClass*. In this case, however, if the service class is found, the code moves the last service class in the array to the location of the deleted class. The code then decrements the counter. One aspect that is not specifically covered in the Winsock 2 specification for name space providers is what happens when a service class is to be deleted but there are still services registered that refer to it. How you choose to handle this is up to you. Our example name space won't allow the removal of a service class if there are services registered that reference it.

The same principle that's used for maintaining *WSASERVICECLASSINFO* structures is also used for keeping track of *WSAQUERYSET* structures. There is an array of these structures named Services, as well as a counter named *dwNumServices*. The addition and deletion of services is handled in the same manner that it is for service classes.

The last bits of information that the service must maintain are for queries. When an application initiates a query, the query parameters must be maintained for the life of the query and assigned a unique handle. This is necessary because *WSALookupServiceNext* refers to the query by the handle only. The other piece of information that must be kept is the state of the query. Each call to *WSALookupServiceNext* can return a single information set. The code must remember the last position within the Services array where data was returned so that subsequent calls to *WSALookupServiceNext* begin where the previous call left off.

Querying the Name Space

The last part of our name space sample is the file RNRCS.C. This is a modified version of the name registration and resolution example presented in Chapter 8. We've made only a few changes to make the example as simple as possible. The first change causes the code to enumerate the installed name space providers but to return only the *NS_MYNSP* provider. Second, when registering a service, RNRCS.C enumerates only the local IP interfaces to use as the address of our service. Our service provider supports the registration of any *SOCKADDR* type. Finally, for service registration, this example does not create an instance of the service; it just registers the name. Otherwise, this example behaves like the Chapter 8 example.

Running the Example

Once all the examples have been compiled, installing and using the provider is simple. The following command installs the provider:

```
Nspinstall.exe install
```

Of course, don't forget to copy MYNSP.DLL to %SystemRoot%\System32. Once the name space is installed, an instance of the service must be running to query and register services. This is done by the following command:

```
Mynspsvc.exe
```

Now you can query and register services using RNRCS.EXE. Table 12-6 shows some commands that you should execute and the order you should follow. This sequence of commands registers two services and then performs a wildcard query and a specific query. Then the command sequence queries for each of the two services and deletes them. Finally, we perform a wildcard query to illustrate that the services have been deleted.

Table 12-6 *Running the Sample Name Space*

Command	Meaning
RNRCS.EXE -s:ASERVICE	Register the service ASERVICE.
RNRCS.EXE -s:BSERVICE	Register the service BSERVICE.
RNRCS.EXE -c:*	Query for all registered services.
RNRCS.EXE -c:BSERVICE	Query only for services named BSERVICE.
RNRCS.EXE -c:ASERVICE -d	Query only for services named ASERVICE, and delete them if found.
RNRCS.EXE -c:BSERVICE -d	Query only for services named BSERVICE, and delete them if found.
RNRCS.EXE -c:*	Query for all registered services.

Conclusion

The Winsock 2 SPI offers software developers a method of extending the capabilities of Winsock 2 by developing a service provider. In this chapter, we explained the details of how to develop an LSP and a name space provider. This chapter concludes our discussion of the Winsock networking technology. The next few chapters discuss accessing Winsock functionality from two other programming languages: Visual Basic and C#.

Chapter 13

.NET Sockets Programming Using C#

Microsoft has introduced a new programming interface called .NET Application Frameworks version 1, which is a large collection of object-oriented classes that allows applications to work better in a highly distributed environment such as the Internet. .NET Frameworks provides a myriad amount of functionality from drawing graphics on a screen to processing Extensible Markup Language (XML) data. This chapter primarily focuses on one of the .NET Framework classes, .NET Sockets, which allows a .NET application to take advantage of the Winsock programming interface in a new object oriented fashion. The .NET Sockets programming interface is actually known as the *System.Net.Sockets* namespace that is included in the new .NET Framework SDK. The .NET Frameworks version 1 can be installed on Windows 98, Windows Me, Windows NT 4.0, Windows 2000, and Windows XP.

The .NET Framework classes can be accessed from Visual Basic, Visual C++, and a new language named C# (pronounced C sharp). This chapter is intended to introduce you to the new .NET Sockets namespace, providing basic information about how to use the methods and properties available in the library to perform Winsock communication. Our discussion will demonstrate .NET Sockets using primarily the IPv4 protocol to perform simple TCP and UDP communication over a socket. We also will describe IPv6 in a limited fashion. The information is presented using the C# programming language. This chapter does not detail all of the methods available to make every Winsock call, functionality, or protocol and assumes you have familiarity with C# and Winsock programming techniques.

Overview

Programming .NET Sockets applications is similar to developing a Winsock application in C/C++ because the interface uses methods that have nearly the same name and parameters as the Winsock API. The basic design centers around creating a Socket object and using methods included in that object. The Socket methods mimic the standard Winsock 1.1 calls that we described previously in this book; internally, however, the calls use Winsock 1 and 2 functionality. Although the methods are similar to the Winsock API, one of the most noticeable programming differences is that you don't have startup Winsock (by calling WSAStartup) to use the Sockets namespace as you do when writing a Winsock application. Instead, once you create a Socket object the startup feature is handled within the interface.

Creating a Socket object is simple: Create a new *System.Net.Sockets.Socket* object. The constructor for this object is defined as

```
System.Net.Sockets.Socket(
    System.Net.Sockets.AddressFamily AddressFamily,
    System.Net.Sockets.SocketType SocketType,
    System.Net.Sockets.ProtocolType ProtocolType
);
```

The socket constructor method takes *System.Net.Sockets* enumeration type parameters *AddressFamily*, *SocketType*, and *ProtocolType* to create the socket. These three enumerations types have values that are similar to the Winsock 1 *socket* parameter types. Table 13-1 contains all of the enumerated types that all of the objects in the Sockets namespace use. Once you have successfully created a Socket object, you can begin developing a client or server application using the available methods defined in Table 13-2.

Table 13-1 *Useful Enumerations in the .NET Sockets Namespace*

Property	Description
AddressFamily	Similar to Address Families in Winsock—use *InterNetwork* for IPv4 addresses and *InterNetworkV6* for IPv6 addresses.
ProtocolFamily	Identifies which protocol to use in the Winsock catalog.
ProtocolType	Similar to protocol types in Winsock—use *IP* for Internet Protocol.
SelectMode	Used for polling on attributes on a Socket.
SocketFlags	Similar to socket flags in Winsock used for *Send* and *Receive* methods.
SocketOptionLevel	Available socket option levels for *GetSocketOption* and *SetSocketOption* methods.
SocketOptionName	Available socket options for *GetSocketOption* and *SetSocketOption* methods.
SocketShutdown	Similar to Winsock *shutdown* parameters.
SocketType	Similar to socket types in Winsock—use *Stream* for TCP sockets.

Table 13-2 *Available Methods in the .NET Sockets Socket Class*

Method	Description
Accept, BeginAccept, EndAccept	Same as Winsock *Accept*; in addition, there is an asynchronous *BeginAccept* and *EndAccept* pair. We will describe aynchronous .NET Sockets later in the chapter.
Bind	Same as Winsock *bind*.
Close	Same as Winsock *closesocket*.
Connect, BeginConnect, EndConnect	Same as Winsock *connect*; in addition, there is an asynchronous *BeginConnect* and *EndConnect* pair.
GetSocketOption	Similar to Winsock *getsockopt*.
IOControl	Similar to Winsock *ioctlsocket*.
Listen	Same as Winsock *listen*.
Poll	Can be used to determine the status of a socket, such as if data is available to be read.
Receive, BeginReceive, EndReceive	Similar to Winsock *recv*; in addition, there is an asynchronous *BeginReceive* and *EndReceive* pair.
ReceiveFrom, BeginReceiveFrom, EndReceiveFrom	Similar to Winsock *recvfrom*; in addition, there is an asynchronous *BeginReceiveFrom* and *EndReceiveFrom* pair.
Select	Similar to Winsock *select*.
Send, *BeginSend*, *EndSend*	Similar to Winsock *send*; in addition, there is an asynchronous *BeginSend* and *EndSend* pair.
SendTo, *BeginSendTo*, *EndSendTo*	Similar to Winsock *sendto*; in addition, there is an asynchronous *BeginSendTo* and *EndSendTo* pair.
SetSocketOption	Similar to Winsock *setsockopt*.
Shutdown	Similar to Winsock *shutdown*.

The following C# code demonstrates how to create a stream socket to communicate over TCP using IPv4:

```
using System;
namespace MySocketApplication
{
 class MySocketClass
 {
 static void Main(string[] args)
  {
      System.Net.Sockets.Socket MySocket = new System.Net.Sockets.Socket(
          System.Net.Sockets.AddressFamily.InterNetwork,
          System.Net.Sockets.SocketType.Stream,
          System.Net.Sockets.ProtocolType.IP
      );
  }
 }
}
```

Addressing Protocols

Once you have created a *Socket* you have to address (or name) one to start communication over a protocol. Addressing protocols in .NET Sockets is done through the *System.Net.Endpoint* class. The *Endpoint* class is derived from a *System.Net.SocketAddress* class, which is similar to a Winsock *sockaddr* structure because all protocol address families can be addressed from its base class through inheritance. In .NET Frameworks version 1, there is currently only one available derived address *Endpoint* class that will support IPv4 addressing: *System.Net.IPEndPoint*. Because of this, in the chapter samples we primarily use IPv4; however, we also have derived a new *Endpoint* class to support IPv6, called *IPv6EndPoint*. In future versions of the .NET Application Frameworks there will eventually be more built-in support to address other protocols, such as IPv6. If you understand the Winsock *sockaddr* structure, you can technically address other protocols such as IPX by creating your own derived *EndPoint* class.

Because IPv4 support is built-in, we will describe *IPEndPoint*. There are two *IPEndPoint* constructor functions that are defined as

```
public IPEndPoint(
   IPAddress address,
   int port
);
public IPEndPoint(
   long address,
   int port
);
```

The first constructor accepts an IP address as a *System.Net.IPAddress* class that is used to manage an IPv4 address. The other constructor takes an IP address as a numeric value. Both functions accept a port as a numeric value. The *System.Net.IPAddress* class provides convenient methods (similar to Winsock 1) that are used to manage and manipulate an IPv4 address. The constructor of this class is defined as

```
public IPAddress(
   long newAddress
);
```

The constructor can take an IPv4 address as a numeric value, however, the real benefit of this class is the methods available to construct an IP address, as described in Table 13-3.

Table 13-3 *Convenient IP Address Methods for IPv4 Address Manipulation*

Method	Description
HostToNetworkOrder	Similar to Winsock *htonl*
IsLoopback	Determines if an IP address is a loopback address
NetworkToHostOrder	Similar to Winsock *ntohl*
Parse	Converts an IP address represented as a string to an *IPAddress* object instance

Equals	Allow comparing two *IPAddresses*
ToString	Converts an IP address to the string format x.x.x.x.

The following code fragment demonstrates how to construct an *IPEndPoint* using the *IPAddress* class.

```
System.Net.IPAddress MyAddress =
    System.Net.IPAddress.Parse("136.149.3.29");
System.Net.IPEndPoint MyEndPoint = new
    System.Net.IPEndPoint(MyAddress, 5150);
```

At this point, you can use the *MyEndPoint in the Bind*, *Connect*, *SendTo*, and *ReceiveFrom* methods of *System.Net.Sockets.Socket* class to address an IPv4 socket.

Name Resolution

As you saw in Chapter 1, Winsock provides a convenient function, *gethostbyname*, which allows you to resolve a hostname to an IPv4 address using name resolution techniques in Windows such as DNS. The .NET Application Frameworks version 1 provides the *System.Net.DNS* class to support domain name resolution functionality for IPv4 addresses. Table 13-4 describes the available methods in this class.

Table 13-4 *Available Methods for the System.Net.DNS Class*

Method	Winsock Equivalent
GetHostByAddress	*gethostbyaddr*
GetHostByName, *BeginGetHostByName*, *EndGetHostByName*	*gethostbyname*
GetHostName	*gethostname*
Resolve, *BeginResolve*, *EndResolve*	*gethostbyname*

The *System.Net.DNS* class relies on the *System.Net.IPHostEntry* class to store IPv4 addresses that a DNS query returns. The *DNS.GetHostByName* method will return a list of host IPv4 addresses to an *IPHostEntry* container. This method can take a string representing both a name and even an IPv4 address in dot notation. The following code fragment demonstrates how to use DNS to resolve IPv4 addresses. It also demonstrates how you can set up an *IPEndpoint* to name a socket from the information we have described so far.

```
// Resolve a host name to an IPv4 address
System.Net.IPHostEntry IPHost =
    System.Net.Dns.GetHostByName("www.Microsoft.com");

// Set up an IPEndPoint using the first address in our IPHostEntry list
System.Net.IPEndPoint ServerEndPoint = new
    System.Net.IPEndPoint(IPHost.AddressList[0], Port);
```

IPv6 DNS name resolution is not available in the .NET Frameworks version 1. Therefore, it is not yet possible to develop agnostic C# socket applications (as describe in Chapter 3) to transparently handle both IPv4 and IPv6 name resolution. IPv6 DNS name resolution is expected in the next version of the Frameworks.

Sending and Receiving Data

Sending and receiving data in .NET sockets is really simple. Once you have created a *Socket* object, you can use the *Send*, *SendTo*, *Receive*, and *ReceiveFrom* methods, which are similar to the *send*, *sendto*, *recv*, and *recvfrom* Winsock 1 APIs. There are several overloaded versions of these send and receive methods. Each one sends and receives data using a simple byte type array.

I/O Methods

.NET Sockets has three basic I/O methods to manage data and connections on a socket: *blocking*, *select*, and *asynchronous*. These resemble some of the I/O methods described in Chapter 5.

Blocking I/O

Blocking I/O is the simplest model to use. Anytime you call an I/O-bound .NET Sockets method, such as *Receive*, when there is no data pending on the receiving socket, the call will do just that—block. If you need your application to do other things or to service additional socket requests, you will have to create additional threads in your application. If your application is just a simple client handling one connection, blocking sockets is a good I/O model to use. The following code fragment demonstrates how to develop a simple client application that can connect to a server and send a simple string using blocking I/O on the *Connect* and *Send* calls. This sample can be found on the companion CD in the TCPClient directory.

```
System.Net.IPAddress ServerAddress =
    System.Net.IPAddress.Parse("136.149.3.29");
System.Net.IPEndPoint ServerEndPoint = new
    System.Net.IPEndPoint(ServerAddress, 5150);

System.Net.Sockets.Socket MySocket = new Socket(
        AddressFamily.InterNetwork,
        SocketType.Stream,
        ProtocolType.IP);

MySocket.Connect(ServerEndPoint);

String s = "Hello - This is a test";

Byte[] buf = System.Text.Encoding.ASCII.GetBytes(s.ToCharArray());

int BytesSent = MySocket.Send(buf);

System.Console.WriteLine("Successfully sent " +
    BytesSent.ToString() + " byte(s)");

MySocket.Shutdown(System.Net.Sockets.SocketShutdown.Both);
MySocket.Close();
```

If you plan to develop an application that manages multiple sockets, we suggest using one of our next two models, *Select* or *Asynchronous*, instead of creating multiple threads.

Select I/O

With the limitations of blocking I/O for managing multiple sockets, .NET Sockets features a *Select* method that is similar to the *Select* Winsock 1 API that allows managing multiple socket I/O from one execution thread. Essentially, you can provide *Select* with a list of sockets to test for readability, writeabilty, and OOB data. The following code fragment demonstrates how to test sockets for readability:

```
// Assume we have 2 connected sockets - Socket1 and Socket2
Socket[] ReadList = new Socket[2];
ReadList[0] = Socket1;
ReadList[1] = Socket2;

Socket.Select(ReadList, null, null, 100000);

// When Select returns either by our timeout
// value 100000 or if data is pending on one
// of our sockets, the ReadList will contain
// only those sockets that need to be read.
for (int i = 0; i < ReadList.Length; i++)
{
        byte [] Buffer = new byte[1024];

        // Receive data from the returned socket;
        ReadList[i].Receive(Buffer);
}
```

Although *Select* can manage multiple sockets from a single thread, we highly recommend using our next model—asynchronous—especially if you are developing a high-performance server. For more information about using the Winsock select API, see Chapter 5.

Asynchronous I/O

The asynchronous model in .NET sockets is the best way to manage I/O from one or more sockets. It is the most efficient model of the three because its design is similar to the I/O completion ports model (described in Chapter 5) and on Windows NT–based systems it will use the completion port I/O model internally. Because of this, you can potentially develop a high-performance, scalable Winsock server in C# and even possibly in Visual Basic. For a complete discussion of how to develop a high-performance, scalable Winsock application, see Chapter 6.

In Table 13-2 we described several methods that may be used to process I/O asynchronously: *BeginAccept*, *EndAccept*, *BeginConnect*, *EndConnect*, *BeginReceive*, *EndReceive*, *BeginSend*, *EndSend*, *BeginSendTo*, and *EndSendTo*. Notice how each one of these methods has a "BeginXXX"-"EndXXX" pair for each of the major I/O-bound socket methods—*Accept*, *Connect*, *Receive*, *Send*, and *SendTo*.

To call one of the I/O socket methods asynchronously, you must call the “BeginXXX” method counterpart and supply a delegate (or callback) method in the “BeginXXX” call. When the “BeginXXX” call completes, your calling thread may continue processing other things while your supplied delegate method internally waits for I/O to complete. When the socket has completed your I/O operation, your delegate method is called to process the completed I/O results. Inside your delegate method, you can retrieve the completed I/O results using the EndXXX counterpart method.

For example, let's describe how to process a *Receive* call asynchronously. We chose *Receive* because it is one of the most common methods that can cause your application to block when you wait for data to arrive on a socket. To call *Receive* asynchronously, you must call *BeginReceive*, which is defined as

```
public IAsyncResult BeginReceive(
   byte[] buffer,
   int offset,
   int size,
   SocketFlags socketFlags,
   AsyncCallback callback,
   object state
);
```

Most of the parameters are similar to the Winsock *recv* API except for the *callback* and *state* parameters. The *callback* parameter accepts a delegate method that is used to handle the completed results of the asynchronous *BeginReceive*. The delegate method must have the following form:

```
public delegate void AsyncCallback(
   IAsyncResult ar
);
```

The *ar* parameter is an input parameter that receives an *IAsyncResult* object, which you can pass to the *EndReceive* counterpart method (alternatively, you can use the *IAsyncResult* object that is returned from the originating *BeginReceive* call). Also, *IAsyncResult* contains an important member variable, *AsyncState*, which contains per-I/O data that was originally passed in the *state* parameter of the originating *BeginReceive* call. Typically, you will use this per-I/O data object to pass buffer and socket information that is related to the receive call.

Once your delegate method is called after *BeginReceive* has completed, you should call *EndReceive* to retrieve the results of the asynchronous *Receive*, which is defined as

```
public int EndReceive(
   IAsyncResult asyncResult
);
```

EndReceive returns the number of bytes received in the buffer that was originally passed to *BeginReceive*. Once you have the completed results, you can begin processing the data received in the buffer.

> When you call *BeginReceive*, *BeginReceiveFrom*, *BeginSend*, or *BeginSendTo*, you are not allowed to access the supplied buffer until your delegate method has been called indicating that the asynchronous method has completed.

The following code fragment demonstrates how to call *Receive* asynchronously using *BeginReceive* and *EndReceive*. On the companion CD we have provided a sample called TCPServer that demonstrates how to call *Accept*, *Receive*, and *Send* asynchronously on a TCP socket.

```
// Assume we have a connected socket named MySocket

PerIOData PData;
PData.s = MySocket;

public AsyncCallback AsyncReceiveCallback = new
AsyncCallback(ProcessReceiveResults);

MySocket.BeginReceive(PData.Buffer, 0, PData.Buffer.Length,
        SocketFlags.None, AsyncReceiveCallback, PData);



public static void ProcessReceiveResults(IAsyncResult ar)
{
        PerIOData PData = (PerIOData) ar.AsyncState;

        int BytesReceived = PData.s.EndReceive(ar);

   // Do something about your received results
   . . .
}

public class PerIOData
{
   // Put whatever data you need here for the delegate method.
   // Most applications will probably define the data buffer
   // here for the received data.
   byte [] Buffer = new byte[4096];
   Socket s;
   . . .
}
```

Exception Handling

By now you are probably wondering how to handle socket errors. In traditional Winsock, handling errors requires checking the return code of every Winsock call that is made in your application. In .NET Sockets using C#, this is not quite true. C# is designed to handle errors through exception handling in which you can wrap one or more .NET Sockets calls in a *try-catch* block of code. Most of the .NET Sockets calls raise a *System.Net.Sockets.SocketException* when a socket error occurs. When the exception is raised, your catch block can receive *System.Net.Sockets.SocketException* object. The *SocketException* object has two important properties: *ErrorCode* and *Message*. The *ErrorCode* property contains the actual Winsock error code that occurred on the socket. The *Message* property is an error message related to the Winsock error code. The following code fragment demonstrates how to handle a *SocketException* error from the *Receive* method.

```
try
{
        MySocket.Receive(Buffer);
}
catch (SocketException err)
{
        Console.WriteLine("The Winsock error code is " + err.ErrorCode);
        Console.WriteLine("The related error message is " + err.Message);
}
```

Another exception worth mentioning is that you can also receive a *System. ObjectDisposedException* if your application closes a socket and you continue to perform methods in the Socket class, such as receiving data on a socket. This can happen when you post several asynchronous calls, such as *BeginReceive*, and you close your socket for whatever reason. When your outstanding *BeginReceive* calls complete because of socket closure, they typically call *EndReceive* to retrieve results. Because the socket is closed, *EndReceive* will raise an *ObjectDisposedException* exception.

There are more exceptions that can be handled on each of the .NET Sockets methods. We highly recommend reviewing the .NET Application Frameworks SDK for more information.

Samples

Four samples are available on the companion CD: TCPCLIENT, TCPSERVER, UDPSENDER, and UDPRECEIVER. TCPCLIENT, UDPSENDER, and UDPRECEIVER are simple applications that handle TCP and UDP sockets using simple blocking calls. The TCPSERVER application handles numerous TCP socket connections demonstrating asynchronous IO methods. The TCP samples support both the IPv4 and IPv6 protocols and the UDP samples support the IPv4 protocol only.

As we mentioned earlier, .NET Frameworks version 1 does not natively supply an *EndPoint* class to support IPv6. Each TCP sample includes new IPv6EndPoint and IPv6Address classes that support IPv6 addressing. At some point, a future version of .NET Application Frameworks may have native classes that are similar to the one we provide.

Conclusion

.NET Sockets offers an exciting new way to develop a Winsock application in a new managed code environment of the .NET Application Frameworks. One of its biggest benefits is that you can develop one simple network application that will run on many platforms. Another strong point is that you can use the available methods in the same manner regardless of the programming language used. This chapter introduced you to the basics of network programming in .NET Application Frameworks using C#. You can also easily use the available .NET Sockets methods in Visual C++ and Visual Basic. Chapter 14 describes Winsock programming using Visual Basic.

Chapter 14

The Microsoft Visual Basic Winsock Control

This chapter describes the Visual Basic Winsock control, whose purpose is to simplify the Winsock interface into an easy-to-use interface natively available from Visual Basic. Before the control was available, the only option for Winsock network programming from Visual Basic was to import all of the Winsock functions from the DLL and redefine the many necessary structures. This process was extremely time-consuming and prone to numerous errors, such as mismatching the type declarations. However, if you need the extra flexibility offered by directly importing Winsock into Visual Basic, take a look at the Visual Basic examples that are available in Chapter 1. Each example contains a file, WINSOCK.BAS, which imports the necessary constants and functions. With the release of .NET Application Frameworks version 1 and Visual Studio .NET, there is a new, much more flexible Winsock interface called .NET Sockets, which we described in Chapter 13. It is capable of supporting most of the Winsock functionality from a Visual Basic application in an efficient managed environment. We highly recommend that Visual Basic applications start using the new .NET Sockets interface for all Winsock programming tasks. However, this chapter focuses only on the Visual Basic Winsock control. We'll first cover the properties and methods of the control and then present several examples that use it.

The first Winsock control was introduced with Visual Basic 5. A revised version of the control became available with the release of Visual Studio Service Pack 2. Visual Basic 6 includes the latest version of the Winsock control. The various version differences are discussed toward the end of this chapter.

The Winsock control provides only a basic interface to the Winsock APIs. Unlike Winsock, which is a protocol-independent interface, the Winsock control can use only the IPv4 transport. In addition, it is based on the Winsock 1.1 specification. The control supports both TCP and UDP, but in a rather limited sense. The control itself is not able to access any socket options, which means that features such as multicasting and broadcasting aren't available. Basically, the Winsock control is useful only if you require basic data networking capabilities. It does not provide the best performance because it buffers data within the control before it passes it to the system, thus adding a bit of overhead and uncertainty.

Properties

Now that you have an idea of what functionality the Winsock control provides, let's look at the properties it exposes. Table 14-1 contains a list of the properties available for affecting the control's behavior and for obtaining information about the control's state.

Table 14-1 *Winsock Control Properties*

Property Name	Return Value	Read-Only?	Description
BytesReceived	*Long*	Yes	Returns the number of bytes pending in the receive buffer. Use the *GetData* method to retrieve the data.
LocalHostName	*String*	Yes	Returns the local machine name.
LocalIP	*String*	Yes	Returns a string of the dotted decimal IP address of the local machine.
LocalPort	*Long*	No	Returns or sets the local port to use. Specifying 0 for the port tells the system to randomly choose an available port. In general, only a client uses 0.
Protocol	*Long*	No	Returns or sets the protocol for the control, which supports either TCP or UDP. The constant values to set are *sckTCPProtocol* and *sckUDPProtocol*, which correspond to 0 and 1, respectively.
RemoteHost	*String*	No	Returns or sets the remote machine name. You can use either the string host name or the dotted decimal string representation.
RemoteHostIP	*String*	Yes	Returns the remote machine's IP address. For TCP connections, this field is set upon a successful connection. For UDP operations, this field is set upon the *DataArrival* event, which then contains the sending machine's IP address.
RemotePort	*Long*	No	Returns or sets the remote port to connect to.
SocketHandle	*Long*	Yes	Returns a value that corresponds to the socket handle.
State	*Integer*	Yes	Returns the state of the control, which is an enumerated type. See Table 14-2 for the socket state constants.

After reading Chapter 1, you should be familiar with these basic properties. They are clearly analogous to the basic Winsock functions presented in the client/server examples discussed in that chapter. A few properties that don't relate well to the Winsock API do need to be set to use the control properly. First, the *Protocol* property needs to be set to tell the control which type of socket you're looking for—*SOCK_STREAM* or *SOCK_DGRAM*. The control performs the actual socket creation under the hood, and this property is the only control you have over it. The *SocketHandle* property can be read after a connection succeeds or after a server binds to wait for connections. This is useful if you want to pass the handle to other Winsock API functions imported from a DLL. The *State* property can be used to get information about what the control is currently doing. This is important because the control is asynchronous, and events can be fired at any time. Use this property to make sure that the socket is in a valid state for any subsequent operations. Table 14-2 contains the possible socket states and their meanings.

Table 14-2 *Socket States*

Constant	Value	Meaning
sckClosed	0	Default. Closed.
sckOpen	1	Open.
sckListening	2	Listening for connections.
sckConnectionPending	3	Connection request has arrived but has not completed yet.
sckResolvingHost	4	Host name is being resolved.
sckHostResolved	5	Host name resolution has completed.
sckConnecting	6	Connection request started but has not completed.
sckConnected	7	Connection completed.
sckClosing	8	Peer has initiated a close.
sckError	9	An error has occurred.

Methods

The Winsock control has only a handful of methods. With a couple of exceptions, most of the method names mirror their Winsock equivalents. The method to read pending data is named *GetData*. Usually, you would call the *GetData* method once the *DataArrival* event is triggered, notifying you that data has arrived. The method for sending data is named *SendData*. In addition, a method named *PeekData* is similar to calling the Winsock *recv* function with the *MSG_PEEK* option. As always, message peeking is evil and should be avoided at all costs. Table 14-3 lists the available methods with their parameters. The methods will be discussed in more detail in the client and server example sections later in this chapter.

Table 14-3 *Winsock Control Methods*

Method	Parameters	Return Value	Description
Accept	*RequestID*	Void	For TCP connections only. Use this method to accept incoming connections when handling a *ConnectionRequest* event.
Bind	*LocalPort* *LocalIP*	Void	Binds the socket to the given local port and IP. Use *Bind* if you have multiple network adapters. *Bind* must be called before *Listen*.
Close	None	Void	Closes the connection or the listening socket.
Connect	*RemoteHost* *RemotePort*	Void	Establishes a TCP connection to the given *RemoteHost* on the given *RemotePort* number.
GetData	*Data* *Type* *MaxLen*	Void	Retrieves the current data pending. The *Type* and *MaxLen* parameters are optional. The *Type* parameter defines the type of data to be read. The *MaxLen* parameter specifies how many bytes or characters to retrieve. *GetData* ignores the *MaxLen* parameter for types other than byte array and string.
Listen	None	Void	Creates a socket and places it in listen mode. *Listen* is used for TCP connections only.
PeekData	*Data* *Type* *MaxLen*	Void	Behaves exactly like *GetData* except that the data is not removed from the system's buffer.
SendData	*Data*	Void	Sends data to the remote computer. If a UNICODE string is passed, it will be converted to an ANSI string first. Always use a byte array for binary data.

Events

Events are asynchronous routines that get called upon a specific event. In your Visual Basic application, you must handle the various events that the Winsock control might generate to use the control successfully. In general, these events are triggered by actions that the peer initiates. For example, the TCP half-close is triggered when one side of a TCP connection closes the socket. The side initiating the close generates a FIN, and the peer responds with an ACK to acknowledge the close request. The peer receiving the FIN has the *Close* event triggered. This tells your Winsock application that the other side is no longer sending data. Your application then reads any remaining data and calls the *Close* method on your end to completely shut down the connection. Table 14-4 lists all possible Winsock events that can be triggered, along with a description of each event.

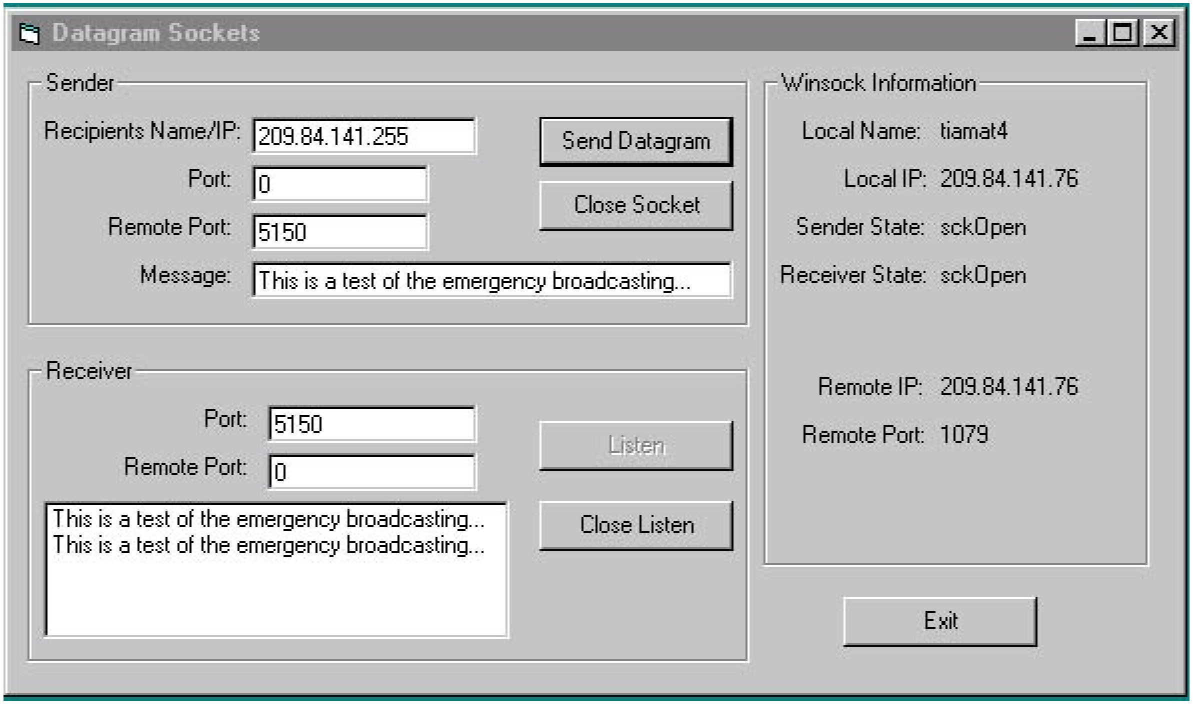
Table 14-4 *Winsock Control Events*

Event	Arguments	Description
Close	None	Occurs when the remote computer closes the connection
Connect	None	Occurs after the *Connect* method completes successfully
ConnectionRequest	*RequestID*	Occurs when a remote machine requests a connection
DataArrival	*bytesTotal*	Occurs when new data arrives
Error	*Number* *Description* *Scode* *Source* *HelpFile* *HelpContext* *CancelDisplay*	Occurs whenever a Winsock error is generated
SendComplete	None	Occurs upon completion of a send operation
SendProgress	*bytesSent* *bytesRemaining*	Occurs while data is being sent

UDP Example

Let's examine a sample UDP application. Look at the sample Visual Basic project SOCKUDP.VBP in the Chapter 14 samples directory on the CD. When the project is compiled and run, you will see a dialog similar to the one illustrated in Figure 14-1.

This sample application both sends and receives UDP messages, so you can use just one instance to send and receive messages. In addition, all the code behind the form, buttons, and Winsock controls is given in the following sample:



Recipients Name/IP: 209.84.141.255

Port: 0

Remote Port: 5150

Message: This is a test of the emergency broadcasting...

Port: 5150

Remote Port: 0

This is a test of the emergency broadcasting...
This is a test of the emergency broadcasting...

Local Name: tiamat4

Local IP: 209.84.141.76

Sender State: sckOpen

Receiver State: sckOpen

Remote IP: 209.84.141.76

Remote Port: 1079

Figure 14-1 *Sample UDP application*

```
Option Explicit

Private Sub cmdExit_Click()
    Unload Me
End Sub

Private Sub cmdSendDgram_Click()
    ' If the socket state is closed, we need to bind to a
    ' local port and also to the remote host's IP address and port
    If (sockSend.State = sckClosed) Then
        sockSend.RemoteHost = txtRecipientIP.Text
        sockSend.RemotePort = CInt(txtSendRemotePort.Text)
        sockSend.Bind CInt(txtSendLocalPort.Text)

        cmdCloseSend.Enabled = True
    End If
    '
    ' Now we can send the data
    '
    sockSend.SendData txtSendData.Text
End Sub
```

```
Private Sub cmdListen_Click()
    ' Bind to the local port
    '
    sockRecv.Bind CInt(txtRecvLocalPort.Text)
    '
    ' Disable this button because it would be an error to bind
    ' twice (a close needs to be done before rebinding occurs)
    '
    cmdListen.Enabled = False
    cmdCloseListen.Enabled = True
End Sub

Private Sub cmdCloseSend_Click()
    ' Close the sending socket, and disable the Close button
    '
    sockSend.Close
    cmdCloseSend.Enabled = False
End Sub

Private Sub cmdCloseListen_Click()
    ' Close the listening socket
    '
    sockRecv.Close
    ' Enable the right buttons
    '
    cmdListen.Enabled = True
    cmdCloseListen.Enabled = False
    lstRecvData.Clear
End Sub
Private Sub Form_Load()
    ' Initialize the socket protocols, and set up some default
    ' labels and values
    '
    sockSend.Protocol = sckUDPProtocol
    sockRecv.Protocol = sckUDPProtocol

    lblHostName.Caption = sockSend.LocalHostName
    lblLocalIP.Caption = sockSend.LocalIP

    cmdCloseListen.Enabled = False
    cmdCloseSend.Enabled = False

    Timer1.Interval = 500
    Timer1.Enabled = True
End Sub

Private Sub sockSend_Error(ByVal Number As Integer, _
        Description As String, ByVal Scode As Long, _
        ByVal Source As String, ByVal HelpFile As String, _
        ByVal HelpContext As Long, CancelDisplay As Boolean)
    MsgBox Description
End Sub

Private Sub sockRecv_DataArrival(ByVal bytesTotal As Long)
    Dim data As String

    ' Allocate a string of sufficient size, and get the data;
    ' then add it to the list box
    data = String(bytesTotal + 2, Chr$(0))
    sockRecv.GetData data, , bytesTotal
    lstRecvData.AddItem data
    ' Update the remote IP and port labels
    '
    lblRemoteIP.Caption = sockRecv.RemoteHostIP
    lblRemotePort.Caption = sockRecv.RemotePort
End Sub

Private Sub sockRecv_Error(ByVal Number As Integer, _
        Description As String, ByVal Scode As Long, _
        ByVal Source As String, ByVal HelpFile As String, _
        ByVal HelpContext As Long, CancelDisplay As Boolean)
    MsgBox Description
End Sub

Private Sub Timer1_Timer()
    ' When the timer goes off, update the socket status labels
    '
     Select Case sockSend.State
        Case sckClosed
            lblSenderState.Caption = "sckClosed"
        Case sckOpen
            lblSenderState.Caption = "sckOpen"
        Case sckListening
            lblSenderState.Caption = "sckListening"
        Case sckConnectionPending
            lblSenderState.Caption = "sckConnectionPending"
        Case sckResolvingHost
            lblSenderState.Caption = "sckResolvingHost"
        Case sckHostResolved
            lblSenderState.Caption = "sckHostResolved"
        Case sckConnecting
            lblSenderState.Caption = "sckConnecting"
        Case sckClosing
            lblSenderState.Caption = "sckClosing"
        Case sckError
            lblSenderState.Caption = "sckError"
        Case Else
            lblSenderState.Caption = "unknown"
    End Select
    Select Case sockRecv.State
        Case sckClosed
            lblReceiverState.Caption = "sckClosed"
        Case sckOpen
            lblReceiverState.Caption = "sckOpen"
        Case sckListening
            lblReceiverState.Caption = "sckListening"
        Case sckConnectionPending
            lblReceiverState.Caption = "sckConnectionPending"
        Case sckResolvingHost
            lblReceiverState.Caption = "sckResolvingHost"
        Case sckHostResolved
            lblReceiverState.Caption = "sckHostResolved"
        Case sckConnecting
            lblReceiverState.Caption = "sckConnecting"
        Case sckClosing
```

```
            lblReceiverState.Caption = "sckClosing"
        Case sckError
            lblReceiverState.Caption = "sckError"
        Case Else
            lblReceiverState.Caption = "unknown"
    End Select
End Sub
```

When you look at the form, you see two Winsock controls. One of them sends datagrams, and the other receives them. You can also see three group boxes: one for the sender, one for the receiver, and one for general Winsock information. For the sender, you need to put the recipient's host name or IP address somewhere. When you set the *RemoteHost* property, you can use either the machine's textual name or a string representation of the dotted-decimal numeric IP address. The control resolves the name if needed. You also need the remote port to which you will send the UDP packets. Also, notice the text box for the local port, *txtSendLocalPort*. For the sender, it doesn't really matter which local port you send the data on, only which port you're sending to. If you leave the local port set to 0, the system will assign an unused port. The last text box, *txtSendData*, is for the string data to be sent. In addition, there are two command buttons: one for sending the data and one for closing the socket. To send datagrams, you must bind the Winsock control to a remote address, a remote port, and a local port before you can send any data. If you want to change any one of these three parameters, you need to close the socket first and then rebind to the new parameters. That is why the form has a Close Socket button.

Sending UDP Messages

Now that you know the sender's general capabilities, let's look at the code behind the scenes. First, look at the *Form_Load* routine. The first step is to set the *Protocol* property of the *sockSend* Winsock control to UDP by using the *sckUDPProtocol* enumerated type. The other commands in this routine don't apply to the sending functionality except for disabling the *cmdCloseSend* command button. We do this for completeness because calling the *Close* method on an already closed control does nothing. Note that the default state of the Winsock control is closed.

Next, look at the *cmdSendDgram_Click* routine, which is triggered by clicking the Send Data button. This is the heart of sending a UDP message. The first step in the code is to check the socket's state. If the socket is in the closed state, the code binds the socket to a remote address, a remote port, and a local port. Once the code binds a UDP Winsock control with these parameters, the state of the control changes from *sckClosed* to *sckOpen*. If the code doesn't perform this check and attempts to bind the socket on every send, the run-time error 40020, “Invalid operation at current state,” will be generated. Once a socket is bound, it remains bound until it is closed. This is why the code enables the Close Socket button for the sending socket once the control is bound. The last step is to call the *SendData* method with the data the user wants to send. When the *SendData* method returns, the code has finished sending data.

Only two other subroutines are associated with sending UDP messages. The first is *cmdCloseSend*, which, as its name implies, closes the sending socket, allowing the user to change the remote host, remote port, or local port parameter before sending data again. The other routine is *sockSend_Error*, which is a Winsock event. This event is triggered whenever a Winsock error is generated. Because UDP is unreliable, few errors will be generated. If an error does occur, the code simply prints out the error's description. The only message a user might see in this application is a destination unreachable message.

Receiving UDP Messages

As you can see, sending a UDP packet with the Winsock control is simple and straightforward. Receiving UDP packets is even easier. Let's go back to the *Form_Load* routine to see what needs to be done to receive a UDP message. As you saw with the sending Winsock control, the code sets the *Protocol* property to UDP. The code also disables the Close Listen button. Again, closing an already closed socket won't hurt, but the code does it for the sake of completeness. Also, it's always a good idea to think, “What could happen if I call method X?” at different points in the program. This is the source of most of the problems developers encounter with the control: calling a method when the state of control is invalid. An example of this is calling the *Connect* method on a Winsock control that is already connected.

To listen for incoming UDP packets, let's look at the *cmdListen_Click* routine. This is the handler for the Listen button. The only necessary step is to call the *Bind* method on the receiving Winsock control, passing the local port on which the user wants to listen for incoming UDP datagrams. When listening for incoming UDP packets, the code needs only the local port—the remote port on which the data was sent is not relevant. After the code binds the control, it disables the *cmdListen* button—this prevents the possibility of the user clicking the Listen button twice. Trying to bind an already bound control will cause a run-time error.

At this point, the *sockRecv* control is registered to receive UDP data. When the control receives UDP data on the port it's bound to, the *DataArrival* event is triggered. This event is implemented in the *sockRecv_DataArrival* routine. The parameter passed into the event, *bytesTotal*, is the number of bytes available to be read. The code allocates a string slightly larger than the amount of data being read. Then it calls the *GetData* method, passing the allocated string as the first parameter. The second parameter defaults to the Visual Basic type *vbString*, and the third parameter specifies the number of bytes that need to be read, which, in this example is the value *bytesTotal*. If the code requests to read a smaller number of bytes than that specified by the *bytesTotal* parameter, a run-time error is generated. Once the data is read into the character buffer, the code adds it to the list box of messages read. The last few steps in this subroutine set the label captions for the remote host's IP address and port number. Upon receipt of each UDP packet, the *RemoteHostIP* and *RemotePort* properties are set to the remote host's IP address and port number for the packet just received. Therefore, if the program receives multiple UDP packets from several hosts, the values of these properties will change often.

The last two subroutines that are associated with receiving UDP messages are *cmdCloseListen_Click* and *sockRecv_Error*. The user invokes the *cmdCloseListen_Click* handler by clicking the Close Listen button. The routine simply calls the *Close* method on the Winsock control. Closing a UDP control frees the underlying socket descriptor. The *sockRecv_Error* event is called whenever a Winsock error is generated. As we mentioned previously in the UDP send section, few UDP errors are generated to begin with because of their unreliable nature.

Obtaining Winsock Information

The last part of our UDP example is the Winsock Information group box. The local name and local IP labels are set at form load time. As soon as the form loads and Winsock controls are instantiated, the properties *LocalHostName* and *LocalIP* are set to the host name and IP address of the host machine and can be read at any time. The next two labels, Sender state and Receiver state, display the current state of the two Winsock controls that the application uses. The state information is updated every half second. This is where the Timer control comes in. Every 500 milliseconds, the Timer control triggers the Timer handler, which queries the socket states and updates the labels. We print the socket states for informative purposes only. The last two labels, Remote IP and Remote Port, are set whenever a UDP message is received, as discussed in the previous paragraph.

Running the UDP Example

Now that you understand how to send and receive UDP messages, let's take a look at the example as it runs. The best way to test it is to run an instance of the application on three separate machines. On one of the applications, click the Listen button. On the other two, set the *Recipient's Name/IP* field to the name of the machine on which the first application is running. This can be either a host name or an IP address. Now click the Send Datagram button a few times, and the messages should appear in the receiver's message window. Upon receipt of each message, the *Winsock Information* fields should be updated with the sender's IP address and the port number on which the message was sent. You can even use the Sender commands on the same application as the receiver to send messages on the same machine.

Another interesting test is using either subnet-directed broadcasts or broadcast datagrams. Assuming that you're testing all three machines on the same subnet, you can send a datagram to a specified subnet and all listening applications receive the message. For example, on our test machines we have two single-homed machines with IP addresses 157.54.185.186 and 157.54.185.224. The last machine is multihomed, with the IP addresses 169.254.26.113 and 157.54.185.206. As you can see, all three machines share the subnet 157.54.185.255.

Let's digress for a moment to discuss an important detail. If you want to receive UDP messages, you must implicitly bind to the first IP address stored in the network bindings when you call the *Bind* method. This is sufficient if your machine has only one network card. In some cases, however, a machine has more than one network interface and therefore more than one IP address. In these cases, the second parameter to the *Bind* method is the IP address on which to bind. Unfortunately, the Winsock control property *LocalIP* returns only one IP address, and the control provides no other method for obtaining other IP addresses associated with the local machine.

Now let's try some broadcasting. Close each sending or listening socket on each of the instances running. On the two single-homed machines, click the Listen button so that each machine can receive datagram messages. We don't use the multihomed machine because we aren't binding to any particular IP address in the code. On the third machine, enter the recipient's address as 157.54.185.255 and click the Send Data button a few times. You should see the message being received by both listening applications. If your sending machine is also multihomed, you might be wondering how it knows which network interface to send the datagram over. It is one of the routing table's functions to determine the best interface to send the message over, given the message's destination address and the address of each interface on the local machine. If you would like to learn more about subnets and routing, consult a book on TCP/IP such as ***TCP/IP Illustrated, Volume 1***, by W. Richard Stevens (Addison-Wesley, 1994), or ***TCP/IP: Architecture, Protocols, and Implementation with IPv6 and IP Security***, by Dr. Sidnie Feit (McGraw-Hill, 1996). The last test to try is to close the sender's socket on the third machine, enter the recipient's address as 255.255.255.255, and click the Send Datagram button a few times. The results should be the same: the other two listening programs should receive the message. However, the only difference on a multihomed machine is that the UDP message is being broadcast on each network attached to the machine.

UDP States

You might be a bit confused by the order in which method calls should be made to successfully send or receive datagrams. As mentioned previously, the most common mistake when programming the Winsock control is to call a method whose operation is not valid for the current state of the control. To help prevent this kind of mistake, look at Figure 14-2, which is a state diagram of the socket states when you are using UDP messages. Notice that the default starting state is always *sckClosed*, and no errors are generated for invalid host names.

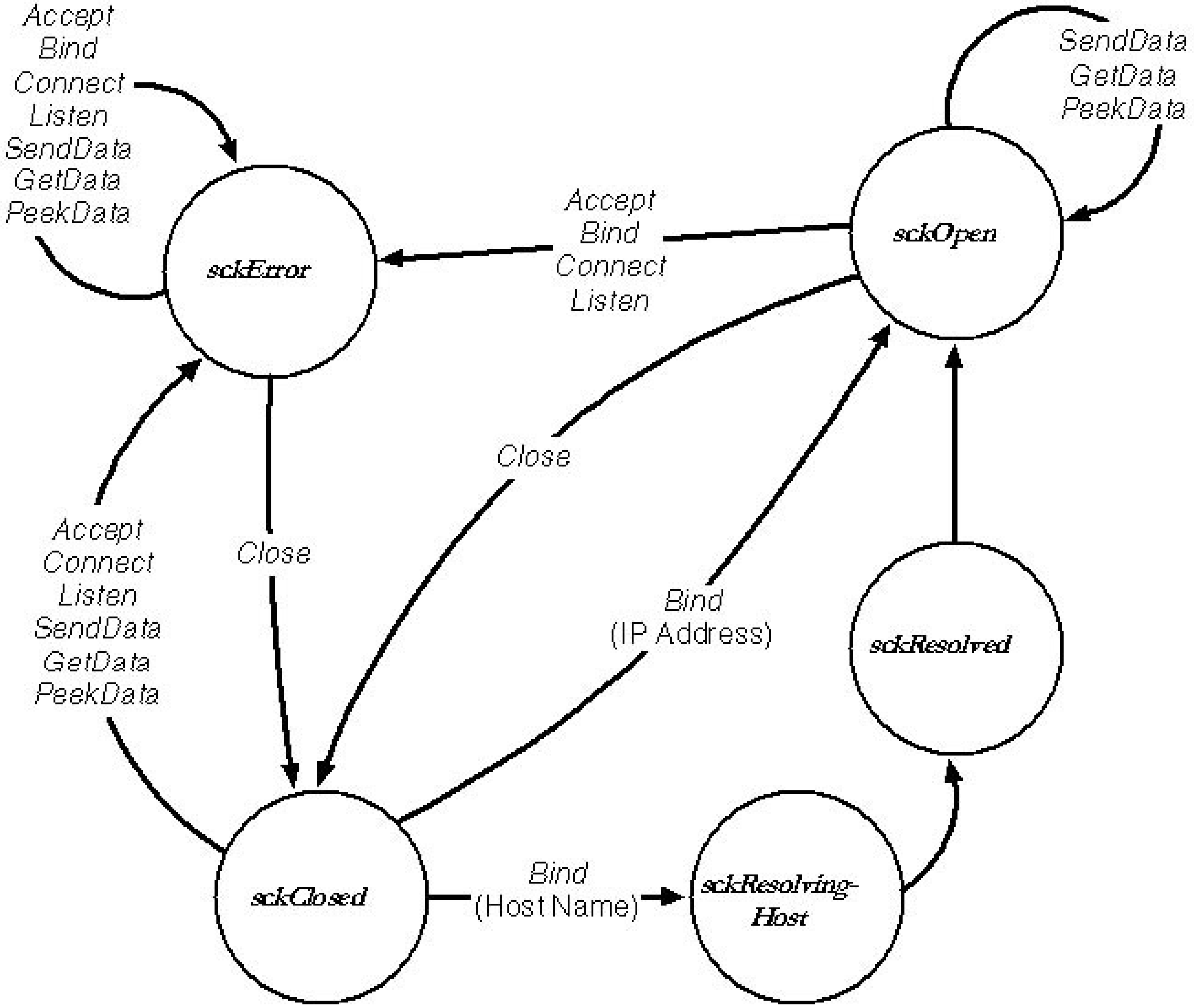
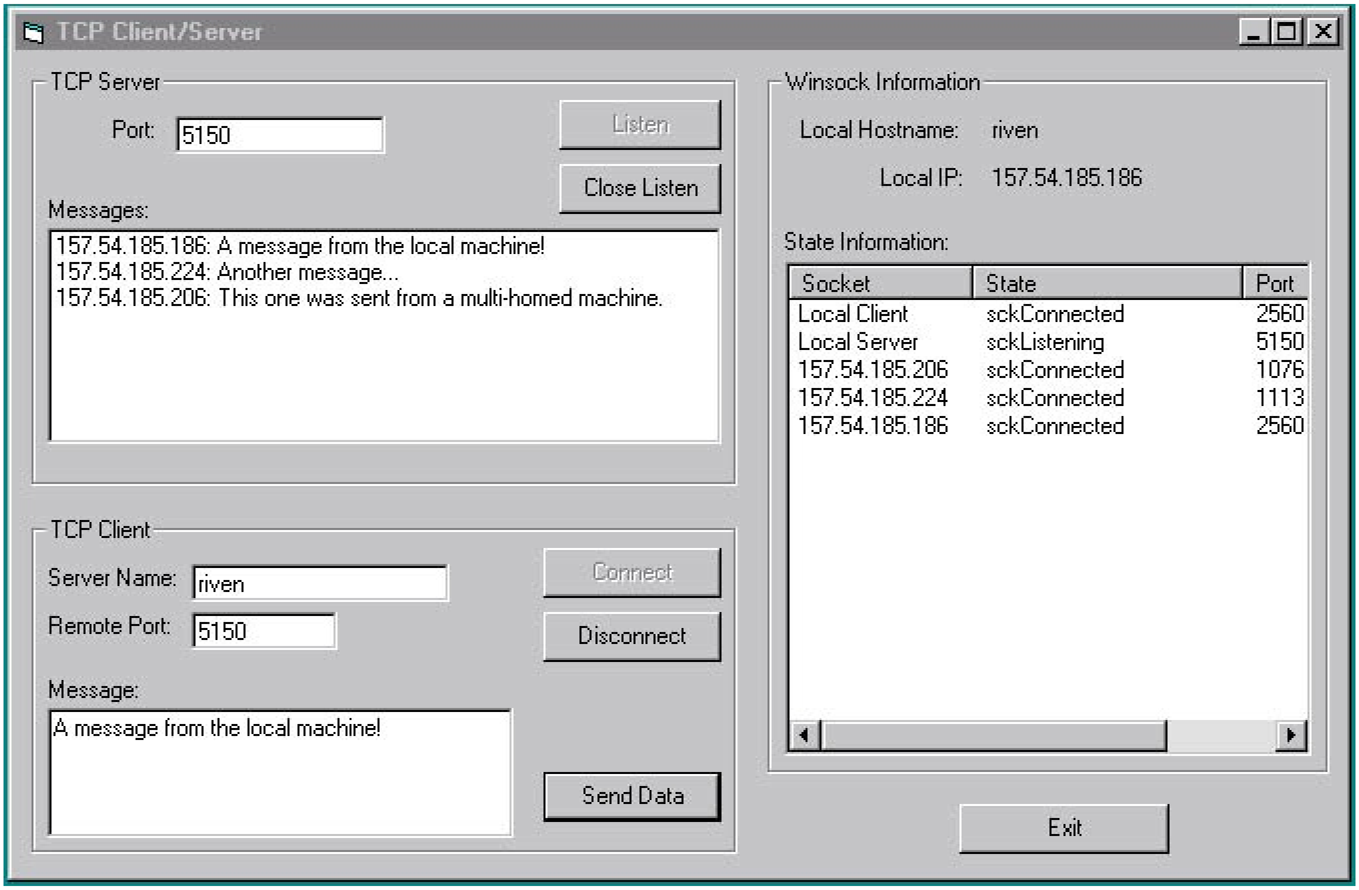


Figure 14-2 *UDP state diagram*

TCP Example

Using a Winsock control with the TCP protocol is a bit more involved and complex than using the control with its UDP counterpart. As we did with UDP, we will present a sample TCP application and go over its specifics in order to gain an understanding of the steps necessary to successfully use a TCP connection. Figure 14-3 shows the application running.



Port: 5150

Messages:

```
157.54.185.186: A message from the local machine!
157.54.185.224: Another message...
157.54.185.206: This one was sent from a multi-homed machine.
```

Server Name: riven

Remote Port: 5150

Message: A message from the local machine!

Local Hostname: riven

Local IP: 157.54.185.186

State Information:

Socket	State	Port
Local Client	sckConnected	2560
Local Server	sckListening	5150
157.54.185.206	sckConnected	1076
157.54.185.224	sckConnected	1113
157.54.185.186	sckConnected	2560

Figure 14-3 *Sample TCP application*

Let's take a look at the form in Figure 14-3 to gain an understanding of this application's capabilities. Again, you'll notice three group boxes: TCP Server, TCP Client, and Winsock Information. First, we'll discuss the TCP Server portion of the application. The server has a text box, *txtServerPort*, for the local port that the server will be bound to in order to listen for incoming client connections. Also, the server has two buttons, one to put the server in listening mode and the other to shut down the server and stop accepting incoming connections. Finally, the server has a single Winsock control named *sockServer*. If you take a look at the properties page, you'll see that the *Index* property has been set to 0. The control is actually an array capable of holding many instances of the Winsock control. The 0 signifies that at form load time only one instance (element 0 of the array) will be created. At any time we can dynamically load another instance of a Winsock control into an element of the array.

The Winsock control array is the basis of our server capabilities. Remember that a single Winsock control has only one socket handle associated with it. In Chapter 1, you learned that when a server accepts an incoming connection, a new socket is created to handle that connection. Our application is designed to dynamically load additional Winsock controls on a client connection so that the connection can be passed to the newly loaded control without interrupting the server socket to handle the connection. Another way to accomplish this is to put ***x*** number of Winsock controls on the form at design time. However, this is wasteful and does not scale well. When the application begins, a great deal of time will be spent loading all the resources necessary for every control; there is also the issue of how many controls to use. By placing *x* number of controls, you limit yourself to *x* number of concurrent clients. If your application requirements allow for only a fixed number of concurrent connections, placing a fixed number of Winsock controls on the form will work and is probably a bit simpler than using an array. For most applications, however, an array of Winsock controls is the best way to go.

The following is the code sample for this section. You can find the code for this Visual Basic project on the companion CD in a file called SOCKTCP.VBP.

```
Option Explicit

' The index value of the last Winsock control dynamically loaded
' in the sockServer array
Private ServerIndex As Long

Private Sub cmdCloseListen_Click()
    Dim itemx As Object
    ' Close the server's listening socket. No more
    ' clients will be allowed to connect.
    '
    sockServer(0).Close
    cmdListen.Enabled = True
    cmdCloseListen.Enabled = False

    Set itemx = lstStates.ListItems.Item(2)
    itemx.SubItems(2) = "-1"
End Sub

Private Sub cmdConnect_Click()
    ' Have the client control attempt to connect to the
    ' specified server on the given port number
    '
    sockClient.LocalPort = 0
    sockClient.RemoteHost = txtServerName.Text
    sockClient.RemotePort = CInt(txtPort.Text)
    sockClient.Connect

    cmdConnect.Enabled = False
End Sub

Private Sub cmdDisconnect_Click()
    Dim itemx As Object
    ' Close the client's connection and set up the command
    ' buttons for subsequent connections
    '
    sockClient.Close

    cmdConnect.Enabled = True
    cmdSendData.Enabled = False
    cmdDisconnect.Enabled = False
    ' Set the port number to -1 to indicate no connection
    '
    Set itemx = lstStates.ListItems.Item(1)
    itemx.SubItems(2) = "-1"
End Sub

Private Sub cmdExit_Click()
    Unload Me
End Sub
Private Sub cmdListen_Click()
    Dim itemx As Object
    ' Put the server control into listening mode on the given
    ' port number
    '
    sockServer(0).LocalPort = CInt(txtServerPort.Text)
    sockServer(0).Listen

    Set itemx = lstStates.ListItems.Item(2)
    itemx.SubItems(2) = sockServer(0).LocalPort

    cmdCloseListen.Enabled = True
    cmdListen.Enabled = False
End Sub

Private Sub cmdSendData_Click()
    ' If we're connected, send the given data to the server
    '
    If (sockClient.State = sckConnected) Then
        sockClient.SendData txtSendData.Text
    Else
        MsgBox "Unexpected error! Connection closed"
        Call cmdDisconnect_Click
    End If
End Sub

Private Sub Form_Load()
    Dim itemx As Object

    lblLocalHostname.Caption = sockServer(0).LocalHostName
    lblLocalHostIP.Caption = sockServer(0).LocalIP

    ' Initialize the Protocol property to TCP because that's
    ' all we'll be using
    '
    ServerIndex = 0
    sockServer(0).Protocol = sckTCPProtocol
    sockClient.Protocol = sckTCPProtocol
    ' Set up the buttons
    '
    cmdDisconnect.Enabled = False
    cmdSendData.Enabled = False
    cmdCloseListen.Enabled = False
    ' Initialize the ListView control that contains the
    ' current state of all Winsock controls created (not
    ' necessarily connected or being used)
    '
    Set itemx = lstStates.ListItems.Add(1, , "Local Client")
    itemx.SubItems(1) = "sckClosed"
    itemx.SubItems(2) = "-1"
    Set itemx = lstStates.ListItems.Add(2, , "Local Server")
    itemx.SubItems(1) = "sckClosed"
    itemx.SubItems(2) = "-1"
    ' Initialize the timer, which controls the rate of refresh
    ' on the socket states above
    '
    Timer1.Interval = 500
    Timer1.Enabled = True
End Sub

Private Sub sockClient_Close()
    sockClient.Close
End Sub

Private Sub sockClient_Connect()
    Dim itemx As Object

    ' The connection was successful: enable the transfer data
    ' buttons
    cmdSendData.Enabled = True
    cmdDisconnect.Enabled = True

    Set itemx = lstStates.ListItems.Item(1)
    itemx.SubItems(2) = sockClient.LocalPort
End Sub

Private Sub sockClient_Error(ByVal Number As Integer, _
        Description As String, ByVal Scode As Long, _
        ByVal Source As String, ByVal HelpFile As String, _
        ByVal HelpContext As Long, CancelDisplay As Boolean)
    ' An error occurred on the Client control: print a message,
    ' and close the control. An error puts the control in the
    ' sckError state, which is cleared only when the Close
    ' method is called.
    MsgBox Description
    sockClient.Close
    cmdConnect.Enabled = True
End Sub
Private Sub sockServer_Close(index As Integer)
    Dim itemx As Object
    ' Close the given Winsock control
    '
    sockServer(index).Close

    Set itemx = lstStates.ListItems.Item(index + 2)
    lstStates.ListItems.Item(index + 2).Text = "---.---.---.---"
    itemx.SubItems(2) = "-1"

End Sub

Private Sub sockServer_ConnectionRequest(index As Integer, _
        ByVal requestID As Long)
    Dim i As Long, place As Long, freeSock As Long, itemx As Object

    ' Search through the array to see whether there is a closed
    ' control that we can reuse
    freeSock = 0
    For i = 1 To ServerIndex
        If sockServer(i).State = sckClosed Then
            freeSock = i
            Exit For
        End If
    Next i
    ' If freeSock is still 0, there are no free controls
    ' so load a new one
    '
    If freeSock = 0 Then
        ServerIndex = ServerIndex + 1
        Load sockServer(ServerIndex)

        sockServer(ServerIndex).Accept requestID
        place = ServerIndex
    Else
        sockServer(freeSock).Accept requestID
        place = freeSock
    End If
    '  If no free controls were found, we added one above.
    '  Create an entry in the ListView control for the new
    '  control. In either case, set the state of the new
    '  connection to sckConnected.
    '
    If freeSock = 0 Then
        Set itemx = lstStates.ListItems.Add(, , _
            sockServer(ServerIndex).RemoteHostIP)
    Else
        Set itemx = lstStates.ListItems.Item(freeSock + 2)
        lstStates.ListItems.Item(freeSock + 2).Text = _
            sockServer(freeSock).RemoteHostIP
    End If
    itemx.SubItems(2) = sockServer(place).RemotePort

End Sub

Private Sub sockServer_DataArrival(index As Integer, _
        ByVal bytesTotal As Long)
    Dim data As String, entry As String

    ' Allocate a large enough string buffer and get the
    ' data
    '
    data = String(bytesTotal + 2, Chr$(0))
    sockServer(index).GetData data, vbString, bytesTotal
    ' Add the client's IP address to the beginning of the
    ' message and add the message to the list box
    '
    entry = sockServer(index).RemoteHostIP & ": " & data
    lstMessages.AddItem entry
End Sub

Private Sub sockServer_Error(index As Integer, _
        ByVal Number As Integer, Description As String, _
        ByVal Scode As Long, ByVal Source As String, _
        ByVal HelpFile As String, ByVal HelpContext As Long, _
        CancelDisplay As Boolean)
    ' Print the error message and close the specified control.
    ' An error puts the control in the sckError state, which
    ' is cleared only when the Close method is called.
    MsgBox Description
    sockServer(index).Close
End Sub

Private Sub Timer1_Timer()
    Dim i As Long, index As Long, itemx As Object

    ' Set the state of the local client Winsock control
    '
    Set itemx = lstStates.ListItems.Item(1)
    Select Case sockClient.State
        Case sckClosed
            itemx.SubItems(1) = "sckClosed"
        Case sckOpen
            itemx.SubItems(1) = "sckOpen"
        Case sckListening
            itemx.SubItems(1) = "sckListening"
        Case sckConnectionPending
            itemx.SubItems(1) = "sckConnectionPending"
        Case sckResolvingHost
            itemx.SubItems(1) = "sckResolvingHost"
        Case sckHostResolved
            itemx.SubItems(1) = "sckHostResolved"
        Case sckConnecting
            itemx.SubItems(1) = "sckConnecting"
        Case sckConnected
            itemx.SubItems(1) = "sckConnected"
        Case sckClosing
            itemx.SubItems(1) = "sckClosing"
        Case sckError
            itemx.SubItems(1) = "sckError"
        Case Else
            itemx.SubItems(1) = "unknown: " & sockClient.State
    End Select
    ' Now set the states for the listening server control as
    ' well as any connected clients
    '
    index = 0
    For i = 2 To ServerIndex + 2
        Set itemx = lstStates.ListItems.Item(i)

        Select Case sockServer(index).State
            Case sckClosed
                itemx.SubItems(1) = "sckClosed"
            Case sckOpen
                itemx.SubItems(1) = "sckOpen"
            Case sckListening
                itemx.SubItems(1) = "sckListening"
            Case sckConnectionPending
                itemx.SubItems(1) = "sckConnectionPending"
            Case sckResolvingHost
                itemx.SubItems(1) = "sckResolvingHost"
            Case sckHostResolved
                itemx.SubItems(1) = "sckHostResolved"
            Case sckConnecting
```

```
                itemx.SubItems(1) = "sckConnecting"
            Case sckConnected
                itemx.SubItems(1) = "sckConnected"
            Case sckClosing
                itemx.SubItems(1) = "sckClosing"
            Case sckError
                itemx.SubItems(1) = "sckError"
            Case Else
                itemx.SubItems(1) = "unknown"
        End Select
        index = index + 1
    Next i
End Sub
```

TCP Server

Now we'll examine the code behind the form. Take a look at the *Form_Load* procedure in the previous code sample. The first two statements simply set two labels to the local machine's host name and IP address. These labels are in the Winsock Information group box, which serves the same purpose as the informational box in the UDP example. Next, you'll see the initialization of the server control, *sockServer*, to the TCP protocol. Element 0 of the Winsock control array is always the listening socket. After this, the procedure disables the Close Listen button, which is enabled again later when the server starts to listen for clients. The last part of the procedure sets up the *ListView* control, *lstStates*. This control is used to display the current state of every Winsock control in use. The code adds entries for the client and server controls so that they are elements 1 and 2, respectively. Any other dynamically loaded Winsock controls will be added after these two. The entry for the server control is named “Local Server.” As in the UDP example, the procedure sets a timer to regulate how often the socket states are updated. By default, the timer triggers the update every half second.

From here, let's take a look at the two buttons that the server uses. The first is the Listen button, whose function is simple. The handler for the Listen button, *cmdListen*, sets the *LocalPort* property to the value the user entered in the *txtServerPort* text box. The local port is the most important field to a listening socket. This is the port all clients attempt to connect to in order to establish a connection. After setting the *LocalPort* property, all the code needs to do is call the *Listen* method. Once the Listen button's handler puts the *sockServer* control in listening mode, the program waits for the *ConnectionRequest* event to be fired on our *sockServer* control to indicate a client connection. The user can click the other button, Close Listen, to shut down the *sockServer* control. The Close Listen button's handler calls the *Close* method on *sockServer*(0), preventing any additional client connections.

The most important event for a TCP server is the *ConnectionRequest* event, which handles incoming client requests. When a client requests a connection, two options exist for handling the request. First, you can use the server socket to handle the client. The drawback of this method is that it will close the listening socket and prevent any other connections from being serviced. This method is accomplished by simply calling the *Accept* method on the server control with the *requestID* that is passed into the event handler. The other way to handle a client's connection request is to pass the connection to a separate control. This is what the companion CD example SockTCP.VBP does. Remember that you have an array of Winsock controls, and element 0 is the listening socket. The first thing to do is search through the array for a control whose state is closed (for example, query the *State* property for the value *sckClosed*). Of course, there are no free controls for the first loop because none are loaded. In this case, the first loop you see is not executed and the variable *freeSock* is still 0, indicating that no free controls were found. The next steps dynamically load a new Winsock control by incrementing the *ServerIndex* counter (the place in the array in which to load the control) and then executing the following statement:

```
Load sockServer(ServerIndex)
```

Now that a new Winsock control is loaded, the procedure can call the *Accept* method with the given request ID. The remaining statements add a new entry in the *lstStates* ListView control so that the program can display the current state of the new Winsock control.

With an already loaded Winsock control whose state is closed, the procedure simply would have reused that control by calling the *Accept* method on it. Continually loading and unloading controls is a bad idea because it decreases performance. The load and unload processes are expensive. There is also a memory leak when the Winsock control is unloaded, which we will discuss in detail later in this chapter.

The remaining server-side functions are straightforward. The *sockServer_Close* event is triggered whenever the client calls the *Close* method on its end. All the server does is close the socket on this side and clear the IP address entry in the ListView control by setting it to "---.---.---.---" and setting the entry's port to -1. The *sockServer_DataArrival* function allocates a buffer to receive the data and then calls the *GetData* method to perform the read. Afterward, the message is added to the *lstMessages* list box. The last server function is the *Error* event handler. Upon an error, the handler displays the text message and closes the control.

TCP Client

Now that you have seen how the server is implemented, let's examine the client. The only initialization that the client performs in the *Form_Load* procedure is setting the protocol of *sockClient* to TCP. Other than the initialization code, three command button handlers belong to the client and several event handlers. The first button is Connect, and its handler is named *cmdConnect_Click*. *LocalPort* is set to 0 because it doesn't matter what the local system assigns because the port number is on our machine. *RemoteHost* and *RemotePort* are set according to the values in the *txtServerName* and *txtPort* fields, respectively. That's all the information required to establish a TCP connection to a server. The only task left is to call the *Connect* method. After that, the control's state is in the process of either resolving the name or connecting (the control's state will be *sckResolvingHost*, *sckResolved*, or *sckConnecting*). When the connection finally is established, the state changes to *sckConnected* and the *Connect* event is triggered. The next section covers the various states and the transitions among them.

Once the connection is established, the handler *sockClient_Connect* is called. This handler simply enables the Send Data and Disconnect buttons. In addition, the port number on which the connection is established on the local machine is updated for the Local Client entry in the *lstStates* ListView control. Now you can send and receive data. There are two other event handlers: *sockClient_Close* and *sockClient_Error*. The *sockClient_Close* event handler simply closes the client Winsock control, and the *sockClient_Error* event handler displays a message box with the error description and then closes the control.

The last two pieces to the client are the remaining command buttons: Send Data and Disconnect. The subroutine *cmdSendData_Click* handles the Send Data button. If the Winsock control is connected, the routine calls the *SendData* method with the string in the *txtSendData* text box. Finally, the Disconnect button is handled by *cmdDisconnect_Click*. This handler simply closes the client control, resets a number of buttons to their initial state, and updates the Local Client entry in the *lstStates* ListView control.

Obtaining Winsock Information

The last part of the TCP example is the Winsock Information section. We have already explained this a bit, but we'll briefly present it here for clarity. As with the UDP example, a timer triggers an update on the current socket states of all loaded Winsock controls. The default refresh rate is set to 500 milliseconds. Upon load, two entries are added to the *lstStates* Listview control. The first is the Local Client label that corresponds to the client Winsock control, *sockClient*. The second entry is Local Server, which refers to the listening socket. Whenever a new client connection is established, a new Winsock control is dynamically loaded and a new entry is added to the *sckStates* control; the name of the entry is the client's IP address. When a client disconnects, the entry is set to the default state with the IP address "---.---.---.---" and port number equal to -1. Of course, if another client connects, it reuses any unused controls in the server array. The local machine's IP address and host name are also displayed.

Running the TCP Example

Again, running the TCP example is a straightforward process. Start three instances of the application, each on a separate machine. With TCP, it doesn't matter if any of the machines are multihomed because the routing table decides which interface is more appropriate for any given TCP connection. On one of the TCP examples, start the listening socket by clicking the Listen button. You'll notice that the Local Server entry in the State Information ListView control changes from *sckClosed* to *sckListening* and the port number is listed as 5150. The server is now ready to accept client connections.

On one of the clients, set the *Server Name* field to the name of the machine running the first instance of the application (the listening server) and then click the Connect button. On the client application, the Local Client entry in the State Information list is now in the *sckConnected* state and the local port on which the connection was made is updated to a non-negative number. In addition, on the server side, an entry is added to the State Information list whose name is the IP address of the client that just connected. The new entry's state is *sckConnected* and also contains the port number on the server side on which the connection was established. Now you can type text into the *Message* text field on the client and click the Send Data button a few times. You will see the messages appearing in the Messages list box on the server side. Next, disconnect the client by clicking the Disconnect button. On the client side, the Local Client entry in the State Information list is set back to *sckClosed* and the port number value to -1. For the server, the entry corresponding to the client is not removed; it is simply marked as unused with the name set to a dashed IP address, the state to *sckClosed*, and the port to -1.

On the third machine, enter the name of the listening server in the Server Name text box and make a client connection. The results are similar to those for the first client except that the server uses the same Winsock control to handle this client as it did for the first. If a Winsock control is in the closed state, it can be used to accept any incoming connection. The final step you might want to try is using the client on the server application to make a connection locally. After you make the connection, a new entry is added to the Socket Information list, as in the earlier examples. The only difference is that the IP listed is the same as the IP address of the server. Play with the clients and server a bit to get a feel for how they interact and what results each command triggers.

TCP States

Using the Winsock control with the TCP protocol is much more complicated than using UDP sockets because many more socket states are possible. Figure 14-4 is a state diagram for a TCP socket. The default start state is *sckClosed*. The transitions are straightforward and they don't require explanation except for the *sckClosing* state. Because of the TCP half-close, there are two transition paths from this state for the *SendData* method. Once one side of the TCP connection issues a *Close* method, that side can't send any more data. The other side of the connection receives the *Close* event and enters the *sckClosing* state but can still send data. This is why there are two paths out of *sckClosing* for the *SendData* method. If the side that issued the *Close* tries to call *SendData*, an error is generated and the state moves to *sckError*. The side that receives the *Close* event can freely send data and receive any remaining data.

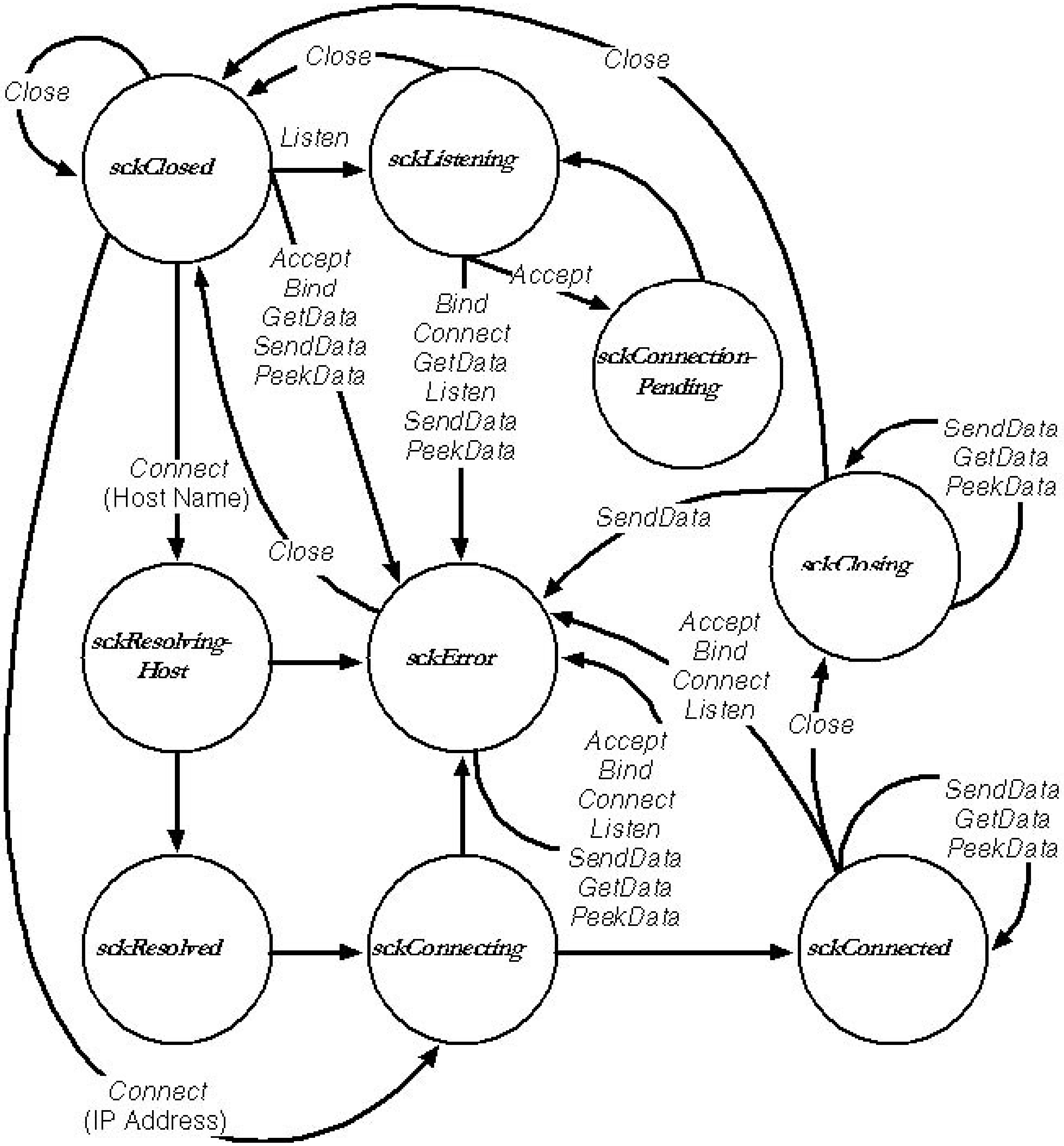


Figure 14-4 *TCP state diagram*

Limitations

The Winsock control is clearly useful and easy to use; unfortunately, a few bugs make the control unusable for mission-critical applications. The bugs discussed in this section apply to the latest version of the control for Visual Basic 5, which is the updated control from Service Pack 2.

The first bug is relatively minor and deals with dynamically loading and unloading the control. A memory leak is incurred when unloading a previously loaded control. This is why we don't load and unload the controls as clients connect and disconnect in our server example. Once the control is loaded in memory, we leave it for possible use by other clients.

The second bug involves closing a socket connection before all data queued is sent on the wire. In some cases, calling the *Close* method after the *SendData* event (when *Close* is processed before *SendData*) causes data to be lost, at least from the receiver's point of view. You can get around this problem by catching the *SendComplete* event (which is triggered when *SendData* has finished putting the data on the wire). Alternatively, you could arrange the send/receive transactions so that the receiver issues the *Close* command first, when it has received all the data expected. This would then trigger the *Close* event on the sender, which would then signal that all the data sent has been received, and that it's now OK to shut down the connection completely.

The last and most severe bug is the dropping of data when a large buffer is submitted for transfer. If a large enough block of data is queued up for network transmission, the control's internal buffers get messed up and some data is dropped. Unfortunately, there is no completely perfect workaround for this problem. The best method is to submit data in chunks less than 1000 bytes. Once a buffer is submitted, wait for the *SendComplete* event to fire before submitting the next buffer. This is a hassle, but it's still the best way to make the control as reliable as possible.

The latest Winsock control shipping in Visual Basic 6 has fixed these bugs except for the second one. If you issue a *Close* command after calling *SendData*, the socket closes immediately without sending all the data. Although it would have been wonderful to have all the bugs fixed, the remaining problem is perhaps the least severe of the three and the easiest to work around.

Common Errors

As you saw in Chapters 1 through 12, an application can encounter quite a few Winsock errors. We won't go into all of them here. However, in the following two sections we will discuss the errors that are most commonly encountered by applications using the Winsock control: “Local address in use” and “Invalid operation at current state.”

Local Address in Use

The “Local address in use” error occurs when you bind to a local port, either through the *Bind* method or the *Connect* method, but find that the port is already in use. This is most often encountered in the TCP server that always binds to a specific port so clients can locate the service. If a socket is not properly closed before an application using that socket exits, the socket goes into the TIME_WAIT state for a short time to ensure that all data has been sent or received on that port. If an attempt is made to bind to that port, the “Local address in use” error is generated. A common mistake on the client side also results in this error. If the *LocalPort* property is set to 0 and a connection is established, *LocalPort* is updated to the port number on which the client connection was made locally. If you plan to reuse the same control to make a subsequent connection, be sure you reset the *LocalPort* value to 0. Otherwise, if the previous connection was not properly shut down, you might run into this error.

Invalid Operation at Current State Run-Time Error

The “Invalid operation at current state” error is probably the most frequently seen error. It occurs when a Winsock control method is called but the current state of the control prohibits that action. Take a look at Figures 14-2 and 14-4 for the state diagrams for UDP and TCP sockets. To write robust code, always check the socket's state before calling a method.

Winsock errors will be generated through the *Error* event. These are the same errors as those from straight Winsock programming. For a more detailed description of Winsock errors, refer to Chapter 1, which covers the most common errors encountered, or consult Chapter 21, which lists all possible Winsock error codes.

The Windows CE Winsock Control

The Visual Basic Toolkit for Windows CE (VBCE) contains a Winsock control that provides many of the same capabilities offered by the “regular” Visual Basic Winsock control. The major difference is that while UDP is unsupported, the Windows CE Winsock control offers the IrDA protocol. Also, some minor differences between the two controls require some programmatic changes from what you have seen with the non–Windows CE Winsock control.

As you read in Chapter 1, Windows CE does not offer the asynchronous Winsock model. The Windows CE Winsock control is no exception. The main difference in programming is that the *Connect* method is blocking. There is no *Connect* event. Once you attempt a connection by calling *Connect*, the call will block until a connection is made or an error is returned.

In addition, VBCE 1 does not support control arrays, which means you will have to change your server design from that presented in the sample TCP application shown earlier in the chapter. As a result, the only way to handle multiple connections is to place a number of Windows CE Winsock controls onto the form. This limits the maximum number of concurrent client connections that you can handle because this solution does not scale at all.

Finally, the *ConnectionRequest* event doesn't have a *RequestID* parameter, which might seem a bit strange. The result is that you must call the *Accept* method on the control to which the connection will be handed off. The connection request that triggers the *ConnectionRequest* event is handled by the control that receives the connection request.

Windows CE Winsock Example

In this section, we'll briefly introduce a sample application using the Windows CE Winsock control. The same principles apply to the Windows CE control as to the desktop Winsock control except for the differences noted previously. The following sample shows the code behind the Windows CE Winsock control:

```
Option Explicit

' This global variable is used to retain the current value
' of the radio buttons. 0 corresponds to TCP, and 2 means
' IrDA (infrared). Note that UDP is not currently supported by the
' control.
Public SocketType

Private Sub cmdCloseListen_Click()
' Close the listening socket, and set the other buttons
' back to the start state
    WinSock1.Close

    cmdConnect.Enabled = True
    cmdListen.Enabled = True
    cmdDisconnect.Enabled = False
    cmdSendData.Enabled = False
    cmdCloseListen.Enabled = False
End Sub

Private Sub cmdConnect_Click()
' Check which type of socket type was chosen, and initiate
' the given connection

    If SocketType = 0 Then
        ' Set the protocol and the remote host name and port
        '
        WinSock1.Protocol = 0

        WinSock1.RemoteHost = txtServerName.Text
        WinSock1.RemotePort = CInt(txtPort.Text)
        WinSock1.LocalPort = 0

        WinSock1.Connect
    ElseIf SocketType = 2 Then
        ' Set the protocol to IrDA, and set the service name
        '
        WinSock1.Protocol = 2
        'WinSock1.LocalPort = 0
        'WinSock1.ServiceName = txtServerName.Text
        WinSock1.RemoteHost = txtServerName.Text

        WinSock1.Connect
    End If
    ' Make sure the connection was successful; if so,
    ' enable/disable some commands
    '
    MsgBox WinSock1.State
    If (WinSock1.State = 7) Then
        cmdConnect.Enabled = False
        cmdListen.Enabled = False
        cmdDisconnect.Enabled = True
        cmdSendData.Enabled = True
    Else
        MsgBox "Connect failed"
        WinSock1.Close
    End If
End Sub

Private Sub cmdDisconnect_Click()
' Close the current client connection, and reset the
' buttons to the start state
    WinSock1.Close

    cmdConnect.Enabled = True
    cmdListen.Enabled = True
    cmdDisconnect.Enabled = False
    cmdSendData.Enabled = False
    cmdCloseListen.Enabled = False
End Sub

Private Sub cmdListen_Click()
' Set the socket to listening mode for the given protocol
' type
```

```
'
    If SocketType = 0 Then
        WinSock1.Protocol = 0
        WinSock1.LocalPort = CInt(txtLocalPort.Text)
        WinSock1.Listen
    ElseIf SocketType = 2 Then
        WinSock1.Protocol = 2
        WinSock1.ServiceName = txtServerName.Text
        WinSock1.Listen
    End If
    ' If we're not in listening mode now, something
    ' went wrong
    '
    If (WinSock1.State = 2) Then
        cmdConnect.Enabled = False
        cmdListen.Enabled = False
        cmdCloseListen.Enabled = True
    Else
        MsgBox "Unable to listen!"
    End If
End Sub

Private Sub cmdSendData_Click()
' Send the data in the box on the current connection
'
    WinSock1.SendData txtSendData.Text
End Sub

Private Sub Form_Load()
' Set the initial values for the buttons, the timer, etc.
'
    optTCP.Value = True
    SocketType = 0

    Timer1.Interval = 750
    Timer1.Enabled = True

    cmdConnect.Enabled = True
    cmdListen.Enabled = True

    cmdDisconnect.Enabled = False
    cmdSendData.Enabled = False
    cmdCloseListen.Enabled = False

    lblLocalIP.Caption = WinSock1.LocalIP
End Sub

Private Sub optIRDA_Click()
' Set the socket type to IrDA
'
    optIRDA.Value = True
    SocketType = 2
End Sub

Private Sub optTCP_Click()
' Set the socket type to TCP
'
```

```
    optTCP.Value = True
    SocketType = 0
    cmdConnect.Caption = "Connect"
End Sub

Private Sub Timer1_Timer()
' This is the event that gets called each time the
' timer expires. Update the socket state label.
'
    Select Case WinSock1.State
        Case 0
            lblState.Caption = "sckClosed"
        Case 1
            lblState.Caption = "sckOpen"
        Case 2
            lblState.Caption = "sckListening"
        Case 3
            lblState.Caption = "sckConnectionPending"
        Case 4
            lblState.Caption = "sckResolvingHost"
        Case 5
            lblState.Caption = "sckHostResolved"
        Case 6
            lblState.Caption = "sckConnecting"
        Case 7
            lblState.Caption = "sckConnected"
        Case 8
            lblState.Caption = "sckClosing"
        Case 9
            lblState.Caption = "sckError"
    End Select
End Sub

Private Sub WinSock1_Close()
' The other side initiated a close, so we'll close our end.
' Reset the buttons to their initial state.
'
    WinSock1.Close

    cmdConnect.Enabled = True
    cmdListen.Enabled = True

    cmdDisconnect.Enabled = False
    cmdSendData.Enabled = False
    cmdCloseListen.Enabled = False
End Sub

Private Sub WinSock1_ConnectionRequest()
' We got a client connection; accept it on the listening
' socket
'
    WinSock1.Accept
End Sub

Private Sub WinSock1_DataArrival(ByVal bytesTotal)
' This is the event for data arrival. Get the data, and
' add it to the list box.
```

```
'
    Dim rdata

    WinSock1.GetData rdata
    List1.AddItem rdata
End Sub

Private Sub WinSock1_Error(ByVal number, ByVal description)
' An error occurred; display the message, and close the socket
'
    MsgBox description
    Call WinSock1_Close
End Sub
```

We won't go into the specifics of this sample code because it is similar to the SockTCP example shown in the sample TCP application. The only differences between the two are the known limitations mentioned in the previous section. You will probably notice that the Windows CE Winsock control is a bare-bones control. It isn't as well polished as the desktop version. The type libraries aren't fully implemented—you must differentiate the protocol type with a simple integer as opposed to an enumerated type. In addition, there is the problem with the socket state enumerated type mentioned in the next section, “Known Problems.”

Handling infrared connections is not that different from handling TCP connections. The one exception occurs when a listening socket is established over the infrared port. An infrared server is known by its ***service name***, which is discussed in detail in the IrDA addressing section of Chapter 3. The Windows CE Winsock control has an additional property named *ServiceName*. You set this property to the text string that clients attempt to connect to. For example, the following code snippet puts the Windows CE Winsock control *CeWinsock* into listening mode under the name “MyServer”:

```
CeWinsock.Protocol = 2                ' Protocol 2 is IrSock
CeWinsock.ServiceName = "MyServer"
CeWinsock.Listen
```

There are no other requirements for publishing a service under infrared sockets. You need to specify only the service name.

Known Problems

The one rather strange problem we encounter with the VBCE Winsock control is the use of the enumerated values for the Winsock states. For some odd reason, these values are defined in the development environment, but on the remote device the following error message pops up every time you reference a *sck* enumerated value in your code: “An error was encountered while running this program.” If you replace the enumerations with their constant equivalents, these errors go away. This has been marked as a bug and will be corrected in a future release of the toolkit.

Conclusion

The Visual Basic Winsock control is useful for simple, non-critical applications that require network communication. A few problems with the Visual Basic 5 version of the control make successfully programming the control difficult, but most of the major problems have been corrected in the latest version of Visual Basic. The control offers the capability to add simple network communication to a Visual Basic application with relatively little effort. Of course, the control is limited in its overall capabilities, and applications that require a great deal of interaction with Winsock should consider either manually importing the necessary functions and constants from the Winsock DLL or use the new .NET Application Frameworks interface. As we mentioned earlier, we have provided some Winsock Visual Basic examples throughout the Winsock chapters that do in fact import Winsock functions from WS2_32.DLL. For examples, see the *TCPClient, TCPServer, UDPSender,* and *UDPReceiver* applications under the Chapter 1 Visual Basic directories of samples and their WINSOCK.BAS file.

Chapter 15

Remote Access Service

So far, this book has described the Winsock API available in Microsoft Windows that allows you to develop applications capable of communicating over a local network. This chapter describes an important service named Remote Access Service (RAS), which allows users to connect their computer to a remote network such as an ISP or a corporate network. Once connected, you can use the network functions described throughout this book as though your computer were connected directly to a remote network.

RAS Client

All Microsoft Windows platforms feature a RAS client, which allows you to connect your computer from a remote location to another computer network featuring a remote access server component. Typically, a RAS client will do this by using a serial communication device such as a modem that connects to a telephone line and calls the remote network by dialing a telephone number. Because of this, the RAS client is sometimes referred to as a dial-up networking (DUN) client. RAS also supports connecting to a remote network by tunneling connections securely over an IP network such as the Internet, which is known as Virtual Private Networking (VPN).

On the remote network, you must have a RAS server awaiting your DUN or VPN connections. A RAS client is capable of establishing a communication link with several types of remote access servers. RAS does this by using industry standard serial framing and IP tunneling protocols. The following protocols are serial framing, where data communication proceeds over a serial device such as a modem:

- Point-to-Point Protocol (PPP) Can transmit IP, IPX, and NetBEUI communication protocols
- Serial Line Internet Protocol (SLIP) Can transmit the IP communication protocol only
- Asynchronous NetBEUI (Microsoft Windows NT 3.1, Microsoft Windows for Workgroups 3.11) Can transmit the NetBEUI communication protocol only

RAS uses the following IP tunneling protocols where data communication proceeds over an existing IP connection:

- Point-to-Point Tunneling Protocol (PPTP) Can securely transmit IP and IPX communication protocols
- Layer 2 Tunneling Protocol (L2TP) Can securely transmit IP and IPX communication protocol

The framing and tunneling protocols describe how data is transmitted over a RAS communication link and dictate which network communication protocols (such as TCP/IP or IPX) can communicate over the link. If a RAS server supports one of the framing protocols defined in the previous list, a RAS client can establish a connection. All Windows 95, Windows 98, Windows Me, and Windows NT platforms feature a RAS server component capable of supporting the serial framing protocols listed. Windows Me and all Windows NT platforms also support the IP tunneling protocols.

Once a connection between a RAS client and server is established, network protocol stacks (depending on the framing or tunneling protocol used) can communicate over the RAS connection to the remote network as if the computers were connected directly over a LAN. Of course, the data communication rate of most RAS connections is typically slower than for a direct LAN connection.

When a RAS server accepts a serial framing or an IP tunneling connection, it first establishes communication with your client by negotiating one of the framing or tunneling protocols in the previous list. Once the protocol connection is established, the RAS server attempts to authenticate the user making the connection. The RAS API functions this chapter describes allow a RAS client to specify a user name, a password, and domain logon credentials to the RAS server. When a Windows NT–based RAS server receives this information, it validates these logon credentials using Windows NT domain security access control. Note that the RAS server does not log your client on to a Windows NT domain; instead, it uses the client credentials to verify that a user is allowed to make a RAS connection. The RAS connection process is not the same as the Windows NT domain logon process. After a RAS connection is successfully established, your computer can log on to a Windows NT domain. On Windows 95, Windows 98, Windows Me, Windows XP, and Windows .NET Server, RAS can automatically log a machine on to a domain after a RAS connection is authenticated through options available in a phonebook entry, as we will discuss later in this chapter.

For serial communications, RAS relies on the Telephony Application Programming Interface (TAPI) to set up and control serial communication devices such as modems on your computer. TAPI controls the hardware settings of these dialing devices. When you set up a RAS connection using a modem, TAPI turns on the modem and sends dialing information from RAS to the modem. As a result, RAS views modems as simple TAPI modem ports that are capable of dialing and making a phone connection to a remote server. As you will see later in this chapter, some of the RAS API functions refer to TAPI modem ports when you set up RAS connection information.

This chapter will explain how you can use RAS programmatically to establish remote network communication. We will begin by describing the header and library files you need to build your application. Next, we will describe the basics of dialing—how you actually establish a remote connection over a serial device. Then we'll describe how you can set up RAS phonebook entries to define detailed communication properties of a RAS connection. Once we've explained the basics of setting up communication, we'll show you how to manage established connections. Finally, we'll describe how to set up a VPN connection.

Compiling and Linking

When you develop a RAS application, you need to include the following header and library files to build your application:

- Contains the function prototypes and data structures RAS API functions use
- Contains predefined error codes used in RAS API functions when they fail
- Library of all RAS API functions

RASERROR.H lists quite a few predefined error codes. In this file, you will notice an error description string associated with each error code used in RAS. RAS features a useful function named that allows you to programmatically retrieve the error strings associated with specific RAS error codes. is defined as

```
DWORD RasGetErrorString(
    UINT uErrorValue,
    LPTSTR lpszErrorString,
    DWORD cBufSize
);
```

The parameter receives a specific RAS error code returned from a RAS function. The parameter is an application-supplied buffer that will receive the error string associated with the error code in . You should make your buffer large enough to hold an error string; otherwise, this function will fail with error . We recommend setting your buffer size to at least 256 characters, which should accommodate any RAS error string available today. The final parameter, , is the size of the buffer you supplied as .

Data Structures and Platform Compatibility Issues

When you compile and build your application, you will find that some of the data structures the RAS functions use have extra data fields included or excluded, based on the value of the define . However, the Windows CE SDK does not define , so this information does not apply to Windows CE. RAS data structures also have a field that you must set to the byte size of the RAS structure you are using. This affects the behavior of the RAS functions that use these structures because they are targeted for a specific platform. The following rules apply to the Windows 95, Windows 98, Windows Me, and Windows NT platforms:

- Indicates that your RAS application is targeted for Windows 95, Windows 98, or Windows NT 4.0 with no service pack
- Indicates that your RAS application is targeted for Windows NT 4 with any service pack
- Indicates that your RAS application is targeted for Windows 2000
- Indicates that your RAS application is targeted for Windows XP and Windows .NET Server

RAS does not lend itself well to having a single executable that can run on all platforms because the RAS data structures will be sized differently during program compilation based on values. Through careful programming, it is possible to support all platforms (except Windows CE, of course) using a single executable. However, we highly recommend targeting a specific platform when you build your RAS applications.

DUN 1.3 Upgrade and Windows 95

A number of Windows 95 releases occurred between the original release and the OSR 2 release. The OSR 2 release was not a retail product; instead, it was available only for original equipment manufacturers (OEMs) to install on a new computer. Each release features a different flavor of the RAS API functionality. Because of this, we recommend on Windows 95 platforms that you install the latest RAS upgrade package—DUN 1.3—to raise RAS to the current functionality level of newer platforms. You can obtain the DUN 1.3 upgrade for Windows 95 at http://www.microsoft.com/support. This chapter assumes that you have installed at least the DUN 1.3 upgrade; we will not address RAS issues from earlier DUN versions.

RasDial

When a RAS client application is ready to make a connection to a remote network, it must call the function. is quite complex, offering many call parameters that are used for dialing, authenticating, and establishing a remote connection to a RAS server. is defined as

```
DWORD RasDial(
    LPRASDIALEXTENSIONS lpRasDialExtensions,
    LPCTSTR lpszPhonebook,
    LPRASDIALPARAMS lpRasDialParams,
    DWORD dwNotifierType,
    LPVOID lpvNotifier,
    LPHRASCONN lphRasConn
);
```

Based on values of the parameter, can execute in two operating modes: and . In synchronous mode, blocks until it either completes a connection or fails to do so. In asynchronous mode, completes a connection immediately, allowing your application to perform other actions while connecting.

Synchronous Mode

If the parameter of is set to , will operate synchronously. When the parameter is , the parameter is ignored. Calling synchronously is the easiest way to use this function; however, you won't be able to monitor the connection progress like you can in asynchronous mode, which we will describe in a moment. The following code sample demonstrates how to call synchronously:

```
RASDIALPARAMS RasDialParams;
HRASCONN hRasConn;
DWORD Ret;

// Always set the size of the RASDIALPARAMS structure

RasDialParams.dwSize = sizeof(RASDIALPARAMS);
hRasConn = NULL;

// Setting this field to an empty string will allow
// RasDial to use default dialing properties

lstrcpy(RasDialParams.szEntryName, "");

lstrcpy(RasDialParams.szPhoneNumber, "867-5309");
lstrcpy(RasDialParams.szUserName, "jenny");
lstrcpy(RasDialParams.szPassword, "mypassword");
lstrcpy(RasDialParams.szDomain, "mydomain");

// Call RasDial synchronously (the fifth parameter
// is set to NULL)
Ret = RasDial(NULL, NULL, &RasDialParams, 0, NULL, &hRasConn);

if (Ret != 0)
```

```
{
    printf("RasDial failed: Error = %d\n", Ret);
}
```

The sample calls by filling fields of the parameter. The parameter is a structure pointer that defines dialing and user authentication parameters that the function uses to establish a remote connection. It's defined as

```
typedef struct _RASDIALPARAMS {
    DWORD dwSize;
    TCHAR szEntryName[RAS_MaxEntryName + 1];
    TCHAR szPhoneNumber[RAS_MaxPhoneNumber + 1];
    TCHAR szCallbackNumber[RAS_MaxCallbackNumber + 1];
    TCHAR szUserName[UNLEN + 1];
    TCHAR szPassword[PWLEN + 1];
    TCHAR szDomain[DNLEN + 1] ;
#if (WINVER >= 0x401)
    DWORD dwSubEntry;
    DWORD dwCallbackId;
#endif
} RASDIALPARAMS;
```

The fields of provide the basics for setting up a RAS connection and are described as

- Should be set to the size (in bytes) of a structure. This allows RAS to internally determine which version you compiled with.
- A string that allows you to identify a phonebook entry contained in the phonebook file listed in the parameter of . This is an important parameter because phonebook entries enable you to fine-tune RAS connection properties, such as selecting a modem or selecting a framing protocol. However, specifying a phonebook entry to use is optional. If this field is an empty string (“”), will select the first available modem installed on your system and will rely on the next parameter, , to dial a connection. We will describe phonebook entries in more detail later.
- A string representing a phone number that overrides the number contained in the phonebook entry specified in the field.
- Allows you to specify a phone number the RAS server can call you back on. If the RAS server permits you to have a callback number, the server will terminate your original connection and call back your client using the callback number you specified. This is a nice feature because it lets your server know where a user is connecting from.
- A string that identifies a logon name used to authenticate a user on a RAS server.
- A string that identifies the password used to authenticate a user on a RAS server.
- Identifies the Windows NT domain where the user account is located.
- Optionally allows you to specify the initial phonebook subentry to dial for a RAS multilink connection. We don't describe the multilink feature in this chapter.
- Allows you to pass an application-defined value to a callback function (which we'll also describe later). If you're not using a callback function, this field is not used.

The previous sample fills in a with a number to dial, user name, password, and domain information for a RAS connection. The structure is then passed into the function, which will synchronously make the remote connection.

Asynchronous Mode

Calling asynchronously is a lot more complicated than calling this function in synchronous mode, but it offers greater flexibility when establishing a connection. If the parameter of is not set to , will operate asynchronously—the call returns immediately but the connection proceeds. Calling asynchronously is the preferred method for making a RAS connection because you can monitor the connection's progress. The parameter can be either a pointer to a function that is called from when a connection activity occurs in or a window handle that receives progress notification via Windows messages. The parameter of determines the type of function or window handle that is passed into . Table 15-1 describes the values that you can specify in .

Table 15-1 *Asynchronous Notification Methods*

Notifier Type	Meaning
0	The parameter causes to use the function pointer to manage connection events.
1	The parameter causes to use the function pointer to manage connection events.
2	The parameter causes to use the function pointer to manage connection events.
0xFFFFFFFF	The parameter makes send a window message during connection events.

Table 15-1 also describes the three callback function prototypes you supply to in the parameter that is called for receiving notification of connection events: , , and . The first one, , is prototyped as

```
VOID WINAPI RasDialFunc(
    UINT unMsg,
    RASCONNSTATE rasconnstate,
    DWORD dwError
);
```

The parameter receives the type of event that has occurred. Currently, this event can be only , which means that this parameter is not useful. The parameter receives the connection activity that the function is about to start. Table 15-2 defines the possible connection activities. The parameter receives a RAS error code if one of the connection activities experiences failure.

Table 15-2 *Asynchronous Operating States*

Activity	State	Description
	Running	A communication port is about to be opened.
	Running	The communication port is open.
	Running	A device is about to be connected.
	Running	The device has connected successfully.
	Running	A physical link has been established.
	Running	The RAS authentication process has started.

	Running	An authentication event has occurred.
	Running	The client has requested another authentication attempt.
	Running	The server has requested a callback number.
	Running	The client has requested to change the password on the RAS account.
	Running	The protocol projection is starting. (We'll describe RAS protocol projections later.)
	Running	The link speed is being calculated.
	Running	An authentication request is being acknowledged.
	Running	The authentication process after a callback is starting.
	Running	The client has completed the authentication successfully.
	Running	The line is about to disconnect to prepare for a callback.
	Running	The client is waiting for the modem to reset before preparing for a callback.
	Running	The client is waiting for an incoming call from the server.
	Running	The protocol projection is complete.
	Running	User authentication is being started or retried. (This applies to Windows 95, Windows 98, and Windows Me only.)
	Running	The client has been called back. (This applies to Windows 95, Windows 98, and Windows Me only.)
	Running	The client is logging on to a remote network. (This applies to Windows 95, Windows 98, and Windows Me only.)
	Running	A subentry of a multilink phonebook entry has connected. The parameter of will contain an index of the subentry connected.
	Running	A subentry of a multilink phone- book entry has disconnected. The parameter of will contain an index of the subentry disconnected.
	Paused	is awaiting new user credentials.
	Paused	is awaiting a callback number from the client.
	Paused	expects the user to supply a new password.
	Paused	On Windows NT platforms, is awaiting a custom user interface to obtain user authentication information.
	Terminal	The RAS connection succeeded and is active.
	Terminal	The RAS connection failed or is inactive.

Table 15-2 shows the three operating states associated with connection activities in an asynchronous call: running, paused, and terminal. The running state indicates that the call is still in progress, and each running-state activity offers progress status information.

The paused state indicates that needs more information to establish the connection. By default, the paused state is disabled. On Windows NT–based operating systems, you can enable paused state notification by setting the option flag in the structure parameter of . When a paused state activity occurs, it indicates one of the following conditions:

- The user needs to supply new logon credentials because the authentication failed.

- The user needs to provide a new password because his or hers has expired.
- The user needs to provide a callback number.

These activities pertain to information supplied in the structure described earlier in this chapter. When a paused state activity occurs, will notify your callback function (or window procedure). If the paused state is disabled, RAS will send an error to your notification function and will fail. If enabled, the function will be in a paused state that allows your application to supply the necessary information through a structure. When is paused, you can resume by calling it again with the original call's connection handle () and notification function (), or you can simply end the paused operation by calling (described later in this chapter). If you resume the paused connection, you will have to supply the necessary user input via the structure passed to the resumed call.

> Do not resume the paused state by calling directly from a notification handler function such as . is not designed to handle this situation, so you should resume directly from you application thread.

The final state—terminal—indicates that the connection has either succeeded or failed. It can also indicate that the function closed the connection.

Now that you have a basic understanding of how you can monitor the connection of an asynchronous call, we'll demonstrate how to set up a simple program that calls asynchronously. The following code shows this procedure. You'll also find an asynchronous example on the companion CD.

```
void main(void)
{
    DWORD Ret;
    RASDIALPARAMS RasDialParams;
    HRASCONN hRasConn;

    // Fill in the RASDIALPARAMS structure with call parameters
    // as was done in the synchronous example
    ...

    if ((Ret = RasDial(NULL, NULL, &RasDialParams, 0,
        &RasDialFunc, &hRasConn)) != 0)
    {
        printf("RasDial failed with error %d\n", Ret);
        return;
    }
    // If RasDial succeeds, it will complete immediately,
    // leaving you the chance to perform other tasks while
    // RasDial is processing
    ...
}

// Callback function RasDialFunc()
void WINAPI RasDialFunc(UINT unMsg, RASCONNSTATE rasconnstate,
    DWORD dwError)
{
    char szRasString[256]; // Buffer for error string

    if (dwError)
    {
        RasGetErrorString((UINT)dwError, szRasString, 256);
        printf("Error: %d - %s\n",dwError, szRasString);
        return;
    }
    // Map each of the RasDial states and display on the
    // screen the next state that RasDial is entering

    switch (rasconnstate)
    {
        case RASCS_ConnectDevice:
            printf ("Connecting device...\n");
            break;
        case RASCS_DeviceConnected:
            printf ("Device connected.\n");
            break;

        // Add other connection activities here
        ...

        default:
            printf ("Unmonitored RAS activity.\n");
            break;
    }
}
```

RAS also features a stand-alone function named that allows your application to determine when an asynchronous RAS connection has been created or terminated. is defined as

```
DWORD RasConnectionNotification(
    HRASCONN hrasconn,
    HANDLE hEvent,
    DWORD dwFlags
);
```

The parameter is the connection handle returned from . The parameter is an event handle that your application creates using the function. The parameter can be set to a combination of several activity flags. The most useful ones are

- Notifies you that a RAS connection has been created. If the parameter is set to , the event is signaled when any RAS connection occurs.
- Notifies you that a RAS connection has been terminated. If the parameter is set to , the event is signaled when any connection ends.

Note that these flags function the same way as the connection activity flags described in Table 15-2. If any of these activities occur during your connection, your event will become signaled. Your application should use operating system wait functions, such as , to determine when the object becomes signaled.

Closing a Connection

Closing a connection established by is simple. All you have to do is call , which is defined as

```
DWORD RasHangUp(
    HRASCONN hrasconn
);
```

The parameter is a handle that is returned from . Although this function is easy to use, you have to consider how connections are managed internally in RAS. A serial connection uses a modem port, and it takes time for the port to reset internally when a connection shuts down. Therefore, you should wait until the port connection closes completely. To do this, you can call to determine when your connection is reset. is defined as

```
DWORD RasGetConnectStatus(
    HRASCONN hrasconn,
    LPRASCONNSTATUS lprasconnstatus
);
```

The parameter is a handle that is returned from . The parameter is a structure that receives the current connection status. A structure is defined as

```
typedef struct _RASCONNSTATUS
{
    DWORD        dwSize;
    RASCONNSTATE rasconnstate;
    DWORD        dwError;
    TCHAR        szDeviceType[ RAS_MaxDeviceType + 1 ];
    TCHAR        szDeviceName[ RAS_MaxDeviceName + 1 ];
} RASCONNSTATUS;
```

These fields are defined as

- Should be set to the size (in bytes) of
- Receives one of the connection activities defined in Table 15-2
- Gets a specific RAS error code if does not return 0
- Receives a string representing the type of device used on the connection
- Receives the name of the current device

We recommend that you check the state of your connection until you receive the activity status. Obviously, you might have to call several times until the connection is reset. Once the connection is reset, you can exit your application or make another connection.

Now that we have described the basics of setting up a RAS connection, we will describe how to set up phonebook entries that allow you to set up advanced communication properties for establishing a RAS connection.

Phonebook

A RAS phonebook is nothing more than a collection of structures (or phonebook entries) that contain phone numbers, data rates, user authentication information, VPN strategies, and other connection information. On Windows 95, Windows 98, Windows Me, and Windows CE systems, the phonebook is stored in the system Registry. On Windows NT systems, the phonebook is stored in files that typically have the file extension .PBK. A structure is defined as

```
typedef struct tagRASENTRY
{
    DWORD     dwSize;
    DWORD     dwfOptions;
    DWORD     dwCountryID;
    DWORD     dwCountryCode;
    TCHAR     szAreaCode[RAS_MaxAreaCode + 1];
    TCHAR     szLocalPhoneNumber[RAS_MaxPhoneNumber + 1];
    DWORD     dwAlternateOffset;
    RASIPADDR ipaddr;
    RASIPADDR ipaddrDns;
    RASIPADDR ipaddrDnsAlt;
    RASIPADDR ipaddrWins;
    RASIPADDR ipaddrWinsAlt;
    DWORD     dwFrameSize;
    DWORD     dwfNetProtocols;
    DWORD     dwFramingProtocol;
    TCHAR     szScript[MAX_PATH];
    TCHAR     szAutodialDll[MAX_PATH];
    TCHAR     szAutodialFunc[MAX_PATH];
    TCHAR     szDeviceType[RAS_MaxDeviceType + 1];
    TCHAR     szDeviceName[RAS_MaxDeviceName + 1];
    TCHAR     szX25PadType[RAS_MaxPadType + 1];
    TCHAR     szX25Address[RAS_MaxX25Address + 1];
    TCHAR     szX25Facilities[RAS_MaxFacilities + 1];
    TCHAR     szX25UserData[RAS_MaxUserData + 1];
    DWORD     dwChannels;
    DWORD     dwReserved1;
    DWORD     dwReserved2;
#if (WINVER >= 0x401)
    DWORD     dwSubEntries;
    DWORD     dwDialMode;
    DWORD     dwDialExtraPercent;
    DWORD     dwDialExtraSampleSeconds;
    DWORD     dwHangUpExtraPercent;
    DWORD     dwHangUpExtraSampleSeconds;
    DWORD     dwIdleDisconnectSeconds;
#endif
#if (WINVER >= 0x500)
    DWORD     dwType;
    DWORD     dwEncryptionType;
    DWORD     dwCustomAuthKey;
    GUID      guidId;
    TCHAR     szCustomDialDll[MAX_PATH];
    DWORD     dwVpnStrategy;
#endif
#if (WINVER >= 0x501)
                DWORD     dwfOptions2;
                DWORD     dwfOptions3;
                WCHAR     szDnsSuffix[RAS_MaxDnsSuffix];
                DWORD     dwTcpWindowSize;
                WCHAR     szPrerequisitePbk[MAX_PATH];
                WCHAR     szPrerequisiteEntry[RAS_MaxEntryName + 1];
                DWORD     dwRedialCount;
                DWORD     dwRedialPause;
#endif
} RASENTRY;
```

As you can see, many fields make up this structure and we will not describe them here. For a complete description of all the fields, consult the Platform SDK.

When you call any RAS API that takes a phonebook file as a parameter (), you can identify the path to a phonebook file. As we mentioned earlier, this parameter must be on Windows 95, Windows 98, Windows Me, and Windows CE systems because phonebook entries are stored in the system Registry. On Windows NT systems, this can be a path to a phonebook file. Typically, this phonebook file will have the extension .PBK. Also, the system default phonebook on Windows NT systems is located under %SystemRoot%\System32\Ras\Rasphone.pbk. If you specify as the phonebook, you will use the system default phonebook file.

Adding Phonebook Entries

RAS provides four functions that allow you to programmatically manage phonebook structures: , , , and . You can use the function to create a new entry or modify an existing entry. This function is defined as

```
DWORD RasSetEntryProperties(
    LPCTSTR lpszPhonebook,
    LPCTSTR lpszEntry,
    LPRASENTRY lpRasEntry,
    DWORD dwEntryInfoSize,
    LPBYTE lpbDeviceInfo,
    DWORD dwDeviceInfoSize
);
```

The parameter is a pointer to a phonebook file name. The parameter is a pointer to a string used to identify an existing or new entry. If a structure exists with this name, the properties are modified; otherwise, a new entry is created in the phonebook. The parameter is a pointer to a structure. You can place a list of null-terminated strings after the structure defining alternate phone numbers. The last string is terminated by two consecutive null characters. The parameter is the size (in bytes) of the structure in the parameter. The parameter is a pointer to a buffer that contains TAPI device configuration information. On Windows 2000 and Windows NT, this parameter is not used and should be set to . The final parameter, , represents the size (in bytes) of the buffer.

The function, defined below, can be used to retrieve the properties of an existing phonebook entry or the default values for a new phonebook entry.

```
DWORD RasGetEntryProperties(
    LPCTSTR lpszPhonebook,
    LPCTSTR lpszEntry,
    LPRASENTRY lpRasEntry,
    LPDWORD lpdwEntryInfoSize,
    LPBYTE lpbDeviceInfo,
    LPDWORD lpdwDeviceInfoSize
);
```

The parameter is a pointer to the name of a phonebook file. The parameter is a pointer to a string identifying an existing phonebook entry. If you set this parameter to , the and parameters will receive default values of a phonebook entry. Retrieving the default values is quite useful: when you need to create a new RAS phonebook entry, you can populate the and fields with correct information about your system before you call the function.

The parameter is a pointer to a buffer that your application supplies to receive a structure. As we described in our discussion of the function, this structure can be followed by an array of null-terminated strings identifying alternate phone numbers for the requested phonebook entry. Therefore, the size of your receiving buffer should be larger than a structure. If you pass a pointer, the parameter will receive the total number of bytes needed to store all the elements of a structure plus any alternate phone numbers. The parameter is a pointer to a containing the number of bytes that are in the receiving buffer your application supplies to the parameter. When this function completes, it will update to the number of bytes actually received in . We highly recommend calling this function with set to and set to 0 to obtain buffer sizing information. Once you have the appropriate size, you can call this function again and retrieve all the information without error.

The parameter is a pointer to an application-supplied buffer that receives TAPI device–specific information for this phonebook entry. If this parameter is set to , the parameter will receive the number of bytes needed to retrieve this information. If you are using Windows 2000, Windows NT, or Windows XP, should be set to . The final parameter, , is a pointer to a that should be set to the number of bytes contained in the buffer supplied to . When returns, will return the number of bytes that are returned in the buffer.

The following code sample demonstrates how an application should use and to create a new phonebook entry:

```
#include <windows.h>
#include <ras.h>
#include <raserror.h>
#include <stdio.h>

void main(void)
{
    DWORD EntryInfoSize = 0;
    DWORD DeviceInfoSize = 0;
    DWORD Ret;
    LPRASENTRY lpRasEntry;
    LPBYTE lpDeviceInfo;

    // Get buffer sizing information for a default phonebook entry
    if ((Ret = RasGetEntryProperties(NULL, "", NULL,
        &EntryInfoSize, NULL, &DeviceInfoSize)) != 0)
    {
        if (Ret != ERROR_BUFFER_TOO_SMALL)
        {
            printf("RasGetEntryProperties sizing failed "
                "with error %d\n", Ret);
            return;
        }
    }

    lpRasEntry = (LPRASENTRY) GlobalAlloc(GPTR, EntryInfoSize);

    if (DeviceInfoSize == 0)
        lpDeviceInfo = NULL;
    else
        lpDeviceInfo = (LPBYTE) GlobalAlloc(GPTR, DeviceInfoSize);

    // Get default phonebook entry
    lpRasEntry->dwSize = sizeof(RASENTRY);

    if ((Ret = RasGetEntryProperties(NULL, "", lpRasEntry,
        &EntryInfoSize, lpDeviceInfo, &DeviceInfoSize)) != 0)
    {
        printf("RasGetEntryProperties failed with error %d\n",
            Ret);
        return;
    }

    // Validate new phonebook name "Testentry"

    if ((Ret = RasValidateEntryName(NULL, "Testentry")) !=
        ERROR_SUCCESS)
    {
        printf("RasValidateEntryName failed with error %d\n",
            Ret);
        return;
    }

    // Install a new phonebook entry, "Testentry", using
    // default properties

    if ((Ret = RasSetEntryProperties(NULL, "Testentry",
        lpRasEntry, EntryInfoSize, lpDeviceInfo,
        DeviceInfoSize)) != 0)
    {
        printf("RasSetEntryProperties failed with error %d\n",
            Ret);
        return;
    }
}
```

Deleting Phonebook Entries

Deleting phonebook entries is easy. To do so, you simply call the function, which is defined as

```
DWORD RasDeleteEntry(
    LPCTSTR lpszPhonebook,
    LPCTSTR lpszEntry
);
```

The parameter is a pointer to a phonebook file name. The parameter is a string representing an existing phonebook entry. If this function succeeds, it returns ; otherwise, it returns .

Managing User Credentials

When a RAS client makes a connection using a phonebook entry through , it can save the user's security credentials and associates them with the phonebook entry. The functions , , , and allow you to man- age user security credentials associated with a phonebook entry. The and functions were introduced in Windows NT 4. (They are also available on all Windows NT systems.) These two functions supersede and . Because and are not available on Windows 95, Windows 98, Windows Me, and Windows CE systems, you can use and for this purpose on all platforms.

The function retrieves user credentials associated with a phonebook entry and is defined as

```
DWORD RasGetCredentials(
    LPCTSTR lpszPhonebook,
    LPCTSTR lpszEntry,
    LPRASCREDENTIALS lpCredentials
);
```

The parameter is a pointer to a phonebook filename. The parameter is a string representing an existing phonebook entry. The parameter, defined below, is a pointer to a structure that can receive the user name, password, and domain associated with the phonebook entry:

```
typedef struct {
    DWORD dwSize;
    DWORD dwMask;
    TCHAR szUserName[UNLEN + 1];
    TCHAR szPassword[PWLEN + 1];
    TCHAR szDomain[DNLEN + 1];
} RASCREDENTIALS, *LPRASCREDENTIALS;
```

The fields of this structure are defined as

- Specifies the size (in bytes) of a structure. You should always set this field to the structure size.
- Is a bitmask field that identifies which of the next three fields in the structure are valid by using predefined flags. The flag applies to , applies to , and applies to . There are more flags, but we will not describe them here.
- Is a null-terminated string that contains a user's logon name.
- Is a null-terminated string that contains a user's password.

- Is a null-terminated string that contains a user's logon domain.

If succeeds, it returns 0. Your application can determine which security credentials are set based on the flags set in the field of the structure.

The function is similar to except that it lets you change security credentials associated with a phonebook entry. The parameters are the same except that features an additional parameter: . is defined as

```
DWORD RasSetCredentials(
    LPCTSTR lpszPhonebook,
    LPCTSTR lpszEntry,
    LPRASCREDENTIALS lpCredentials,
    BOOL fClearCredentials
);
```

The parameter is a Boolean operator that, if set to , causes to change credentials identified in the field of the structure to an empty string (“”) value. For example, if contains the flag, the password stored is replaced with an empty string. If the function succeeds, it returns 0.

You can also use and to manage user security credentials associated with phonebook entries. is defined as

```
DWORD RasGetEntryDialParams(
    LPCTSTR lpszPhonebook,
    LPRASDIALPARAMS lprasdialparams,
    LPBOOL lpfPassword
);
```

The parameter is a pointer to a phonebook file name. The parameter is a pointer to a structure. The parameter is a Boolean flag that returns if the user's password was retrieved in the structure.

The function changes the connection information that was last set by the call on a particular phonebook entry. is defined as

```
DWORD RasSetEntryDialParams(
    LPCTSTR lpszPhonebook,
    LPRASDIALPARAMS lprasdialparams,
    BOOL fRemovePassword
);
```

The and parameters are exactly the same as the first two parameters in . The parameter is a Boolean flag that, if set to , tells to remove the password associated with the phonebook entry identified in the structure.

Connection Management

RAS has two useful functions that allow you to retrieve the properties of connections established on your system: and . can retrieve all the available active RAS connections on your system. This is useful when you need to obtain connection specific information about a RAS connection on your system using , as you will see later in this chapter. The is defined as

```
DWORD RasEnumConnections(
    LPRASCONN lprasconn,
    LPDWORD lpcb,
    LPDWORD lpcConnections
);
```

The parameter is an application buffer that will receive an array of structures. A structure is defined as

```
typedef struct _RASCONN
{
    DWORD dwSize;
    HRASCONN hrasconn;
    TCHAR szEntryName[RAS_MaxEntryName + 1];
#if (WINVER >= 0x400)
    TCHAR szDeviceType[RAS_MaxDeviceType + 1];
    TCHAR szDeviceName[RAS_MaxDeviceName + 1];
#endif
#if (WINVER >= 0x401)
    TCHAR szPhonebook[MAX_PATH];
    DWORD dwSubEntry;
#endif
#if (WINVER >= 0x500)
    GUID guidEntry;
#endif
#if (WINVER >= 0x501)
    DWORD dwSessionId;
    DWORD dwFlags;
    LUID luid;
#endif
} RASCONN;
```

The most useful field in the structure is the , which receives the connection handle that originally created. You can use this handle to retrieve more connection information from , which is described later. You need to pass to a large enough buffer to hold several structures; otherwise, this function will fail with the error . Also, the first structure in your buffer must have the field set to the byte size of a structure. The next parameter, , is a pointer to a variable that you must set to the size (in bytes) of your array. When this function returns, will contain the number of bytes required to enumerate all connections. If you don't supply a large enough buffer, you can always try again with the correct buffer size returned in . The parameter is a pointer to a variable that receives a count of the number of structures written to .

With the RAS connection handles you receive from , you can obtain network protocol–specific information that is used over an established RAS connection. This is known as . A remote access server uses projection information to represent a remote client on the network. For example, when you make a RAS connection that uses IP over a framing protocol, IP configuration information (such as an assigned IP address) is established from the RAS to your client. You can retrieve projection information for the protocols that travel over the PPP framing protocol by calling the function, which is defined as

```
DWORD RasGetProjectionInfo(
    HRASCONN hrasconn,
    RASPROJECTION rasprojection,
    LPVOID lpprojection,
    LPDWORD lpcb
);
```

The parameter is a RAS connection handle. The parameter is a enumeration type that allows you to specify a protocol to receive connection information for. The parameter receives a data structure that is associated with the enumeration type specified in . The final parameter, , is a pointer to a variable that you must set to the size of your structure. When this function completes, this variable will contain the size of the buffer needed to obtain the projection information.

The following enumeration types allow you to receive connection information:

-
-
-
-
-
-
-

To retrieve IP address information for a PPP framing protocol connection, you must specify the enumeration type. When you specify a , you will receive a structure that is defined as

```
typedef  struct  _RASPPPIP {
  DWORD    dwSize;
  DWORD    dwError;
  TCHAR    szIpAddress[ RAS_MaxIpAddress + 1 ];
#ifndef WINNT35COMPATIBLE
  TCHAR    szServerIpAddress[ RAS_MaxIpAddress + 1 ];
#endif
#if (WINVER >= 0x500)
  DWORD    dwOptions;
  DWORD    dwServerOptions;
#endif
} RASPPPIP;
```

The fields are defined as

- Should be set to the size (in bytes) of a structure
- Receives an error code from the PPP negotiation process
- Receives a string representing the client's IP address
- Receives a string representing the server's IP address
- Compression options for the local host
- Compression options for the remote host

The following code demonstrates how to call to retrieve the IP addresses assigned to a client when a PPP connection is made over RAS:

```
lpProjection = (RASPPPIP *) GlobalAlloc(GPTR, cb);
lpProjection->dwSize = sizeof(RASPPPIP);
cb = sizeof(RASPPPIP);

Ret = RasGetProjectionInfo(hRasConn, RASP_PppIp,
    lpProjection, &cb);

if (Ret != ERROR_SUCCESS)
{
    printf("RasGetProjectionInfo failed with error %d", Ret);
    return;
}
 else
{
    printf("\nRas Client IP address: %s\n",
        lpProjection->szIpAddress);
    printf("Ras Server IP address: %s\n",
        lpProjection->szServerIpAddress);
}
```

VPN

As we mentioned previously, RAS can also be used to form secure VPN connections over an IP network. You can do this as long as you have IP connectivity established on your computer either through RAS or through a network adapter. Today some computers have IP connectivity established either through Digital Subscriber Line (DSL) or cable modems and do not use RAS serial devices to connect to the Internet. If you do not have IP connectivity established, you can use *RasDial* first to establish IP connectivity over a serial device, as we have already described. Once IP connectivity is established either through RAS or through a network adapter, you can use *RasDial* to establish a secure link to your remote network over the existing IP connection.

Using the RAS client to establish a VPN connection requires building a phonebook entry that contains the remote VPN server's IP address information on the remote network instead of telephone dialing information. In a phonebook *RASENTRY* structure you must set the *szDeviceType* field to the string "*RASDT_Vpn*". Once this field is set, the *szLocalPhoneNumber* field can be used to specify either a DNS name or an IP address string of the remote VPN server instead of a phone number. After the phonebook entry is established, *RasDial* can be called using the phonebook entry to establish the VPN link.

Conclusion

This chapter presented the basics of using RAS to extend your computer's networking capability. We described how to call the *RasDial* function to communicate with remote networks. We also discussed how to maximize RAS's capabilities by creating phonebook entries.

Chapter 16

IP Helper Functions

This chapter will introduce you to API functions that allow you to query and manage IP characteristics on your computer. The functions are designed to help you programmatically achieve the functionality that is available in the following standard IP utilities:

- **IPCONFIG.EXE (or WINIPCFG.EXE in Microsoft Windows 95, Windows 98, and Windows Me)** Displays IPv4 configuration information and permits you to release and renew DHCP-assigned IPv4 addresses.
- **IPV6.EXE** A new API has been introduced that enumerates IPv6 addresses similar to the IPV6.EXE or NETSH.EXE commands. This utility will simply be entitled IPCONFIGV6.EXE.
- **NETSTAT.EXE** Displays the TCP connection table, the UDP listener table, and the IPv4 protocol statistics.
- **ROUTE.EXE** Displays and manipulates IPv4 routing tables.
- **ARP.EXE** Displays and modifies the IPv4-to-physical address translation tables that ARP uses.

The functions described in this chapter are available mainly in Windows 98, Windows Me, Windows 2000, and Windows XP. Several are also available in Windows NT 4.0 Service Pack 4 or later; however, none are available in Windows 95. We will point out platform specifics as we discuss each function. The prototypes for all of the functions described in this chapter are defined in IPHLPAPI.H. In addition, many of the data structures are defined in IPTYPES.H. When you are building your application, you must link it to the library file IPHLPAPI.LIB.

The samples provided for this chapter, which mimic the well-known system utilities, are located in the directory SAMPLES\CHAPTER16.

Note that the IP Helper APIs were developed before the availability of IPv6 on the Windows platforms. Therefore, all of the APIs return information about IPv4 only except for a single new IP Helper API *GetAdaptersAddresses*, which is discussed in the next section.

Ipconfig

The IPCONFIG.EXE utility presents two pieces of information: IPv4 configuration information and IPv4 configuration parameters specific to each network adapter installed on your machine. To retrieve IP configuration information, use the *GetNetworkParams* function, which is defined as

```
DWORD GetNetworkParams(
    PFIXED_INFO pFixedInfo,
    PULONG pOutBufLen
);
```

The *pFixedInfo* parameter receives a pointer to a buffer that receives a *FIXED_INFO* data structure your application must provide to retrieve the IPv4 configuration information. The *pOutBufLen* parameter is a pointer to a variable that specifies the size of the buffer you passed in the *pFixedInfo* parameter. If your buffer is not large enough, *GetNetworkParams* returns *ERROR_BUFFER_OVERFLOW* and sets the *pOutBufLen* parameter to the required buffer size.

The *FIXED_INFO* structure used in *GetNetworkParams* is defined as

```
typedef struct
{
    char            HostName[MAX_HOSTNAME_LEN + 4] ;
    char            DomainName[MAX_DOMAIN_NAME_LEN + 4];
    PIP_ADDR_STRING CurrentDnsServer;
    IP_ADDR_STRING  DnsServerList;
    UINT            NodeType;
    char            ScopeId[MAX_SCOPE_ID_LEN + 4];
    UINT            EnableRouting;
    UINT            EnableProxy;
    UINT            EnableDns;
} FIXED_INFO, *PFIXED_INFO;
```

The fields are defined as follows:

- ***HostName*** Represents your computer's name as recognized by DNS.
- ***DomainName*** Specifies the DNS domain your computer belongs to.
- ***CurrentDnsServer*** Contains the current DNS server's IPv4 address.
- ***DnsServerList*** Is a linked list containing the DNS servers that your machine uses.
- ***NodeType*** Specifies how the system resolves NetBIOS names over an IPv4 network. Table 16-1 contains the possible values.
- ***ScopeId*** Identifies a string value that is appended to a NetBIOS name to logically group two or more computers for NetBIOS communication over TCP/IP.
- ***EnableRouting*** Specifies whether the system will route IPv4 packets between the networks it is connected to.
- ***EnableProxy*** Specifies whether the system will act as a WINS proxy agent on a network. A WINS

proxy agent answers broadcast queries for names that it has resolved through WINS and allows a network of b-node computers to connect to servers on other subnets registered with WINS.

- ***EnableDns*** Specifies whether NetBIOS will query DNS for names that cannot be resolved by WINS, broadcast, or the *LMHOSTS* file.

Table 16-1 *Possible Node Type Values*

Value	Description
BROADCAST_NODETYPE	Known as b-node NetBIOS name resolution, in which the system uses IP broadcasting to perform NetBIOS name registration and name resolution.
PEER_TO_PEER_NODETYPE	Known as p-node NetBIOS name resolution, in which the system uses point-to-point communication with a NetBIOS name server (such as WINS) to register and resolve computer names to IP addresses.
MIXED_NODETYPE	Known as m-node (mixed node) NetBIOS name resolution, in which the system uses both the b-node and p-node techniques. The b-node method is used first; if it fails, the p-node method is used.
HYBRID_NODETYPE	Known as h-node (hybrid node) NetBIOS name resolution, in which the system uses both the b-node and p-node techniques. The p-node method is used first; if it fails, the b-node method is used next.

The *DnsServerList* field of a *FIXED_INFO* structure is an *IP_ADDR_STRING* structure that represents the beginning of a linked list of IPv4 addresses. This field is defined as

```
typedef struct _IP_ADDR_STRING
{
    struct _IP_ADDR_STRING* Next;
    IP_ADDRESS_STRING       IpAddress;
    IP_MASK_STRING          IpMask;
    DWORD                   Context;
} IP_ADDR_STRING, *PIP_ADDR_STRING;
```

The *Next* field identifies the next DNS server IPv4 address in the list. If *Next* is set to *NULL*, it indicates the end of the list. The *IpAddress* field is a string of characters that represents an IPv4 address as a dotted decimal string. The *IpMask* field is a string of characters that represents the subnet mask associated with the IPv4 address listed in *IpAddress*. The final field, *Context*, identifies the IPv4 address with a unique value on the system.

The IPCONFIG.EXE utility is also capable of retrieving IP configuration information specific to a network interface. A network interface can be a hardware Ethernet adapter or even a RAS dial-up adapter. You can retrieve adapter information by calling *GetAdaptersInfo*, which is defined as

```
 DWORD GetAdaptersInfo (
    PIP_ADAPTER_INFO pAdapterInfo,
    PULONG pOutBufLen
);
```

Use the *pAdapterInfo* parameter to pass a pointer to an application-provided buffer that receives an *ADAPTER_INFO* data structure with the adapter configuration information. The *pOutBufLen* parameter is a pointer to a variable that specifies the size of the buffer you passed in the *pAdapterInfo* parameter. If your buffer is not large enough, *GetAdaptersInfo* returns *ERROR_BUFFER_ OVERFLOW* and sets the *pOutBufLen* parameter to the required buffer size.

The *IP_ADAPTER_INFO* structure is actually a list of structures containing IPv4 configuration information specific to every network adapter available on your machine. *IP_ADAPTER_INFO* is defined as

```
typedef struct _IP_ADAPTER_INFO
{
    struct _IP_ADAPTER_INFO* Next;
    DWORD           ComboIndex;
    char            AdapterName[MAX_ADAPTER_NAME_LENGTH + 4];
    char            Description[MAX_ADAPTER_DESCRIPTION_LENGTH + 4];
    UINT            AddressLength;
    BYTE            Address[MAX_ADAPTER_ADDRESS_LENGTH];
    DWORD           Index;
    UINT            Type;
    UINT            DhcpEnabled;
    PIP_ADDR_STRING CurrentIpAddress;
    IP_ADDR_STRING  IpAddressList;
    IP_ADDR_STRING  GatewayList;
    IP_ADDR_STRING  DhcpServer;
    BOOL            HaveWins;
    IP_ADDR_STRING  PrimaryWinsServer;
    IP_ADDR_STRING  SecondaryWinsServer;
    time_t          LeaseObtained;
    time_t          LeaseExpires;
} IP_ADAPTER_INFO, *PIP_ADAPTER_INFO;
```

The fields of the structure are defined as follows:

- ***Next*** Identifies the next adapter in the buffer. A *NULL* value indicates the end of the list.
- ***ComboIndex*** Is not used and will be set to 0.
- ***AdapterName*** Identifies the name of the adapter.
- ***Description*** Is a simple description of the adapter.
- ***AddressLength*** Identifies how many bytes make up the physical address of the adapter in the *Address* field.
- ***Address*** Identifies the physical address of the adapter.
- ***Index*** Identifies a unique internal index number of the network interface that this adapter is assigned to.
- ***Type*** Specifies the type of the adapter as a numeric value. Table 16-2 defines the most common adapter types. A full listing of the supported adapter types can be found in IPIFCONS.H.

Table 16-2 *Adapter Types*

Adapter Type Value	Description

MIB_IF_TYPE_ETHERNET	Ethernet adapter
MIB_IF_TYPE_FDDI	Fiber Distributed Data Interface (FDDI) adapter
MIB_IF_TYPE_LOOPBACK	Loopback adapter
MIB_IF_TYPE_OTHER	Other type of adapter
MIB_IF_TYPE_PPP	PPP adapter
MIB_IF_TYPE_SLIP	Slip adapter
MIB_IF_TYPE_TOKENRING	Token Ring adapter

- ***DhcpEnabled*** Specifies whether DHCP is enabled on this adapter.
- ***CurrentIpAddress*** Is not used and will be set to a NULL value.
- ***IpAddressList*** Specifies a list of IPv4 addresses assigned to the adapter.
- ***GatewayList*** Specifies a list of IPv4 addresses representing the default gateway.
- ***DhcpServer*** Specifies a list with only one element representing the IPv4 address of the DHCP server used.
- ***HaveWins*** Specifies whether the adapter uses a WINS server.
- ***PrimaryWinsServer*** Specifies a list with only one element representing the IPv4 address of the primary WINS server used.
- ***SecondaryWinsServer*** Specifies a list with only one element representing the IPv4 address of the secondary WINS server used.
- ***LeaseObtained*** Identifies when the lease for the IPv4 address was obtained from a DHCP server.
- ***LeaseExpires*** Identifies when the lease on the IPv4 address obtained from DHCP expires.

The *GetAdaptersInfo* returns a great deal of information about the physical adapter and the IPv4 addresses assigned to it, but it does not return any IPv6 information. Instead, a new API *GetAdaptersAddresses* has been introduced to fill this gap as it returns address information for both IPv4 and IPv6. The API is declared as

```
DWORD WINAPI GetAdaptersAddresses(
                ULONG Family,
                DWORD Flags,
                PVOID Reserved,
                PIP_ADAPTER_ADDRESSES pAdapterAddresses,
                PULONG pOutBufLen
);
```

The *Family* parameter indicates which address family should be enumerated. The valid values are: AF_INET, AF_INET6, or AF_UNSPEC, depending on whether you want IPv4, IPv6, or all IP information. The *Flags* parameter controls the type of addresses returned. Table 16-3 lists the possible values and their meaning. Note that more than one flag can be specified by *OR*ing multiple flags together. By default, all addresses are returned. The last two parameters are the buffer that the IP information is returned in and the length of the buffer.

Table 16-3 *GetAdaptersAddresses Flag*

Flag	Description
GAA_FLAG_SKIP_UNICAST	Exclude unicast addresses.
GAA_FLAG_SKIP_ANYCAST	Exclude anycast addresses.
GAA_FLAG_SKIP_MULTICAST	Exclude multicast addresses.
GAA_FLAG_SKIP_DNS_SERVER	Exclude DNS server addresses.

The IP information is returned in the form of an *IP_ADAPTER_ADDRESSES* structure. This structure is defined as

```
typedef struct _IP_ADAPTER_ADDRESSES {
        union {
               ULONGLONG Alignment;
               struct {
                      ULONG Length;
                      DWORD IfIndex;
             }
         }
         struct _IP_ADAPTER_ADDRESSES *Next;
         PCHAR  AdapterName;
         PIP_ADAPTER_UNICAST_ADDRESS     FirstUnicastAddress;
         PIP_ADAPTER_ANYCAST_ADDRESS     FirstAnycastAddress;
         PIP_ADAPTER_MULTICAST_ADDRESS   FirstMulticastAddress;
         PIP_ADAPTER_DNS_SERVER_ADDRESS  FirstDnsServerAddress;
         PWCHAR   DnsSuffix;
         PWCHAR   Description;
         PWCHAR   FriendlyName;
         BYTE     PhysicalAddress[MAX_ADAPTER_ADDRESS_LENGTH];
         DWORD    PhysicalAddressLength;
         DWORD               Flags;
         DWORD               Mtu;
         DWORD               IfType;
         IF_OPER_STATUS      OperStatus;
         DWORD               Ipv6IfIndex;
         DWORD               ZoneIndices[16];
         PIP_ADAPTER_PREFIX   FirstPrefix;
} IP_ADAPTER_ADDRESSES, *PIP_ADAPTER_ADDRESSES;
```

- ***Length*** Specifies the length of the structure.
- ***IfIndex*** Specifies the interface index that can be cross-referenced with the interface indices that *GetAdaptersInfo* returns.
- ***Next*** Specifies the next *IP_ADAPTER_ADDRESSES* structure returned.
- ***AdapterName*** Specifies the adapter name these addresses are assigned to.
- ***FirstUnicastAddress*** Pointer to a list of *IP_ADAPER_UNICAST_ ADDRESS* structures that contain information about each unicast address assigned to this adapter.
- ***FirstAnycastAddress*** Pointer to a list of *IP_ADAPTER_ANYCAST_ ADDRESS* structures that contain information about each anycast address assigned to this adapter.
- ***FirstMulticastAddress*** Pointer to a list of *IP_ADAPTER_MULTICAST_ ADDRESS* structures that contain information about each multicast address assigned to this adapter. This is extremely useful

because it lists each multicast address joined on each physical interface.

- ***FirstDnsServerAddress*** Pointer to a list of *IP_ADAPTER_DNS_ SERVER_ADDRESS* structures that contain information about each DNS server assigned to this adapter.
- ***DnsSuffix*** Specifies the Unicode DNS suffix string associated with this adapter.
- ***Description*** Contains a Unicode string description of the adapter.
- ***FriendlyName*** Contains a Unicode string description of the adapter that is usually more easily human readable than the Description field.
- ***PhysicalAddress*** Specifies the physical address of the adapter in an array of bytes. For an Ethernet adapter, this would specify the MAC address.
- ***PhysicalAddressLength*** Specifies the number of bytes that comprise the physical address contained in the *PhysicalAddress* field.
- ***Flags*** Indicates the state of the adapter with respect to DDNS, DNS, and DHCP. Table 16-4 lists the possible flags.

Table 16-4 *IP_ADAPTERS_ADDRESSES Flags*

Flag	Description
IP_ADAPTER_DDNS_ENABLED	Dynamic DNS is enabled on this adapter.
IP_ADAPTER_REGISTER_ADAPTER_SUFFIX	The DNS suffix for this adapter is registered.
IP_ADAPTER_DHCP_ENABLED	DHCP is enabled on this adapter.
IP_ADAPTER_RECEIVE_ONLY	The interface is capable of receiving data only.
IP_ADAPTER_NO_MULTICAST	The interface is not capable of receiving multicast data.
IP_ADAPTER_IPV6_OTHER_STATEFUL_CONFIG	Indicates the “O” bit in the most recently received IPv6 router advertisement was set. This indicates the presence of stateful configuration information such as DHCPv6.

- ***Mtu*** Specifies the maximum transmission unit support on this adapter.
- ***IfType*** Specifies the type of adapter; see Table 16-2 for the possible values.
- ***OperStatus*** Specifies the operational status of the adapter. For more information about this field, see RFC 2863.
- ***Ipv6IfIndex*** Specifies the interface index of the adapter for the IPv6 addresses assigned to this interface. Note that this field should be used with IPv6 addresses and not the IfIndex field.
- ***ZoneIndices*** Specifies the scope-IDs for the 16 different scope levels. The most popular scope levels are defined in the enumerated type SCOPE_LEVEL. Consult RFC2373 for more information.
- ***FirstPrefix*** A linked list of *IP_ADAPTER_PREFIX* structures which contain the subnet prefixes which are on-link for this interface.

The last piece of the *IP_ADAPTERS_ADDRESSES* structure is the adapter structures. These four structures are very similar and for the most part contain the same kind of information. We'll discuss only the unicast version of the structure because the remaining address structures can be inferred from this one. The unicast structure is defined as

```
typedef struct _IP_ADAPTER_UNICAST_ADDRESS {
        union {
              ULONGLONG Alignment;
              struct {
                     ULONG Length;
                     DWORD Flags;
              };
         };
         struct _IP_ADAPTER_UNICAST_ADDRESS *Next;
         SOCKET_ADDRESS      Address;
         IP_PREFIX_ORIGIN    PrefixOrigin;
         IP_SUFFIX_ORIGIN    SuffixOrigin;
         IP_DAD_STATE        DadState;
         ULONG               ValidLifetime;
         ULONG               PreferredLifeTime;
         ULONG               LeaseLifeTime;
} IP_ADAPTER_UNICAST_ADDRESS, *PIP_ADAPTER_UNICAST_ADDRESS;
```

- ***Length*** Specifies the length of this structure in bytes.
- ***Flags*** Specifies the type of address. Table 16-5 contains the possible flag values.

Table 16-5 *Per Address Flags*

Per Address Flags	Description
IP_ADAPTER_ADDRESS_DNS_ELIGIBLE	Address can be registered with DNS (such as DHCP or RA assigned).
IP_ADAPTER_ADDRESS_TRANSIENT	Address is not a permanent address (such as IPv6 privacy address).

- ***Next*** Specifies the next address structure in the link list.
- ***PrefixOrigin*** Specifies how the network prefix was obtained. Table 16-6 lists the possible values, which are an enumerated type defined in IPTYPES.H.

Table 16-6 *Prefix Origin Values*

Prefix Origin Value	Description
IpPrefixOriginOther	Prefix obtained from source other than those listed in this table.
IpPrefixOriginManual	Prefix was manually configured—for example, assigning a static IPv4 address.
IpPrefixOriginWellKnown	Prefix is a well-known address—for example, the loopback address.
IpPrefixOriginDhcp	Prefix is assigned by DHCP.
IpPrefixOriginRouterAdvertisement	Prefix is assigned by a router advertisement—for example, an IPv6 site local or global address.

- ***SuffixOrigin*** Specifies how the host portion of the address was obtained. Table 16-7 lists these values, which are also an enumerated type.

Table 16-7 *Suffix Origin Values*

Suffix Origin Value	Description
IpSuffixOriginOther	Suffix obtained from a source other than those listed in this table.
IpSuffixOriginManual	Suffix was configured manually—for example, a statically assigned IP address.
IpSuffixOriginWellKnown	Suffix is a well-known address—for example, the loopback adapter.
IpSuffixOriginDhcp	Suffix is assigned by DHCP.
IpSuffixOriginLinkLayerAddress	Suffix is obtained from the lower network layer. For example, IPv6 link local addresses.
IpSuffixOriginRandom	Suffix is a randomly assigned value. For example, IPv6 privacy addresses.

- ***DadState*** Indicates the state of the address. Duplicate address detection (DAD) is the process by which an address is validated. Table 16-8 lists the values and their meanings.

Table 16-8 *Address States*

Address State	Description
IpDadStateInvalid	Address is in the process of being deleted.
IpDadStateTentative	Duplicate address detection is in progress.
IpDadStateDuplicate	A duplicate address has been detected.
IpDadStateDeprecated	Address is no longer preferred for new connections.
IpDatStatePreferred	Address is the preferred address.

- ***ValidLifeTime*** Specifies in seconds how long the address is valid. A value of 0xFFFFFFFF indicates the address does not expire.
- ***PreferredLifetime*** Specifies in seconds how long the address is the preferred address. After the preferred lifetime expires, the address goes into the deprecated state. A value of 0xFFFFFFFF indicates the address does not expire.
- ***LeaseLifeTime*** Specifies in seconds how long the DHCP lease is valid. A value of 0xFFFFFFFF indicates the lease does not expire.

Releasing and Renewing IPv4 Addresses

The IPCONFIG.EXE utility also features the ability to release and renew IPv4 addresses obtained from the DHCP server by specifying the /release and /renew command line parameters. If you want to programmatically release an IPv4 address, you can call the *IPReleaseAddress* function, which is defined as

```
DWORD IpReleaseAddress (
    PIP_ADAPTER_INDEX_MAP AdapterInfo
);
```

If you want to renew an IP address, you can call the *IPRenewAddress* function, which is defined as

```
DWORD IpRenewAddress (
    PIP_ADAPTER_INDEX_MAP AdapterInfo
);
```

Each of these two functions features an *AdapterInfo* parameter that is an *IP_ADAPTER_INDEX_MAP* structure, which identifies the adapter to release or renew the address for. The *IP_ADAPTER_INDEX_MAP* structure is defined as

```
typedef struct _IP_ADAPTER_INDEX_MAP
{
    ULONG Index;
    WCHAR Name[MAX_ADAPTER_NAME];
}IP_ADAPTER_INDEX_MAP, *PIP_ADAPTER_INDEX_MAP;
```

The fields of this structure are defined as follows:

- ***Index*** Identifies the internal index of the network interface that the adapter is assigned to.
- ***Name*** Identifies the name of the adapter.

You can retrieve the *IP_ADAPTER_INDEX_MAP* structure for a particular adapter by calling the *GetInterfaceInfo* function, which is defined as

```
DWORD GetInterfaceInfo (
    IN PIP_INTERFACE_INFO pIfTable,
    OUT PULONG dwOutBufLen
);
```

The *pIfTable* parameter is a pointer to an *IP_INTERFACE_INFO* application buffer that will receive interface information. The *dwOutBufLen* parameter is a pointer to a variable that specifies the size of the buffer you passed in the *pIfTable* parameter. If the buffer is not large enough to hold the interface information, *GetInterfaceInfo* returns the error *ERROR_INSUFFICIENT_BUFFER* and sets the *dwOutBufLen* parameter to the required buffer size.

The *IP_INTERFACE_INFO* structure is defined as

```
typedef struct _IP_INTERFACE_INFO
{
    LONG                 NumAdapters;
    IP_ADAPTER_INDEX_MAP Adapter[1];
} IP_INTERFACE_INFO,*PIP_INTERFACE_INFO;
```

Its fields are defined as follows:

- ***NumAdapters*** Identifies the number of adapters in the Adapter field.
- ***Adapter*** Is an array of *IP_ADAPTER_INDEX_MAP* structures, defined on the preceding page.

Once you have obtained the *IP_ADAPTER_INDEX_MAP* structure for a particular adapter, you can release or renew the DHCP-assigned IPv4 address using the *IPReleaseAddress* and *IPRenewAddress* functions we just described.

Changing IPv4 Addresses

The IPCONFIG.EXE utility does not allow you to change an IP address for a network adapter (except in the case of DHCP). However, two functions will allow you to add or delete an IP address for a particular adapter: the *AddIpAddress* and *DeleteIpAddress* IP Helper functions. These require you to understand adapter index numbers and IP context numbers. In Windows, every network adapter has a unique index ID (which we described earlier), and every IP address has a unique context ID. Adapter index IDs and IP context numbers can be retrieved using *GetAdaptersInfo*. The *AddIpAddress* function is defined as

```
DWORD AddIPAddress (
    IPAddr Address,
    IPMask IpMask,
    DWORD IfIndex,
    PULONG NTEContext,
    PULONG NTEInstance
);
```

The *Address* parameter specifies the IPv4 address to add as an unsigned long value. The *IpMask* parameter specifies the subnet mask for the IPv4 address as an unsigned long value. The *IfIndex* parameter specifies the adapter index to add the address to. The *NTEContext* parameter receives the context value associated with the IPv4 address added. The *NTEInstance* parameter receives an instance value associated with an IPv4 address.

If you want to programmatically delete an IPv4 address for an adapter, you can call *DeleteIpAddress*, which is defined as

```
 DWORD DeleteIPAddress (
    ULONG NTEContext
);
```

The *NTEContext* parameter identifies a context value associated with an IPv4 address. This value can be obtained from *GetAdaptersInfo*, which we described earlier in the chapter.

Note that IPv4 addresses added via the *AddIpAddress* function are persistent only until reboot.

Netstat

The NETSTAT.EXE utility displays the TCP connection table, the UDP listener table, and the IPv4 protocol statistics on your computer. The functions used to retrieve this information work with Windows NT 4.0 (Service Pack 4 and later), Windows 98, and Windows Me.

Retrieving the TCP Connection Table

The *GetTcpTable* function retrieves the TCP connection table. This is the same information you see when you execute NETSTAT.EXE with the -p tcp -a options. *GetTcpTable* is defined as

```
DWORD GetTcpTable(
    PMIB_TCPTABLE pTcpTable,
    PDWORD pdwSize,
    BOOL bOrder
);
```

The *pTcpTable* parameter is a pointer to an *MIB_TCPTABLE* application buffer that will receive the TCP connection information. The *pdwSize* parameter is a pointer to a variable that specifies the size of the buffer you passed in the *pTcpTable* parameter. If the buffer is not large enough to hold the TCP information, the function sets this parameter to the required buffer size. The *bOrder* parameter specifies whether the returned information should be sorted.

The *MIB_TCPTABLE* structure returned from *GetTcpTable* is defined as

```
typedef struct _MIB_TCPTABLE
{
    DWORD dwNumEntries;
    MIB_TCPROW table[ANY_SIZE];
} MIB_TCPTABLE, *PMIB_TCPTABLE;
```

The fields of this structure are defined as follows:

- ***dwNumEntries*** Specifies how many entries are in the *table* field.
- ***table*** Is a pointer to an array of *MIB_TCPROW* structures that contain TCP connection information.

The *MIB_TCPROW* structure contains the IPv4 address pair that comprises a TCP connection. This structure is defined as

```
typedef struct _MIB_TCPROW
{
    DWORD dwState;
    DWORD dwLocalAddr;
    DWORD dwLocalPort;
    DWORD dwRemoteAddr;
    DWORD dwRemotePort;
} MIB_TCPROW, *PMIB_TCPROW;
```

Its fields are defined as follows:

- ***dwState*** Specifies the state of the TCP connection, as defined in Table 16-9. See Chapter 1 for information about TCP states.

Table 16-9 *TCP Connection States*

Connection State	RFC 793 Description
MIB_TCP_STATE_CLOSED	Known as the "CLOSED" state
MIB_TCP_STATE_CLOSING	Known as the "CLOSING" state
MIB_TCP_STATE_CLOSE_WAIT	Known as the "CLOSE WAIT" state
MIB_TCP_STATE_DELETE_TCB	Known as the "DELETE" state
MIB_TCP_STATE_ESTAB	Known as the "ESTABLISHED" state
MIB_TCP_STATE_FIN_WAIT1	Known as the "FIN WAIT1" state
MIB_TCP_STATE_FIN_WAIT2	Known as the "FIN WAIT2" state
MIB_TCP_STATE_LAST_ACK	Known as the "LAST ACK" state
MIB_TCP_STATE_LISTEN	Known as the "LISTENING" state
MIB_TCP_STATE_SYN_RCVD	Known as the "SYN RCVD" state
MIB_TCP_STATE_SYN_SENT	Known as the "SYN SENT" state
MIB_TCP_STATE_TIME_WAIT	Known as the "TIME WAIT" state

- ***dwLocalAddr*** Specifies a local IPv4 address for the connection.
- ***dwLocalPort*** Specifies a local port for the connection.
- ***dwRemoteAddr*** Specifies the remote IPv4 address for the connection.
- ***dwRemotePort*** Specifies the remote port for the connection.

Retrieving the UDP Listener Table

The *GetUdpTable* function retrieves the UDP listener table. This is the same information you see if you execute NETSTAT.EXE with the -p udp -a options. *GetUdpTable* is defined as

```
DWORD GetUdpTable(
    PMIB_UDPTABLE pUdpTable,
    PDWORD pdwSize,
    BOOL bOrder
);
```

The *pUdpTable* parameter is a pointer to an *MIB_UDPTABLE* application buffer that will receive the UDP listener information. The *pdwSize* parameter is a pointer to a variable that specifies the size of the buffer you passed in the *pUdpTable* parameter. If the buffer is not large enough to hold the UDP information, the function sets this parameter to the required buffer size. The *bOrder* parameter specifies whether the returned information should be sorted.

The *MIB_UDPTABLE* structure returned from *GetUdpTable* is defined as

```
typedef struct _MIB_UDPTABLE
```

```
{
    DWORD dwNumEntries;
    MIB_UDPROW table[ANY_SIZE];
} MIB_UDPTABLE, * PMIB_UDPTABLE;
```

The fields of this structure are defined as follows:

- ***dwNumEntries*** Specifies how many entries are in the *table* field.
- ***table*** Is a pointer to an array of *MIB_UDPROW* structures that contain UDP listener information.

The *MIB_UDPROW* structure contains the IPv4 address in which UDP is listening for datagrams. This structure is defined as

```
typedef struct _MIB_UDPROW
{
    DWORD dwLocalAddr;
    DWORD dwLocalPort;
} MIB_UDPROW, * PMIB_UDPROW;
```

Its fields are defined as follows:

- ***dwLocalAddr*** Specifies the local IPv4 address.
- ***dwLocalPort*** Specifies the local port.

Retrieving IPv4 Protocol Statistics

Four functions are available for receiving IPv4 statistics: *GetIpStatistics*, *GetIcmpStatistics*, *GetTcpStatistics*, and *GetUdpStatistics*. These functions produce the same information that is returned from NETSTAT.EXE when you call it with the -s parameter. The first statistics function, *GetIpStatistics*, retrieves the IPv4 statistics for the current computer and is defined as

```
DWORD GetIpStatistics(
    PMIB_IPSTATS pStats
);
```

The *pStats* parameter is a pointer to an *MIB_IPSTATS* structure that receives the current IPv4 statistics for your computer. The *MIB_IPSTATS* structure is defined as

```
typedef struct _MIB_IPSTATS
{
    DWORD dwForwarding;
    DWORD dwDefaultTTL;
    DWORD dwInReceives;
    DWORD dwInHdrErrors;
    DWORD dwInAddrErrors;
    DWORD dwForwDatagrams;
    DWORD dwInUnknownProtos;
    DWORD dwInDiscards;
    DWORD dwInDelivers;
    DWORD dwOutRequests;
    DWORD dwRoutingDiscards;
    DWORD dwOutDiscards;
    DWORD dwOutNoRoutes;
    DWORD dwReasmTimeout;
    DWORD dwReasmReqds;
    DWORD dwReasmOks;
    DWORD dwReasmFails;
    DWORD dwFragOks;
    DWORD dwFragFails;
    DWORD dwFragCreates;
    DWORD dwNumIf;
    DWORD dwNumAddr;
    DWORD dwNumRoutes;
} MIB_IPSTATS, *PMIB_IPSTATS;
```

The fields of this structure are defined as follows:

- ***dwForwarding*** Specifies whether IPv4 forwarding is enabled or disabled on your computer.
- ***dwDefaultTTL*** Specifies the initial TTL value for datagrams originating on your computer.
- ***dwInReceives*** Specifies the number of datagrams received.
- ***dwInHdrErrors*** Specifies the number of datagrams received with bad headers.
- ***dwInAddrErrors*** Specifies the number of datagrams received with bad addresses.
- ***dwForwDatagrams*** Specifies the number of datagrams forwarded.
- ***dwInUnknownProtos*** Specifies the number of datagrams received with an unknown protocol.
- ***dwInDiscards*** Specifies the number of datagrams received that were discarded.
- ***dwInDelivers*** Specifies the number of datagrams received that were delivered.
- ***dwOutRequests*** Specifies the number of datagrams that IPv4 has requested to transmit.
- ***dwRoutingDiscards*** Specifies the number of outgoing datagrams discarded.
- ***dwOutDiscards*** Specifies the number of transmitted datagrams discarded.
- ***dwOutNoRoutes*** Specifies the number of datagrams that did not have a routing destination.
- ***dwReasmTimeout*** Specifies the maximum amount of time for a fragmented datagram to arrive.
- ***dwReasmReqds*** Specifies the number of datagrams that require assembly.
- ***dwReasmOks*** Specifies the number of datagrams that were successfully reassembled.
- ***dwFragFails*** Specifies the number of datagrams that could not be fragmented.
- ***dwFragCreates*** Specifies the number of datagrams that were fragmented.
- ***dwNumIf*** Specifies the number of IPv4 interfaces available on your computer.
- ***dwNumAddr*** Specifies the number of IPv4 addresses identified on your computer.
- ***dwNumRoutes*** Specifies the number of routes available in the routing table.

The second statistics function, *GetIcmpStatistics*, retrieves ICMP statistics and is defined as

```
DWORD GetIcmpStatistics(
   PMIB_ICMP pStats
);
```

The *pStats* parameter is a pointer to an *MIB_ICMP* structure that receives the current ICMP statistics for your computer. The *MIB_ICMP* structure is defined as

```
typedef struct _MIB_ICMP
{
    MIBICMPINFO stats;
} MIB_ICMP,*PMIB_ICMP;
```

As you can see, *MIB_ICMP* is a structure containing a *MIBICMPINFO* structure that is defined as

```
typedef struct _MIBICMPINFO
{
    MIBICMPSTATS icmpInStats;
    MIBICMPSTATS icmpOutStats;
} MIBICMPINFO;
```

The *MIBICMPINFO* structure receives incoming or outgoing ICMP information through an *MIBICMPSTATS* structure. The *icmpInStats* parameter receives incoming data and *icmpOutStats* receives outgoing data. The *MIBICMPSTATS* structure is defined as

```
typedef struct _MIBICMPSTATS
{
    DWORD dwMsgs;
    DWORD dwErrors;
    DWORD dwDestUnreachs;
    DWORD dwTimeExcds;
    DWORD dwParmProbs;
    DWORD dwSrcQuenchs;
    DWORD dwRedirects;
    DWORD dwEchos;
    DWORD dwEchoReps;
    DWORD dwTimestamps;
    DWORD dwTimestampReps;
    DWORD dwAddrMasks;
    DWORD dwAddrMaskReps;
} MIBICMPSTATS;
```

The fields of this structure are defined as follows:

- ***dwMsgs*** Specifies the number of messages sent or received.
- ***dwErrors*** Specifies the number of errors sent or received.
- ***dwDestUnreachs*** Specifies the number of “destination unreachable” messages sent or received.
- ***dwTimeExcds*** Specifies the number of TTL-exceeded messages sent or received.
- ***dwParmProbs*** Specifies the number of messages sent or received that indicate a datagram contains bad IPv4 information.
- ***dwSrcQuenchs*** Specifies the number of source quench messages sent or received.

- ***dwRedirects*** Specifies the number of redirection messages sent or received.
- ***dwEchos*** Specifies the number of ICMP echo requests sent or received.
- ***dwEchoReps*** Specifies the number of ICMP echo replies sent or received.
- ***dwTimestamps*** Specifies the number of timestamp requests sent or received.
- ***dwTimestampReps*** Specifies the number of timestamp replies sent or received.
- ***dwAddrMasks*** Specifies the number of address masks sent or received.
- ***dwAddrMaskReps*** Specifies the number of address mask replies sent or received.

The third statistics function, *GetTcpStatistics*, retrieves TCP statistics on your computer and is defined as

```
DWORD GetTcpStatistics(
    PMIB_TCPSTATS pStats
);
```

The *pStats* parameter is a pointer to an *MIB_TCPSTATS* structure that receives the current IP statistics for your computer. The *MIB_TCPSTATS* structure is defined as

```
typedef struct _MIB_TCPSTATS
{
    DWORD dwRtoAlgorithm;
    DWORD dwRtoMin;
    DWORD dwRtoMax;
    DWORD dwMaxConn;
    DWORD dwActiveOpens;
    DWORD dwPassiveOpens;
    DWORD dwAttemptFails;
    DWORD dwEstabResets;
    DWORD dwCurrEstab;
    DWORD dwInSegs;
    DWORD dwOutSegs;
    DWORD dwRetransSegs;
    DWORD dwInErrs;
    DWORD dwOutRsts;
    DWORD dwNumConns;
} MIB_TCPSTATS, *PMIB_TCPSTATS;
```

The fields of this structure are defined as follows:

- ***dwRtoAlgorithm*** Specifies which retransmission algorithm is being used. The valid values are *MIB_TCP_RTO_CONSTANT*, *MIB_TCP_ RTO_RSRE*, *MIB_TCP_RTO_VANJ*, and *MIB_TCP_RTO_OTHER*, which is for other types.
- ***dwRtoMin*** Specifies the minimum retransmission timeout in milliseconds.
- ***dwRtoMax*** Specifies the maximum retransmission timeout in milliseconds.
- ***dwMaxConn*** Specifies the maximum number of connections allowed.
- ***dwActiveOpens*** Specifies how many times the machine is initiating a connection with a server.

- ***dwPassiveOpens*** Specifies how many times the machine is listening for a connection from a client.
- ***dwAttemptFails*** Specifies how many connection attempts have failed.
- ***dwEstabResets*** Specifies the number of established connections that have been reset.
- ***dwCurrEstab*** Specifies the number of connections that are currently established.
- ***dwInSegs*** Specifies the number of segments received.
- ***dwOutSegs*** Specifies the number of segments transmitted (excluding segments that have been retransmitted).
- ***dwRetransSegs*** Specifies the number of segments retransmitted.
- ***dwInErrs*** Specifies the number of errors received.
- ***dwOutRsts*** Specifies the number of segments transmitted with the reset flag set.
- ***dwNumConns*** Specifies the total number of connections.

The last statistics function, *GetUdpStatistics*, retrieves UDP statistics on your computer and is defined as

```
DWORD GetUdpStatistics(
    PMIB_UDPSTATS pStats
);
```

The *pStats* parameter is a pointer to an *MIB_UDPSTATS* structure that receives the current IPv4 statistics for your computer. The *MIB_UDPSTATS* structure is defined as

```
typedef struct _MIB_UDPSTATS
{
    DWORD dwInDatagrams;
    DWORD dwNoPorts;
    DWORD dwInErrors;
    DWORD dwOutDatagrams;
    DWORD dwNumAddrs;
} MIB_UDPSTATS,*PMIB_UDPSTATS;
```

This structure's fields are defined as follows:

- ***dwInDatagrams*** Specifies the number of datagrams received.
- ***dwNoPorts*** Specifies the number of datagrams discarded because the port number was bad.
- ***dwInErrors*** Specifies the number of erroneous datagrams received (excluding the datagrams counted in *dwNoPorts*).
- ***dwOutDatagrams*** Specifies the number of datagrams transmitted.
- ***dwNumAddrs*** Specifies the total number of UDP entries in the listener table.

Route

The ROUTE.EXE command allows you to print and modify the routing table. The routing table determines which IPv4 interface a connection request or a datagram occurs on. The IP Helper library offers several functions for manipulating the routing table. All of the functions related to routing are available in Windows 98, Windows Me, and Windows NT 4.0 (Service Pack 4 or later).

First, let's discuss the ROUTE.EXE command's capabilities. Its most basic function is to print the routing table. A route consists of a destination address, a netmask, a gateway, a local IP interface, and a metric. You also have the ability to add and delete a route. To add a route, you must specify all the parameters we just described. To delete a route, you must specify the destination address only. In this section, we'll look at the IP Helper functions that print the routing table. Then we'll discuss adding and deleting a route.

Getting the Routing Table

The most basic action that ROUTE.EXE performs is printing the table. This is accomplished with the *GetIpForwardTable* function, which is defined as

```
DWORD GetIpForwardTable (
    PMIB_IPFORWARDTABLE pIpForwardTable,
    PULONG pdwSize,
    BOOL bOrder
);
```

The first parameter, *pIpForwardTable*, contains the routing table information upon return. When you call the function, this parameter should point to a buffer of sufficient size. If you call the function with *pIpForwardTable* equal to *NULL* (or if the buffer size is insufficient to begin with), the *pdwSize* parameter returns the length of the buffer needed for the call to complete successfully. The last parameter, *bOrder*, indicates whether the results should be sorted upon return. The default sorting order is

1. Destination address
2. Protocol that generated the route
3. Multipath routing policy
4. Next-hop address

The routing information is returned in the form of the *MIB_IPFORWARDTABLE* structure, which is defined as

```
typedef struct _MIB_IPFORWARDTABLE
{
    DWORD            dwNumEntries;
    MIB_IPFORWARDROW table[ANY_SIZE];
} MIB_IPFORWARDTABLE, *PMIB_IPFORWARDTABLE;
```

This structure is a wrapper for an array of *MIB_IPFORWARDROW* structures. The *dwNumEntries* field indicates the number of these structures present in the array. The *MIB_IPFORWARDROW* structure is defined as

```
 typedef struct _MIB_IPFORWARDROW
{
    DWORD dwForwardDest;
    DWORD dwForwardMask;
    DWORD dwForwardPolicy;
    DWORD dwForwardNextHop;
    DWORD dwForwardIfIndex;
    DWORD dwForwardType;
    DWORD dwForwardProto;
    DWORD dwForwardAge;
    DWORD dwForwardNextHopAS;
    DWORD dwForwardMetric1;
    DWORD dwForwardMetric2;
    DWORD dwForwardMetric3;
    DWORD dwForwardMetric4;
    DWORD dwForwardMetric5;
} MIB_IPFORWARDROW, *PMIB_IPFORWARDROW;
```

The fields of this structure are defined as follows:

- ***dwForwardDest*** Is the IPv4 address of the destination host.
- ***dwForwardMask*** Is the subnet mask for the destination host.
- ***dwForwardPolicy*** Specifies a set of conditions that would cause the selection of a multipath route. Usually these conditions are in the form of IP TOS. For more information about TOS, see Chapter 7 and the *IP_TOS* option. For more information about multipath routing, see RFC 1354.
- ***dwForwardNextHop*** Is the IPv4 address for the next hop in the route.
- ***dwForwardIfIndex*** Indicates the index of the interface for this route.
- ***dwForwardType*** Indicates the route type as defined in RFC 1354. Table 16-10 lists the possible values and meanings for this field.
- ***dwForwardProto*** Is the protocol that generated the route. Table 16-11 lists the possible values for this field. The values for IPX protocols are defined in ROUTPROT.H, and the IP entries are included in IPRTRMIB.H.
- ***dwForwardAge*** Indicates the age of the route in seconds.
- ***dwForwardNextHopAS*** Is the autonomous system number of the next hop.
- ***dwForwardMetric1*** Is a routing protocol–specific metric value. For more information, see RFC 1354. The field contains the route metric value that you normally see when executing the ROUTE.EXE print command. For this and the following four fields, if the entry is unused, the value is *MIB_IPROUTE_METRIC_UNUSED* (-1).
- ***dwForwardMetric2*** Is a routing protocol–specific metric value. For more information, see RFC 1354.
- ***dwForwardMetric3*** Is a routing protocol–specific metric value. For more information, see RFC

1354.

- ***dwForwardMetric4*** Is a routing protocol–specific metric value. For more information, see RFC 1354.
- ***dwForwardMetric5*** Is a routing protocol–specific metric value. For more information, see RFC 1354.

Table 16-10 *Possible Route Types for a Routing Table Entry*

Forward Type	Description
MIB_IPROUTE_TYPE_INDIRECT	Next hop is not the final destination (remote route)
MIB_IPROUTE_TYPE_DIRECT	Next hop is the final destination (local route)
MIB_IPROUTE_TYPE_INVALID	Route is invalid
MIB_IPROUTE_TYPE_OTHER	Other route

Table 16-11 *Forward Protocols*

Protocol Identifier	Description
MIB_IPPROTO_OTHER	Protocol not listed
MIB_IPPROTO_LOCAL	Route generated by the stack
MIB_IPPROTO_NETMGMT	Route added by ROUTE.EXE utility or SNMP
MIB_IPPROTO_ICMP	Route from ICMP redirects
MIB_IPPROTO_EGP	Exterior Gateway Protocol
MIB_IPPROTO_GGP	Gateway Gateway Protocol
MIB_IPPROTO_HELLO	HELLO routing protocol
MIB_IPPROTO_RIP	Routing Information Protocol
MIB_IPPROTO_IS_IS	IP Intermediate System to Intermediate System Protocol
MIB_IPPROTO_ES_IS	IP End System to Intermediate System Protocol
MIB_IPPROTO_CISCO	IP Cisco protocol
MIB_IPPROTO_BBN	BBN protocol
MIB_IPPROTO_OSPF	Open Shortest Path First routing protocol
MIB_IPPROTO_BGP	Border Gateway Protocol
MIB_IPPROTO_NT_AUTOSTATIC	Routes that were originally added by a routing protocol but are not static
MIB_IPPROTO_NT_STATIC	Routes added by the routing user interface or the ROUTEMON.EXE utility
MIB_IPPROTO_STATIC_NON_DOD	Identical to *PROTO_IP_NT_STATIC* except that these routes will not cause Dial on Demand (DOD)
IPX_PROTOCOL_RIP	Routing Information Protocol for IPX
IPX_PROTOCOL_SAP	Service Advertisement Protocol
IPX_PROTOCOL_NLSP	Netware Link Services Protocol

Adding a Route

The next function of the route command is adding a route. Remember that to add a route, the destination IPv4 address, netmask, gateway, local IPv4 interface, and metric must be specified. When adding a route, you should verify that the given local IPv4 interface is valid. In addition, when adding a route you will need to refer to the local IPv4 interface on which the route is based by its internal index value. You can obtain this information by calling the *GetIpAddrTable* function, which is defined as

```
DWORD GetIpAddrTable (
    PMIB_IPADDRTABLE pIpAddrTable,
    PULONG pdwSize,
    BOOL bOrder
);
```

The first parameter, *pIpAddrTable*, is a buffer of sufficient size that will return an *MIB_IPADDRTABLE* structure. The *pdwSize* parameter is the size of the buffer passed as the first parameter. The last parameter, *bOrder*, specifies whether to return the local IPv4 interfaces by ascending IP addresses. To find out the required buffer size, you can pass *NULL* for *pIpAddrTable*. Upon return, *pdwSize* will indicate the required buffer size. The *MIB_IPADDRTABLE* structure is defined as

```
 typedef struct _MIB_IPADDRTABLE
{
    DWORD dwNumEntries
    MIB_IPADDRROW table[ANY_SIZE];
} MIB_IPADDRTABLE, *PMIB_IPADDRTABLE;
```

This structure is a wrapper for an array of *MIB_IPADDRROW* structures. The *dwNumEntries* field indicates how many *MIB_IPADDRROW* entries are present in the *table* field array. The *MIB_IPADDRROW* structure is defined as

```
typedef struct _MIB_IPADDRROW
{
    DWORD    dwAddr;
    DWORD    dwIndex;
    DWORD    dwMask;
    DWORD    dwBCastAddr;
    DWORD    dwReasmSize;
    unsigned short unused1;
    unsigned short unused2;
} MIB_IPADDRROW, *PMIB_IPADDRROW;
```

The fields of this structure are defined as follows:

- ***dwAddr*** Is the IPv4 address for a given interface.
- ***dwIndex*** Is the index of the interface associated with the IPv4 address.
- ***dwMask*** Is the subnet mask for the IPv4 address.
- ***dwBCastAddr*** Is the broadcast address.
- ***dwReasmSize*** Is the maximum reassembly size for datagrams received.

- ***unused1*** and *unused2* Are not currently used.

Using GetIpAddrTable, you can verify that the local IPv4 interface for the given route is valid. The function for adding the route to the routing table is *SetIpForwardEntry*, which is defined as

```
DWORD SetIpForwardEntry (
    PMIB_IPFORWARDROW pRoute
);
```

The only parameter is *pRoute*, which is a pointer to an *MIB_IPFORWARDROW* structure. This structure defines the elements needed to establish a new route. We have already discussed this structure and its member fields. To add a route, the values must be specified for the fields *dwForwardIfIndex*, *dwForwardDest*, *dwForwardMask*, *dwForwardNextHop*, and *dwForwardPolicy*.

Deleting a Route

The last action for the route utility is deleting a route, which is the easiest command to implement. When invoking the route command to delete a route, you must specify the destination address to delete. Then you can search for the *MIB_IPFORWARDROW* structure returned from *GetIpForwardTable* that corresponds to the destination address. The *MIB_IPFORWARDROW* structure can then be passed to the *DeleteIpForwardEntry* function to remove the given entry. This function is defined as

```
DWORD DeleteIpForwardEntry (
    PMIB_IPFORWARDROW pRoute
);
```

Alternatively, you can specify the fields of *pRoute* yourself. The fields that are required to remove a route are *dwForwardIfIndex*, *dwForwardDest*, *dwForwardMask*, *dwForwardNextHop*, and *dwForwardPolicy*.

ARP

The ARP.EXE utility is used to view and manipulate the ARP cache. The Platform SDK sample that emulates ARP.EXE by using the IP Helper functions is named IPARP.EXE. ARP (which, as you'll recall, stands for address resolution protocol) is responsible for resolving an IPv4 address to a physical MAC address. Machines cache this information for performance reasons, and it is possible to access it through the ARP.EXE utility. Using this utility, you can display the ARP table with the -a option, delete an entry with the -d option, or add an entry with the -s option. In the next section, we will describe how to print the ARP cache, add an entry to the ARP table, and delete ARP entries.

All of the IP Helper functions discussed in this section are available in Windows 98, Windows Me, and Windows NT 4.0 (Service Pack 4 or later).

The simplest function is obtaining the ARP table. The IP Helper function that obtains this table is *GetIpNetTable*, which is defined as

```
DWORD GetIpNetTable (
    PMIB_IPNETTABLE pIpNetTable,
    PULONG          pdwSize,
    BOOL            bOrder
);
```

The first parameter, *pIpNetTable*, is a pointer to an *MIB_IPNETTABLE* structure that returns the ARP information. You must supply a sufficiently large buffer when calling this function. As with most other IP Helper functions, passing *NULL* for this parameter will return the buffer size needed as the parameter *pdwSize* and the error *ERROR_INSUFFICIENT_BUFFER*. Otherwise, *pdwSize* indicates the size of the buffer passed as *pIpNetTable*. The last parameter, *bOrder*, indicates whether the returned IPv4 entries should be sorted in ascending IPv4 order.

The *MIB_IPNETTABLE* structure is a wrapper for an array of *MIB_IPNETROW* structures and is defined as

```
typedef struct _MIB_IPNETTABLE
{
    DWORD        dwNumEntries;
    MIB_IPNETROW table[ANY_SIZE];
} MIB_IPNETTABLE, *PMIB_IPNETTABLE;
```

The *dwNumEntries* field indicates the number of array entries present in the *table* field. The *MIB_IPNETROW* structure contains the actual ARP entry information and is defined as

```
typedef struct _MIB_IPNETROW {
    DWORD dwIndex;
    DWORD dwPhysAddrLen;
    BYTE  bPhysAddr[MAXLEN_PHYSADDR];
    DWORD dwAddr;
    DWORD dwType;
} MIB_IPNETROW, *PMIB_IPNETROW;
```

The fields of this structure are as follows:

- ***dwIndex*** Specifies the index of the adapter.
- ***dwPhysAddrLen*** Indicates the length, in bytes, of the physical address contained in the *bPhysAddr* field.
- ***bPhysAddr*** Is an array of bytes that contains the physical (MAC) address of the adapter.
- ***dwAddr*** Specifies the IP address of the adapter.
- ***dwType*** Indicates the type of the ARP entry. Table 16-12 shows the possible values for this field.

Table 16-12 *Possible ARP Entry Types*

ARP Type	Meaning
MIB_IPNET_TYPE_STATIC	Static entry
MIB_IPNET_TYPE_DYNAMIC	Dynamic entry
MIB_IPNET_TYPE_INVALID	Invalid entry
MIB_IPNET_TYPE_OTHER	Other entry

Adding an ARP Entry

The next function of ARP is adding an entry to the ARP cache, which is another relatively simple operation. The IP Helper function to add an ARP entry is *SetIpNetEntry* and is defined as

```
DWORD SetIpNetEntry (
    PMIB_IPNETROW pArpEntry
);
```

The only argument is the *MIB_IPNETROW* structure, which we covered in the previous section. To add an ARP entry, simply fill in the structure with the new ARP information. First, you need to set the *dwIndex* field to the index of a local IPv4 address that indicates the network on which the ARP entry applies. Remember that if you are given the IP address, you can map the IP to the index with the *GetIpAddrTable* function. The next field, *dwPhysAddrLen*, is typically set to 6. (Most physical addresses, such as ETHERNET MAC addresses, are 6 bytes long.) The *bPhysAddr* byte array must be set to the physical address. Most MAC addresses are represented as 12 characters—for example, 00-A0-C9-A7-86-E8. These characters need to be encoded into the proper byte array locations of the *bPhysAddr* field. For example, the sample MAC address would be encoded into the following bytes:

```
00000000 10100000 11001001 10100111 10000110 11101000
```

The encoding method is the same used for encoding IPX and ATM addresses. (See Chapter 4 for more information.) The *dwAddr* field must be set to the IP address that belongs to the remote host and the specified MAC address. The last field, *dwType*, is set to one of the ARP entry types listed in Table 16-12. Once the structure is filled, call *SetIpNetEntry* to add the ARP entry to the cache. Upon success, *NO_ERROR* is returned.

Deleting an ARP Entry

Deleting an ARP entry is similar to adding one except that the only information required is the interface index, *dwIndex*, and the IPv4 address of the ARP entry to delete, *dwAddr*. The function to remove an ARP entry is *DeleteIpNetEntry*, which is defined as

```
DWORD DeleteIpNetEntry (
    PMIB_IPNETROW pArpEntry
);
```

Again, the only parameter is an *MIB_IPNETROW* structure, and the only information necessary for removing an ARP entry is the local IPv4 index and the IPv4 address of the entry to delete. Remember that the index number to a local IPv4 interface can be obtained with the function *GetIpAddrTable*. Upon success, *NO_ERROR* is returned.

Sending an ARP Request

It is sometimes useful to send an ARP request to populate the ARP cache with the physical address of a destination. For example, sending a UDP datagram larger than the link MTU to a destination that is not in the ARP cache will always fail; on IPv4 it will fail silently and IPv6 will indicate an error. The *SendArp* function will attempt to resolve the given IPv4 address to its MAC address. There is no IPv6 equivalent function but at least IPv6 will indicate that an error has occurred so that the application can retransmit the packet once the physical address has been resolved. The function is declared as

```
DWORD SendArp(
                IPAddr DestIP,
                IPAddr SrcIP,
                PULONG pMacAddr,
                PULONG PhyAddrLen
);
```

The *DestIP* indicates the IPv4 destination address that ARP will attempt to resolve. *SrcIP* is the optional local IPv4 interface to send the ARP request on. If zero, the routing table will determine which local interface to use. *pMacAddr* is a data buffer that receives the destination's physical address. Lastly, *PHyAddrLen* will indicate the length of the physical address returned in the *pMacAddr* buffer.

Conclusion

This chapter introduced the IP Helper APIs in terms of several well-known system utilities. This allows you to easily see how to programmatically obtain useful information from the IPv4 network stack. The IP Helper APIs currently enumerate only IPv4 information except for the new IP Helper API *GetAdaptersAddresses*, which obtains IPv6 addressing information.

Chapter 17

NetBIOS

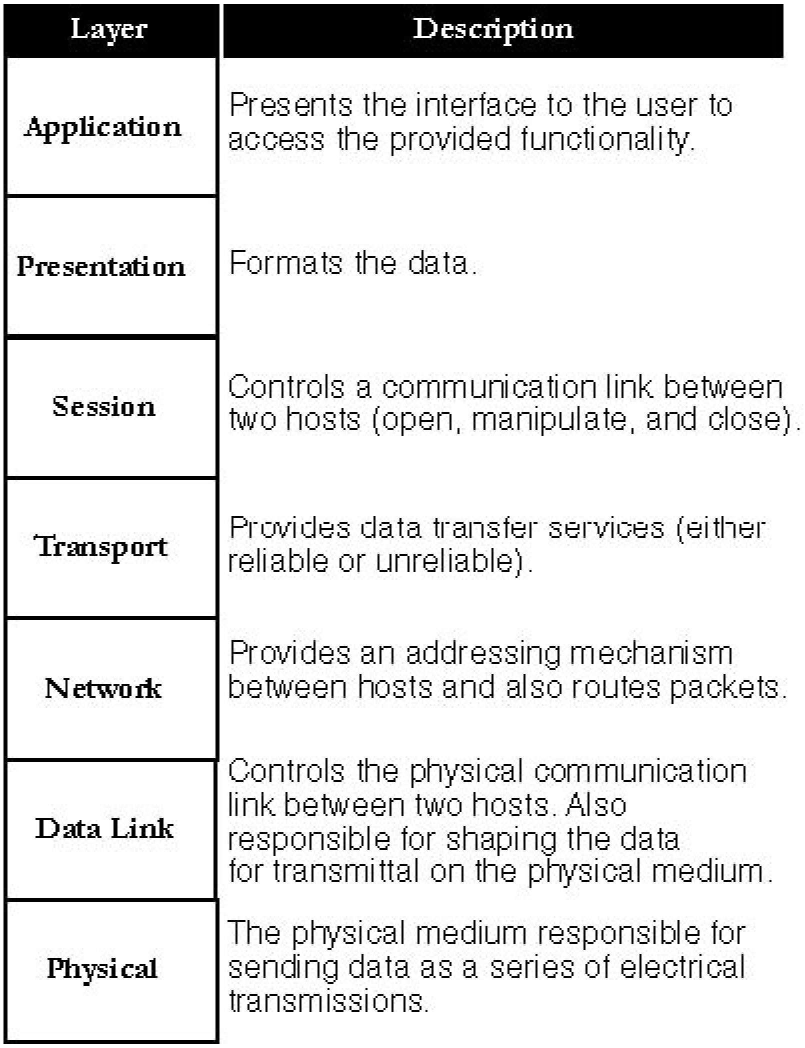
Network Basic Input/Output System (NetBIOS) is a standard application programming interface (API) developed for IBM in 1983 by Sytek Corporation. NetBIOS defines a programming interface for network communication but doesn't detail how the physical frames are transmitted over a network. In 1985, IBM created the NetBIOS Extended User Interface (NetBEUI), which was integrated with the NetBIOS interface to form an exact protocol. The NetBIOS interface became popular enough that vendors started implementing the NetBIOS programming interface on other protocols such as TCP/IP and IPX/SPX. Platforms and applications throughout the world rely on NetBIOS to this day, including many components of Microsoft Windows NT, Windows 2000, Windows 95, and Windows 98.

> Microsoft Windows CE does not support the NetBIOS API, even though it supports TCP/IP as a transport protocol and NetBIOS names and name resolution.

The Windows NetBIOS interface offers backward compatibility with older applications. This chapter discusses the fundamentals of NetBIOS programming. First we cover the NetBIOS basics, beginning with a discussion of NetBIOS names and LANA numbers. We'll follow this with a discussion of basic services offered by NetBIOS, such as session-oriented and connectionless (datagram) communications. In each section, we present a simple client and server example. We'll wrap up this chapter with some common pitfalls and bugs that programmers often run into. Chapter 22 provides a command reference that summarizes each NetBIOS command with its required parameters and a short description of its behavior.

The OSI Network Model

The Open Systems Interconnection (OSI) model offers a high-level representation of network systems. The OSI model contains seven layers that fully describe fundamental network concepts from the application down to the physical method of data transmissions. Figure 17-1 illustrates the seven layers of the OSI model.



Layer	Description
Application	Presents the interface to the user to access the provided functionality.
Presentation	Formats the data.
Session	Controls a communication link between two hosts (open, manipulate, and close).
Transport	Provides data transfer services (either reliable or unreliable).
Network	Provides an addressing mechanism between hosts and also routes packets.
Data Link	Controls the physical communication link between two hosts. Also responsible for shaping the data for transmittal on the physical medium.
Physical	The physical medium responsible for sending data as a series of electrical transmissions.

Figure 17-1 *The OSI network model*

Relative to the OSI model, NetBIOS fits primarily into the session and transport layers.

Microsoft NetBIOS

As we mentioned, NetBIOS API implementations exist for numerous network protocols, making the interface protocol-independent. In other words, if you develop your application according to the NetBIOS specification, the application can run over TCP/IP, NetBEUI, or even IPX/SPX. This is a useful feature because it allows a well-written NetBIOS application to run on almost any machine, regardless of the machine's physical network. However, there are a few considerations. For two NetBIOS applications to communicate with each other over the network, they must be running on workstations that have at least one transport protocol in common. For example, if Joe's machine has only TCP/IP installed and Mary's machine has only NetBEUI, NetBIOS applications on Joe's machine won't be able to communicate with applications on Mary's machine.

Additionally, only certain protocols implement a NetBIOS interface. Microsoft TCP/IP and NetBEUI offer a NetBIOS interface by default; however, IPX/SPX does not. Therefore Microsoft provides a version of IPX/SPX that does implement the interface, which is something to keep in mind when designing a network. When installing protocols, the NetBIOS-capable IPX/SPX protocol usually states something to that effect. For example, Windows 2000 offers the protocol NWLink IPX/SPX/NetBIOS Compatible Transport Protocol. In Windows 95 and Windows 98, the IPX/SPX protocol Properties dialog box has a check box that enables NetBIOS over IPX/SPX.

One other important bit of information is that NetBEUI is not a routable protocol. If there is a router between the client machine and the server machine, applications on those machines will not be able to communicate. The router will drop the packets as it receives them. TCP/IP and IPX/SPX are both routable protocols and do not suffer from this limitation. Keep in mind that if you plan to use NetBIOS heavily, you might want to build your networks with at least one of the routable transport protocols. For more information on protocol characteristics and considerations, see Chapter 2.

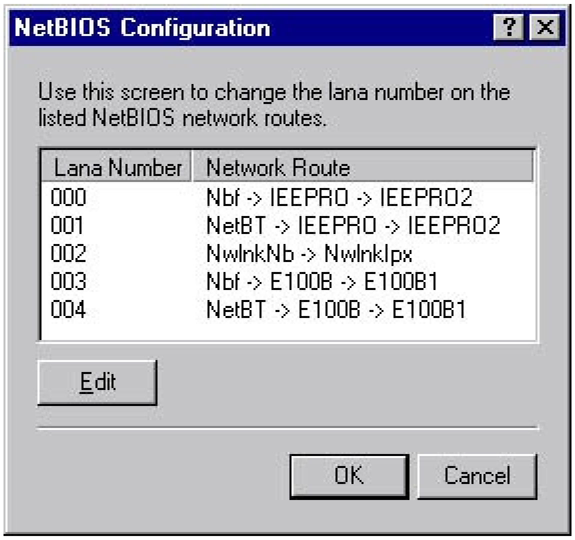
LANA Numbers

You might be wondering how transport protocols relate to NetBIOS from the programming aspect. The answer is LAN Adapter (LANA) numbers, which are the key to understanding NetBIOS. In the original implementations of NetBIOS, each physical network card was assigned a unique value: a LANA number. Under Windows this becomes a bit more problematic, as a workstation can have multiple network protocols installed as well as multiple network interface cards.

A LANA number corresponds to the unique pairings of network adapter with transport protocol. For example, if a workstation has two network cards and two NetBIOS-capable transports (such as TCP/IP and NetBEUI), there will be four LANA numbers. The numbers might correspond to the pairings as follows:

1. TCP/IP—Network Card 1
2. NetBEUI—Network Card 1
3. TCP/IP—Network Card 2

4. NetBEUI—Network Card 2



Use this screen to change the lana number on the listed NetBIOS network routes.

Lana Number	Network Route
000	Nbf -> IEEPRO -> IEEPRO2
001	NetBT -> IEEPRO -> IEEPRO2
002	NwlnkNb -> NwlnkIpx
003	Nbf -> E100B -> E100B1
004	NetBT -> E100B -> E100B1

Figure 17-2 *NetBIOS Configuration dialog box. This machine is multihomed with two network cards and three transport protocols: TCP/IP (NetBT), NetBEUI (Nbf), and IPX/SPX (NwlnkNb).*

NetBIOS Names

Now that we know what LANA numbers are, let's move on to NetBIOS names. A process—or application, if you prefer—registers a name on each LANA number that it wants to communicate with. A NetBIOS name is 16 characters long, with the 16th character reserved for special use. When adding a name to the name table, you should initialize the name buffer to spaces. In the Windows environment, each process has a NetBIOS name table for each available LANA number. Adding a name to LANA 0 means that your application is available to clients connecting on your LANA 0 only. The maximum number of names that can be added to each LANA is 254, with numbering from 1 to 254 (0 and 255 are reserved for the system); however, each operating system sets a default maximum number less than 254 that you can change when resetting each LANA number.

Additionally, there are two types of NetBIOS names: unique and group. A unique name is exactly that: no other process on the network can have that name registered. If another machine does have the name registered, you will receive a duplicate name error. As you might know, machine names in Microsoft networks are NetBIOS names. When a machine boots, it registers its name with the local Windows Internet Naming Service (WINS) server, which reports an error if another machine has that name in use. A WINS server maintains a list of all registered NetBIOS names. Additionally, protocol-specific information can be kept along with the name. For example, on TCP/IP networks, WINS maintains a pairing of NetBIOS names with the IP address that registered the name. If the network is configured without a WINS server, machines perform duplicate name checking by broadcasting a message on the network. If no other machine challenges the message, the network allows the sender to use that name. On the other hand, group names are used to send data to multiple recipients or, conversely, receive data destined for multiple recipients. A group name need not be unique. Group names are used for multicast data transmissions.

The 16th character in NetBIOS names distinguishes most Microsoft networking services. Various networking service and group names are registered with a WINS server by direct name registration from WINS-enabled computers or by broadcast on the local subnet by non-WINS-enabled computers. The *Nbtstat* command is a utility that you can use to obtain information about NetBIOS names that are registered on the local (or remote) computer. In the example shown in Table 17-1, the *Nbtstat -n* command produced this list of registered *NetBIOS* names for user Davemac logged on to a computer configured as a primary domain controller and running Windows NT Server with Microsoft Internet Information Services (IIS).

Table 17-1 *NetBIOS Name Table*

Name	16th Byte	Name Type	Service
DAVEMAC1	<00>	Unique	Workstation service name
DAVEMAC1	<20>	Unique	Server services name
DAVEMACD	<00>	Group	Domain name
DAVEMACD	<1C>	Group	Domain controller name
DAVEMACD	<1B>	Unique	Master browser name
DAVEMAC1	<03>	Unique	Messenger name
Inet~Services	<1C>	Group	IIS group name
IS~DAVEMAC1	<00>	Unique	IIS unique name
DAVEMAC1	<BF>	Unique	Network monitor name

The *Nbtstat* command is installed only when TCP/IP is an installed protocol. This utility can also query name tables of remote machines by using the *-a* parameter followed by the remote machine's name, or by using the *-A* parameter followed by the remote machine's IP address.

Table 17-2 lists the default 16th byte value appended to unique NetBIOS computer names by various Microsoft networking services.

Table 17-2 *Unique Name Qualifiers*

16th Byte	Identifies
<00>	Workstation service name. In general, this is the NetBIOS computer name.
<03>	Messenger service name used when receiving and sending messages. This is the name that is registered with the WINS server as the messenger service on the WINS client and is usually added to the computer name and to the name of the user currently logged on to the computer.
<1B>	Domain master browser name. This name identifies the primary domain controller and indicates which clients and other browsers to use to contact the domain master browser.
<06>	Remote Access Service (RAS) server service.
<1F>	Network Dynamic Data Exchange (NetDDE) service.
<20>	Server service name used to provide share points for file sharing.
<21>	RAS client.
<BE>	Network Monitor Agent.
<BF>	Network Monitor utility.

Table 17-3 lists the default 16th byte character added to commonly used NetBIOS group names.

Table 17-3 *Group Name Qualifiers*

16th Byte	Identifies
<1C>	A domain group name that contains a list of the specific addresses of computers that have registered the domain name. The domain controller registers this name. WINS treats this as a domain group: each member of the group must renew its name individually or be released. The domain group is limited to 25 names. When a static 1C name is replicated that clashes with a dynamic 1C name on another WINS server, a union of the members is added, and the record is marked as static. If the record is static, members of the group do not have to renew their IP addresses.
<1D>	The master browser name used by clients to access the master browser. There is one master browser on a subnet. WINS servers return a positive response to domain name registrations but do not store the domain name in their databases. If a computer sends a domain name query to the WINS server, the WINS server returns a negative response. If the computer that sent the domain name query is configured as h-node or m-node, it will then broadcast the name query to resolve the name. The node type refers to how the client attempts to resolve a name. Clients configured for b-node resolution send broadcast packets to advertise and resolve NetBIOS names. The p-node resolution uses point-to-point communication to a WINS server. The m-node resolution is a mix of b-node and p-node in which b-node is used first and then, if necessary, p-node is used. The last resolution method is h-node, or hybrid node. It always attempts to use p-node registration and resolution first, falling back on b-node only on failure. Windows installations default to h-node.
<1E>	A normal group name. Browsers can broadcast to this name and listen on it to elect a master browser. These broadcasts are for the local subnet and should not cross routers.
<20>	An Internet group name. This type of name is registered with WINS servers to identify groups of computers for administrative purposes. For example, printersg could be a registered group name used to identify an administrative group of print servers.
MSBROWSE	Instead of a single appended 16th character, _MSBROWSE_ is appended to a domain name and broadcast on the local subnet to announce the domain to other master browsers.

So many qualifiers might seem overwhelming. Think of them as a reference. You probably shouldn't be using them in your NetBIOS names. To prevent accidental name collisions with your NetBIOS names, you should avoid using the unique name qualifiers. You should be even more careful with group names—no error will be generated if your name collides with an existing group name. If this happens, you might start receiving data intended for someone else.

NetBIOS Features

NetBIOS offers both connection-oriented services and connectionless (datagram) services. A connection-oriented service allows two entities to establish a session, or virtual circuit, between them. A session is a two-way communication stream whereby each entity can send the other one messages. Session-oriented services provide guaranteed delivery of any data flowing between the two endpoints. In session-oriented services, a server usually registers itself under a certain known name. Clients look for this name to communicate with the server. In NetBIOS terms, the server process adds its name to the name table for each LANA it wants to communicate over. Clients on other machines resolve a service name to a machine name and then ask to connect to the server process. As you can see, a few steps are necessary to establish this circuit, and some overhead is involved in initially setting up the connection. Session-oriented communication guarantees reliability and packet ordering; however, it is still message-based. That is, if a connected client issues a read command, the server will return only one packet of data on the stream, even if the client supplies a buffer large enough for several packets.

On the other hand, there are also connectionless, or datagram, services. In this case a server does register itself under a particular name, but the client simply gathers data and sends it to the network without setting up any connection beforehand. The client addresses the data to the NetBIOS name of the server process. This type of service offers no guarantees, but it offers better performance than connection-oriented services. Furthermore, with datagram services no overhead is involved in setting up a connection. For example, a client might quickly send thousands of bytes of data to a server that crashed two days earlier. The client will never receive any error notifications unless it relies on responses from the server (in which case, it could deduce that something was amiss after not receiving any response for some period of time). Datagram services do not guarantee reliability, nor do they preserve message order.

NetBIOS Programming Basics

Now that we have gone over some of the basic concepts of NetBIOS, we will discuss the NetBIOS API set, which is easy because only one function exists:

```
UCHAR Netbios(PNCB pNCB);
```

All the function declarations, constants, and so on for NetBIOS are defined in the header file NB30.H. The only library necessary for linking NetBIOS applications is NETAPI32.LIB. The most important feature of this function is the parameter *pNCB*, which is a pointer to a network control block (NCB). This is a pointer to an *NCB* structure that contains all the information that the required *Netbios* function needs to execute a NetBIOS command. The definition of this structure is as follows:

```
typedef struct _NCB
{
    UCHAR     ncb_command;
    UCHAR     ncb_retcode;
    UCHAR     ncb_lsn;
    UCHAR     ncb_num;
    PUCHAR    ncb_buffer;
    WORD      ncb_length;
    UCHAR     ncb_callname[NCBNAMSZ];
    UCHAR     ncb_name[NCBNAMSZ];
    UCHAR     ncb_rto;
    UCHAR     ncb_sto;
    void      (*ncb_post) (struct _NCB *);
    UCHAR     ncb_lana_num;
    UCHAR     ncb_cmd_cplt;
    UCHAR     ncb_reserve[10];
    HANDLE    ncb_event;
} * PNCB, NCB;
```

Not all members of the structure will be used in every call to NetBIOS; some of the data fields are output parameters (in other words, set on the return from the *Netbios* call). It is always a good idea to zero out the *NCB* structure before filling in members prior to a *Netbios* call. Take a look at Table 17-4, which describes the usage of each field. Additionally, the command reference in Chapter 22 contains a detailed summary of each NetBIOS command and its required (and optional) fields in an *NCB* structure.

Table 17-4 *NCB Structure Members*

Field	Definition
ncb_command	Specifies the NetBIOS command to execute. Many commands can be executed synchronously or asynchronously by bitwise *OR*ing the *ASYNCH* (0x80) flag and the command.
ncb_retcode	Specifies the return code for the operation. The function sets this value to *NRC_PENDING* while an asynchronous operation is in progress.
	Identifies the local session number that uniquely identifies a session within the

ncb_lsn	current environment. The function returns a new session number after a successful *NCBCALL* or *NCBLISTEN* command.
ncb_num	Specifies the number of the local network name. A new number is returned for each call with an *NCBADDNAME* or *NCBADDGRNAME* command. You must use a valid number on all datagram commands.
ncb_buffer	Points to the data buffer. For commands that send data, this buffer is the data to send. For commands that receive data, this buffer will hold the data on the return from the *Netbios* function. For other commands, such as *NCBENUM*, the buffer will be the predefined structure *LANA_ENUM*.
ncb_length	Specifies the length of the buffer in bytes. For receive commands, *Netbios* sets this value to the number of bytes received. If the specified buffer is not large enough, *Netbios* returns the error *NRC_BUFLEN*.
ncb_callname	Specifies the name of the remote application.
ncb_name	Specifies the name by which the application is known.
ncb_rto	Specifies the timeout period for receive operations. This value is specified as a multiple of 500-millisecond units. The value 0 implies no timeout. This value is set for *NCBCALL* and *NCBLISTEN* commands that affect subsequent *NCBRECV* commands.
ncb_sto	Specifies the timeout period for send operations in 500- millisecond units. The value 0 implies no timeout. This value is set for *NCBCALL* and *NCBLISTEN* commands that affect subsequent *NCBSEND* and *NCBCHAINSEND* commands.
ncb_post	Specifies the address of the post routine to call on completion of the asynchronous command. The function is defined as `void CALLBACK PostRoutine(PNCB pncb);` where *pncb* points to the NCB of the completed command.
ncb_lana_num	Specifies the LANA number to execute the command on.
ncb_cmd_cplt	Specifies the return code for the operation. *Netbios* sets this value to *NRC_PENDING* while an asynchronous operation is in progress.
ncb_reserve	Reserved; must be 0.
ncb_event	Specifies a handle to a Windows event object set to the nonsignaled state. When an asynchronous command is completed, the event is set to its signaled state. Only manual reset events should be used. This field must be 0 if *ncb_command* does not have the *ASYNCH* flag set or if *ncb_post* is not 0; otherwise, *Netbios* returns the error *NRC_ILLCMD*.

Synchronous vs. Asynchronous

When calling the *Netbios* function, you have the option of making the call synchronous or asynchronous. All NetBIOS commands by themselves are synchronous, which means the call to *Netbios* blocks until the command completes. For an *NCBLISTEN* command, the call to *Netbios* does not return until a client establishes a connection or until an error of some kind occurs. To make a command asynchronous, perform a logical *OR* of the NetBIOS command with the flag *ASYNCH*. If you specify the *ASYNCH* flag, you must specify either a post routine in the *ncb_post* field or an event handle in the *ncb_event* field. When an asynchronous command is executed, the value returned from

Netbios is *NRC_GOODRET* (0x00) but the *ncb_cmd_cplt* field is set to *NRC_PENDING* (0xFF). Additionally, the *Netbios* function sets the *ncb_cmd_cplt* field of the *NCB* structure to *NRC_PENDING* until the command completes. After the command completes, the *ncb_cmd_cplt* field is set to the return value of the command. *Netbios* also sets the *ncb_retcode* field to the return value of the command on completion.

Common NetBIOS Routines

In this section, we examine a basic server NetBIOS application. We examine the server first because the design of the server dictates how the client should act. Because most servers are designed to handle multiple clients simultaneously, the asynchronous NetBIOS model fits best. We present server samples using both the asynchronous callback routines and the event model. However, we first introduce source code that implements some common functions necessary to most NetBIOS applications. The following example is from file NBCOMMON.C, which you'll find on the companion CD.

```
// Nbcommon.c

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

#include "nbcommon.h"

//
// Enumerate all LANA numbers
//
int LanaEnum(LANA_ENUM *lenum)
{
    NCB                ncb;

    ZeroMemory(&ncb, sizeof(NCB));
    ncb.ncb_command = NCBENUM;
    ncb.ncb_buffer = (PUCHAR)lenum;
    ncb.ncb_length = sizeof(LANA_ENUM);
     if (Netbios(&ncb) != NRC_GOODRET)
    {
        printf("ERROR: Netbios: NCBENUM: %d\n", ncb.ncb_retcode);
        return ncb.ncb_retcode;
    }
    return NRC_GOODRET;
}

//
// Reset each LANA listed in the LANA_ENUM structure. Also, set
// the NetBIOS environment (max sessions, max name table size),
// and use the first NetBIOS name.
//
int ResetAll(LANA_ENUM *lenum, UCHAR ucMaxSession,
             UCHAR ucMaxName, BOOL bFirstName)
{
    NCB                ncb;
    int                i;

    ZeroMemory(&ncb, sizeof(NCB));
    ncb.ncb_command = NCBRESET;
    ncb.ncb_callname[0] = ucMaxSession;
    ncb.ncb_callname[2] = ucMaxName;
    ncb.ncb_callname[3] = (UCHAR)bFirstName;
    for(i = 0; i < lenum->length; i++)
    {
        ncb.ncb_lana_num = lenum->lana[i];
        if (Netbios(&ncb) != NRC_GOODRET)
        {
            printf("ERROR: Netbios: NCBRESET[%d]: %d\n",
                ncb.ncb_lana_num, ncb.ncb_retcode);
            return ncb.ncb_retcode;
        }
    }
    return NRC_GOODRET;
}

//
// Add the given name to the given LANA number. Return the name
// number for the registered name.
//
int AddName(int lana, char *name, int *num)
{
    NCB                ncb;

    ZeroMemory(&ncb, sizeof(NCB));
    ncb.ncb_command = NCBADDNAME;
    ncb.ncb_lana_num = lana;
    memset(ncb.ncb_name, ' ', NCBNAMSZ);
    strncpy(ncb.ncb_name, name, strlen(name));

    if (Netbios(&ncb) != NRC_GOODRET)
    {
        printf("ERROR: Netbios: NCBADDNAME[lana=%d;name=%s]: %d\n",
            lana, name, ncb.ncb_retcode);
        return ncb.ncb_retcode;
    }
    *num = ncb.ncb_num;
    return NRC_GOODRET;
}

//
// Add the given NetBIOS group name to the given LANA
// number. Return the name number for the added name.
//
int AddGroupName(int lana, char *name, int *num)
{
    NCB                ncb;

    ZeroMemory(&ncb, sizeof(NCB));
    ncb.ncb_command = NCBADDGRNAME;
    ncb.ncb_lana_num = lana;
    memset(ncb.ncb_name, ' ', NCBNAMSZ);
    strncpy(ncb.ncb_name, name, strlen(name));

    if (Netbios(&ncb) != NRC_GOODRET)
    {
        printf("ERROR: Netbios: NCBADDGRNAME[lana=%d;name=%s]: %d\n",
            lana, name, ncb.ncb_retcode);
        return ncb.ncb_retcode;
    }
    *num = ncb.ncb_num;
    return NRC_GOODRET;
}

//
// Delete the given NetBIOS name from the name table associated
// with the LANA number
//
int DelName(int lana, char *name)
{
    NCB                 ncb;

    ZeroMemory(&ncb, sizeof(NCB));
    ncb.ncb_command = NCBDELNAME;
    ncb.ncb_lana_num = lana;
    memset(ncb.ncb_name, ' ', NCBNAMSZ);
    strncpy(ncb.ncb_name, name, strlen(name));

    if (Netbios(&ncb) != NRC_GOODRET)
    {
        printf("ERROR: Netbios: NCBADDNAME[lana=%d;name=%s]: %d\n",
            lana, name, ncb.ncb_retcode);
        return ncb.ncb_retcode;
    }
    return NRC_GOODRET;
}

//
// Send len bytes from the data buffer on the given session (lsn)
// and lana number
//
int Send(int lana, int lsn, char *data, DWORD len)
{
    NCB                 ncb;
    int                 retcode;

    ZeroMemory(&ncb, sizeof(NCB));
    ncb.ncb_command = NCBSEND;
    ncb.ncb_buffer = (PUCHAR)data;
    ncb.ncb_length = len;
    ncb.ncb_lana_num = lana;
    ncb.ncb_lsn = lsn;

    retcode = Netbios(&ncb);

    return retcode;
}

//
// Receive up to len bytes into the data buffer on the given session
// (lsn) and lana number
//
int Recv(int lana, int lsn, char *buffer, DWORD *len)
{
    NCB                 ncb;

    ZeroMemory(&ncb, sizeof(NCB));
    ncb.ncb_command = NCBRECV;
    ncb.ncb_buffer = (PUCHAR)buffer;
    ncb.ncb_length = *len;
    ncb.ncb_lana_num = lana;
    ncb.ncb_lsn = lsn;

    if (Netbios(&ncb) != NRC_GOODRET)
    {
        *len = -1;
        return ncb.ncb_retcode;
    }
    *len = ncb.ncb_length;

    return NRC_GOODRET;
}
//
// Disconnect the given session on the given lana number
//
int Hangup(int lana, int lsn)
{
    NCB                ncb;
    int                retcode;

    ZeroMemory(&ncb, sizeof(NCB));
    ncb.ncb_command = NCBHANGUP;
    ncb.ncb_lsn = lsn;
    ncb.ncb_lana_num = lana;

    retcode = Netbios(&ncb);

    return retcode;
}

//
// Cancel the given asynchronous command denoted in the NCB
// structure parameter
//
int Cancel(PNCB pncb)
{
    NCB                ncb;

    ZeroMemory(&ncb, sizeof(NCB));
    ncb.ncb_command = NCBCANCEL;
    ncb.ncb_buffer = (PUCHAR)pncb;
    ncb.ncb_lana_num = pncb->ncb_lana_num;

    if (Netbios(&ncb) != NRC_GOODRET)
    {
        printf("ERROR: NetBIOS: NCBCANCEL: %d\n", ncb.ncb_retcode);
         return ncb.ncb_retcode;
    }
    return NRC_GOODRET;
}

//
// Format the given NetBIOS name so that it is printable. Any
// unprintable characters are replaced by a period. The outname
// buffer is the returned string, which is assumed to be at least
// NCBNAMSZ + 1 characters in length.
//
int FormatNetbiosName(char *nbname, char *outname)
{
    int        i;

    strncpy(outname, nbname, NCBNAMSZ);
    outname[NCBNAMSZ - 1] = '\0';
    for(i = 0; i < NCBNAMSZ - 1; i++)
    {
        // If the character isn't printable, replace it with a "."
        //
        if (!((outname[i] >= 32) && (outname[i] <= 126)))
            outname[i] = '.';
    }
    return NRC_GOODRET;
}
```

The first of the common routines in NBCOMMON.C is *LanaEnum*. This is the most basic routine that almost all NetBIOS applications use. This function enumerates the available LANA numbers on a given system. The function initializes an *NCB* structure to 0, sets the *ncb_command* field to *NCBENUM*, assigns a *LANA_ENUM* structure to the *ncb_buffer* field, and sets the *ncb_length* field to the size of the *LANA_ENUM* structure. With the *NCB* structure correctly initialized, the only action that the *LanaEnum* function needs to take to invoke the *NCBENUM* command is to call the *Netbios* function. As you can see, executing a NetBIOS command is fairly easy. For synchronous commands, the return value from *Netbios* will tell you whether the command succeeded. The constant *NRC_GOODRET* always indicates success.

A successful NetBIOS call fills the supplied *LANA_ENUM* structure with the count of LANA numbers on the current machine as well as the actual LANA numbers. The *LANA_ENUM* structure is defined as follows:

```
typedef struct LANA_ENUM
{
    UCHAR   length;
    UCHAR   lana[MAX_LANA + 1];
} LANA_ENUM, *PLANA_ENUM;
```

The *length* member indicates how many LANA numbers the local machine has. The *lana* field is the array of actual LANA numbers. The value of *length* corresponds to how many elements of the *lana* array will be filled with LANA numbers.

The next function is *ResetAll*. Again, this function is used in all NetBIOS applications. A well-written NetBIOS program should reset each LANA number that it plans to use. Once you have a *LANA_ENUM* structure with LANA numbers from *LanaEnum*, you can reset them by calling the *NCBRESET* command on each LANA number in the structure. That's exactly what *ResetAll* does; the function's first parameter is a *LANA_ENUM* structure. A reset requires only that the function set *ncb_command* to *NCBRESET* and *ncb_lana_num* to the LANA it needs to reset. Although some platforms, such as Windows 95, do not require you to reset each LANA number that you use, it is good practice to do so. Windows NT requires you to reset each LANA number prior to use;

otherwise, any other calls to *Netbios* will return Error 52 (*NRC_ENVNOTDEF*).

Additionally, when resetting a LANA number, you can set certain NetBIOS environment settings via the character fields of *ncb_callname*. *ResetAll*'s other parameters correspond to these environmental settings. The function uses the *ucMaxSession* parameter to set character 0 of *ncb_callname*, which specifies the maximum number of concurrent sessions. Normally, the operating system imposes a default that is less than the maximum. For example, Windows NT 4.0 defaults to 64 concurrent sessions. *ResetAll* sets character 2 of *ncb_callname* (which specifies the maximum number of NetBIOS names that can be added to each LANA) to the value of the *ucMaxName* parameter. Again, the operating system imposes a default maximum. Finally, *ResetAll* sets character 3, used for NetBIOS clients, to the value of its *bFirstName* parameter. By setting this parameter to *TRUE*, a client uses the machine name as its NetBIOS process name. As a result, a client can connect to a server and send data without allowing any incoming connections. This option is used to save on initialization time because adding a NetBIOS name to the local name table can be costly.

Adding a name to the local name table is another common function. This is what *AddName* does. The parameters are simply the name to add and which LANA number to add it to. Remember that a name table is on a per-LANA basis, and if your application wants to communicate on every available LANA, you need to add the name of the process to every LANA. The command for adding a unique name is *NCBADDNAME*. The other required fields are the LANA number to add the name to and the name to add, which must be copied into *ncb_name*. *AddName* initializes the *ncb_name* buffer to spaces first and assumes that the *name* parameter points to a null-terminated string. After adding a name successfully, *Netbios* returns the NetBIOS name number associated with the newly added name in the *ncb_num* field. You use this value with datagrams to identify the originating NetBIOS process. We discuss datagrams in greater detail later in this chapter. The most common error encountered when adding a unique name is *NRC_DUPNAME*, which occurs when the name is already in use by another process on the network.

AddGroupName works the same way as *AddName,* except that it issues the command *NCBADDGRNAME* and never causes the *NRC_DUPNAME* error.

DelName, another related function, deletes a NetBIOS name from the name table. It requires only the LANA number you want to remove the name from and the name itself.

The next two functions shown in the file NBCOMMON.C, *Send* and *Recv*, are for sending and receiving data in a connected session. These functions are almost identical except for the *ncb_command* field setting. The command field is set to either *NCBSEND* or *NCBRECV*. The LANA number on which to send the data and the session number are both required parameters. A successful *NCBCALL* or *NCBLISTEN* command returns the session number. Clients use the *NCBCALL* command to connect to a known service, and servers use *NCBLISTEN* to "wait" for incoming client connections. When either of these commands succeeds, the NetBIOS interface establishes a session with a unique integer identifier. *Send* and *Recv* also require parameters that map to *ncb_buffer* and *ncb_length*. When sending data, *ncb_buffer* points to the buffer containing the data to send. The length field is the number of characters in the buffer that should be sent. When receiving data, the buffer field points to the block of memory that incoming data is copied to. The length field is the size of the memory chunk. When the *Netbios* function returns, it updates the length field with the number of bytes successfully received. One important aspect of sending data in a session-oriented connection is that a call to the *Send* function will wait until the receiver has posted a *Recv* function. This means that if the sender is pushing a great deal of data and the receiver is not reading it, a lot of resources are being used to buffer the data locally. Therefore, it's a good idea to issue only a few *NCBSEND* or *NCBCHAINSEND* commands simultaneously. To circumvent this problem, use the *Netbios* commands *NCBSENDNA* and *NCBCHAINSENDNA*. With these commands, the sending of the data is performed without waiting for an acknowledgment of receipt from the receiver.

The last two functions near the end of this sample, *Hangup* and *Cancel*, are for closing established sessions or canceling an outstanding command. You can call the NetBIOS command *NCBHANGUP* to gracefully shut down an established session. When you execute this command, all outstanding receive calls for the given session terminate and return with the session-closed error, *NRC_SCLOSED* (0x0A). If any send commands are outstanding, the hangup command blocks until they complete. This delay occurs whether the commands are transferring data or are waiting for the remote side to issue a receive command.

Session Server: Asynchronous Callback Model

Now that we have the basic NetBIOS functions out of the way, we can look at the server that will listen for incoming client connections. Our server will be a simple echo server; it will send back any data that it receives from a connected client. The following sample contains server code that uses asynchronous callback functions. The code is also available as file CBNBSVR.C on the companion CD. If you look at the function *main*, you will see that first we enumerate the available LANA numbers with *LanaEnum*, and then we reset each LANA with *ResetAll*. Remember that these two steps are generally required of all NetBIOS applications.

```
// Cbnbsvr.c

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

#include "..\Common\nbcommon.h"

#define MAX_BUFFER      2048
#define SERVER_NAME     "TEST-SERVER-1"

DWORD WINAPI ClientThread(PVOID lpParam);

//
// Function: ListenCallback
//
// Description:
//    This function is called when an asynchronous listen completes.
//    If no error occurred, create a thread to handle the client.
//    Also, post another listen for other client connections.
//
void CALLBACK ListenCallback(PNCB pncb)
{
    HANDLE      hThread;
    DWORD       dwThreadId;

    if (pncb->ncb_retcode != NRC_GOODRET)
    {
        printf("ERROR: ListenCallback: %d\n", pncb->ncb_retcode);
        return;
    }
    Listen(pncb->ncb_lana_num, SERVER_NAME);

    hThread = CreateThread(NULL, 0, ClientThread, (PVOID)pncb, 0,
        &dwThreadId);
     if (hThread == NULL)
    {
        printf("ERROR: CreateThread: %d\n", GetLastError());
        return;
    }
    CloseHandle(hThread);

    return;
}

//
// Function: ClientThread
//
// Description:
//    The client thread blocks for data sent from clients and
//    simply sends it back to them. This is a continuous loop
//    until the session is closed or an error occurs. If
//    the read or write fails with NRC_SCLOSED, the session
//    has closed gracefully--so exit the loop.
//
DWORD WINAPI ClientThread(PVOID lpParam)
{
    PNCB        pncb = (PNCB)lpParam;
    NCB         ncb;
    char        szRecvBuff[MAX_BUFFER];
    DWORD       dwBufferLen = MAX_BUFFER,
                dwRetVal = NRC_GOODRET;
    char        szClientName[NCBNAMSZ+1];

    FormatNetbiosName(pncb->ncb_callname, szClientName);

    while (1)
    {
        dwBufferLen = MAX_BUFFER;

        dwRetVal = Recv(pncb->ncb_lana_num, pncb->ncb_lsn,
            szRecvBuff, &dwBufferLen);
        if (dwRetVal != NRC_GOODRET)
            break;
        szRecvBuff[dwBufferLen] = 0;
        printf("READ [LANA=%d]: '%s'\n", pncb->ncb_lana_num,
            szRecvBuff);

        dwRetVal = Send(pncb->ncb_lana_num, pncb->ncb_lsn,
            szRecvBuff, dwBufferLen);
        if (dwRetVal != NRC_GOODRET)
            break;
    }
    printf("Client '%s' on LANA %d disconnected\n", szClientName,
        pncb->ncb_lana_num);

    if (dwRetVal != NRC_SCLOSED)
    {
        // Some other error occurred; hang up the connection
        //
        ZeroMemory(&ncb, sizeof(NCB));
        ncb.ncb_command = NCBHANGUP;
        ncb.ncb_lsn = pncb->ncb_lsn;
        ncb.ncb_lana_num = pncb->ncb_lana_num;

        if (Netbios(&ncb) != NRC_GOODRET)
        {
            printf("ERROR: Netbios: NCBHANGUP: %d\n", ncb.ncb_retcode);
            dwRetVal = ncb.ncb_retcode;
        }
        GlobalFree(pncb);
        return dwRetVal;
    }
    GlobalFree(pncb);
    return NRC_GOODRET;
}

//
// Function: Listen
//
// Description:
//    Post an asynchronous listen with a callback function. Create
//    an NCB structure for use by the callback (since it needs a
//    global scope).
//
int Listen(int lana, char *name)
{
    PNCB        pncb = NULL;

    pncb = (PNCB)GlobalAlloc(GMEM_FIXED | GMEM_ZEROINIT, sizeof(NCB));
    pncb->ncb_command = NCBLISTEN | ASYNCH;
    pncb->ncb_lana_num = lana;
    pncb->ncb_post = ListenCallback;
    //
    // This is the name clients will connect to
    //
    memset(pncb->ncb_name, ' ', NCBNAMSZ);
    strncpy(pncb->ncb_name, name, strlen(name));
    //
    // An '*' means we'll take a client connection from anyone. By
     // specifying an actual name here, we restrict connections to
    // clients with that name only.
    //
    memset(pncb->ncb_callname, ' ', NCBNAMSZ);
    pncb->ncb_callname[0] = '*';

    if (Netbios(pncb) != NRC_GOODRET)
    {
        printf("ERROR: Netbios: NCBLISTEN: %d\n", pncb->ncb_retcode);
        return pncb->ncb_retcode;
    }
    return NRC_GOODRET;
}

//
// Function: main
//
// Description:
//    Initialize the NetBIOS interface, allocate some resources, add
//    the server name to each LANA, and post an asynch NCBLISTEN on
//    each LANA with the appropriate callback. Then wait for incoming
//    client connections, at which time, spawn a worker thread to
//    handle them. The main thread simply waits while the server
//    threads are handling client requests. You wouldn't do this in a
//    real application, but this sample is for illustrative purposes
//    only.
//
int main(int argc, char **argv)
{
    LANA_ENUM   lenum;
    int         i,
                num;

    // Enumerate all LANAs and reset each one
    //
    if (LanaEnum(&lenum) != NRC_GOODRET)
        return 1;
    if (ResetAll(&lenum, 254, 254, FALSE) != NRC_GOODRET)
        return 1;
    //
    // Add the server name to each LANA, and issue a listen on each
    //
    for(i = 0; i < lenum.length; i++)
    {
        AddName(lenum.lana[i], SERVER_NAME, &num);
        Listen(lenum.lana[i], SERVER_NAME);
    }
    while (1)
    {
        Sleep(5000);
    }
}
```

The next thing that the function *main* does is add your process's name to each LANA number on which you want to accept connections. The server adds its process name, TEST-SERVER-1, to each LANA number in a loop. This is the name the clients will use to connect to our server (padded with spaces, of course). Every character in a NetBIOS name is significant when trying to establish or accept a connection. We can't stress this point enough. Most problems encountered when coding NetBIOS clients and servers involve mismatched names. Be consistent in padding names either with spaces or with some other character. Spaces are the most popular pad character because when they are enumerated and printed out, they are human-readable.

The last and most crucial step for a server is to post a number of *NCBLISTEN* commands. The *Listen* function first allocates an *NCB* structure. When you use asynchronous NetBIOS calls, the *NCB* structure that you submit must persist from the time you issue the call until the call completes. This requires that you either dynamically allocate each *NCB* structure before issuing the command or maintain a global pool of *NCB* structures for use in asynchronous calls. For *NCBLISTEN*, set the LANA number that you want the call to apply to. Note that the code listing in the file NBCOMMON.C logically *OR*s the *NCBLISTEN* command with the *ASYNCH* command. When specifying the *ASYNCH* command, you must make either the *ncb_post* field or the *ncb_event* field nonzero; if you don't, the *Netbios* call will fail with *NRC_ILLCMD*. In the file CBNBSVR.C, the *Listen* function sets the *ncb_post* field to our callback function, *ListenCallback*. Next, *Listen* sets the *ncb_name* field to the name of the server process. This is the name that clients will connect to. The function also sets the first character of the *ncb_callname* field to an asterisk (*), signifying that the server will accept a connection from any client. Alternatively, you could place a specific name in the *ncb_callname* field, which would allow only the client who registered that specific name to connect to the server. Finally, *Listen* makes a call to *Netbios*. The call completes immediately, and the *Netbios* function sets the *ncb_cmd_cplt* field of the submitted *NCB* structure to *NRC_PENDING* (0xFF) until the command has completed.

Once *main* resets and posts an *NCBLISTEN* command to each LANA number, the main thread goes into a continuous loop.

> Because this server is only a sample, the design is very basic. When writing your own NetBIOS servers, you can do other processing in the main loop or post a synchronous *NCBLISTEN* in the main loop for one of the LANA numbers.

The callback function executes only when an incoming connection is accepted on a LANA number. When the *NCBLISTEN* command accepts a connection, it calls the function in the *ncb_post* field with the originating *NCB* structure as a parameter. The *ncb_retcode* is set to the return code. Always check this value to see whether the client connection succeeded. A successful connection will result in an *ncb_retcode* of *NRC_GOODRET* (0x00).

If the connection was successful, post another *NCBLISTEN* on the same LANA number. This is necessary because once the original listen succeeds, the server stops listening for client connections on that LANA until another *NCBLISTEN* is submitted. Thus, if your servers require a high availability, you can post multiple *NCBLISTEN* commands on the same LANA so that connections from multiple clients can be accepted simultaneously. Finally, the callback function creates a thread that will service the client. In this example, the thread simply loops and calls a blocking read (*NCBRECV*) followed by a blocking send (*NCBSEND*). The server implements an echo server, which reads messages from connected clients and echoes them back. The client thread loops until the client breaks the connection, at which point the client thread issues an *NCBHANGUP* command to close the connection on its end. From there the client thread frees the *NCB* structure and exits.

For connection-oriented sessions, data is buffered by the underlying protocols, so it is not necessary to always have outstanding receive calls. When a receive command is posted, the *Netbios* function immediately transfers available data to the supplied buffer and the call returns. If no data is available, the receive call blocks until data is present or until the session is disconnected. The same is true for the send command: if the network stack is able either to send data immediately on the wire or to buffer the data in the stack for transmission, the call returns immediately. If the system does not have the buffer space to send the data immediately, the send call blocks until the buffer space becomes available. To circumvent this blocking, you can use the *ASYNCH* command on sends and receives. The buffer supplied to asynchronous sends and receives must have a scope that extends beyond the calling procedure. Another way around blocking sends and receives is using the *ncb_sto* and *ncb_rto* fields. The *ncb_sto* field is for send timeouts. By specifying a nonzero value, you set an upper limit for how long a send will block before returning. This number is specified in 500-millisecond units. If a command times out, the data is not sent. The same is true of the receive timeout: if no data arrives within the prescribed amount of time, the call returns with no data transferred into the buffers.

Session Server: Asynchronous Event Model

The following code sample illustrates an echo server that is similar to the one in CBNBSVR.C but uses Windows events as the signaling mechanism for completion. The event model is similar to the callback model. The only difference is that with the callback model, the system executes your code when the asynchronous operation completes, whereas with the event model, your application has to check for the completion of the operation by checking the event status. Because these are standard Windows events, you can use any of the synchronization routines available, such as *WaitForSingleEvent* and *WaitForMultipleEvents*. The event model is more efficient because it forces the programmer to structure the program to consciously check for completion.

```
// Evnbsvr.c

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

#include "..\Common\nbcommon.h"

#define MAX_SESSIONS    254
#define MAX_NAMES       254

#define MAX_BUFFER      2048
#define SERVER_NAME     "TEST-SERVER-1"

NCB    *g_Clients=NULL;        // Global NCB structure for clients

//
// Function: ClientThread
//
// Description:
//    This thread takes the NCB structure of a connected session
//    and waits for incoming data, which it then sends back to the
//    client until the session is closed
//
DWORD WINAPI ClientThread(PVOID lpParam)
{
    PNCB        pncb = (PNCB)lpParam;
    NCB         ncb;
    char        szRecvBuff[MAX_BUFFER],
                szClientName[NCBNAMSZ + 1];
    DWORD       dwBufferLen = MAX_BUFFER,
                dwRetVal = NRC_GOODRET;

    // Send and receive messages until the session is closed
    //
    FormatNetbiosName(pncb->ncb_callname, szClientName);
    while (1)
    {
        dwBufferLen = MAX_BUFFER;
        dwRetVal = Recv(pncb->ncb_lana_num, pncb->ncb_lsn,
            szRecvBuff, &dwBufferLen);
        if (dwRetVal != NRC_GOODRET)
            break;

        szRecvBuff[dwBufferLen] = 0;
        printf("READ [LANA=%d]: '%s'\n", pncb->ncb_lana_num,
            szRecvBuff);

        dwRetVal = Send(pncb->ncb_lana_num, pncb->ncb_lsn,
            szRecvBuff, dwBufferLen);
        if (dwRetVal != NRC_GOODRET)
            break;
    }
    printf("Client '%s' on LANA %d disconnected\n", szClientName,
        pncb->ncb_lana_num);
    //
    // If the error returned from a read or a write is NRC_SCLOSED,
    // all is well; otherwise, some other error occurred, so hang up
    // the connection from this side
    //
    if (dwRetVal != NRC_SCLOSED)
    {
        ZeroMemory(&ncb, sizeof(NCB));
        ncb.ncb_command = NCBHANGUP;
        ncb.ncb_lsn = pncb->ncb_lsn;
        ncb.ncb_lana_num = pncb->ncb_lana_num;

        if (Netbios(&ncb) != NRC_GOODRET)
        {
            printf("ERROR: Netbios: NCBHANGUP: %d\n",
                ncb.ncb_retcode);
            GlobalFree(pncb);
            dwRetVal = ncb.ncb_retcode;
        }
    }
    // The NCB structure passed in is dynamically allocated, so
    // delete it before we go
    //
    GlobalFree(pncb);
    return NRC_GOODRET;
}

//
// Function: Listen
```

```
//
// Description:
//    Post an asynchronous listen on the given LANA number.
//    The NCB structure passed in already has its ncb_event
//    field set to a valid Windows event handle.
//
int Listen(PNCB pncb, int lana, char *name)
{
    pncb->ncb_command = NCBLISTEN | ASYNCH;
    pncb->ncb_lana_num = lana;
    //
    // This is the name clients will connect to
    //
    memset(pncb->ncb_name, ' ', NCBNAMSZ);
    strncpy(pncb->ncb_name, name, strlen(name));
    //
    // An '*' means we'll accept connections from anyone.
    // We can specify a specific name, which means that only a
    // client with the specified name will be allowed to connect.
    //
    memset(pncb->ncb_callname, ' ', NCBNAMSZ);
    pncb->ncb_callname[0] = '*';

    if (Netbios(pncb) != NRC_GOODRET)
    {
        printf("ERROR: Netbios: NCBLISTEN: %d\n", pncb->ncb_retcode);
        return pncb->ncb_retcode;
    }
    return NRC_GOODRET;
}

//
// Function: main
//
// Description:
//    Initialize the NetBIOS interface, allocate some resources, and
//    post asynchronous listens on each LANA using events. Wait for
//    an event to be triggered, and then handle the client
//    connection.
//
int main(int argc, char **argv)
{
    PNCB        pncb=NULL;
    HANDLE      hArray[64],
                hThread;
    DWORD       dwHandleCount=0,
                dwRet,
                dwThreadId;
    int         i,
                num;
    LANA_ENUM   lenum;

    // Enumerate all LANAs and reset each one
    //
    if (LanaEnum(&lenum) != NRC_GOODRET)
        return 1;
    if (ResetAll(&lenum, (UCHAR)MAX_SESSIONS, (UCHAR)MAX_NAMES,
            FALSE) != NRC_GOODRET)
        return 1;
    //
    // Allocate an array of NCB structures (one for each LANA)
    //
    g_Clients = (PNCB)GlobalAlloc(GMEM_FIXED | GMEM_ZEROINIT,
            sizeof(NCB) * lenum.length);
    //
    // Create the events, add the server name to each LANA, and issue
    // the asynchronous listens on each LANA.
    //
    for(i = 0; i < lenum.length; i++)
    {
        hArray[i] = g_Clients[i].ncb_event = CreateEvent(NULL, TRUE,
            FALSE, NULL);

        AddName(lenum.lana[i], SERVER_NAME, &num);
        Listen(&g_Clients[i], lenum.lana[i], SERVER_NAME);
    }
    while (1)
    {
        // Wait until a client connects
        //
        dwRet = WaitForMultipleObjects(lenum.length, hArray, FALSE,
            INFINITE);
        if (dwRet == WAIT_FAILED)
        {
            printf("ERROR: WaitForMultipleObjects: %d\n",
                GetLastError());
            break;
        }
        // Go through all the NCB structures to see whether more
        // than one succeeded. If ncb_cmd_plt is not NRC_PENDING,
        // there is a client; create a thread, and hand off a
        // new NCB structure to the thread. We need to reuse
        // the original NCB for other client connections.
        //
        for(i = 0; i < lenum.length; i++)
        {
            if (g_Clients[i].ncb_cmd_cplt != NRC_PENDING)
            {
                pncb = (PNCB)GlobalAlloc(GMEM_FIXED, sizeof(NCB));
                memcpy(pncb, &g_Clients[i], sizeof(NCB));
                pncb->ncb_event = 0;

                hThread = CreateThread(NULL, 0, ClientThread,
                    (LPVOID)pncb, 0, &dwThreadId);
                CloseHandle(hThread);
                //
                // Reset the handle, and post another listen
                //
                ResetEvent(hArray[i]);
                Listen(&g_Clients[i], lenum.lana[i], SERVER_NAME);
            }
        }
    }
    // Clean up
    //
    for(i = 0; i < lenum.length; i++)
    {
        DelName(lenum.lana[i], SERVER_NAME);
        CloseHandle(hArray[i]);
    }
    GlobalFree(g_Clients);

    return 0;
}
```

Our event-model server starts out exactly the same as the callback server, with the following steps:

1. Enumerate the LANA numbers.
2. Reset each LANA.
3. Add the server's name to each LANA.
4. Post a listen on each LANA.

The only difference is that you need to keep track of all outstanding listen commands because you must associate event completion with the respective *NCB* blocks that initiate a particular command. This code allocates an array of *NCB* structures equal to the number of LANA numbers (as you want to post one *NCBLISTEN* command on each number). Additionally, the code creates an event for each of the *NCB* structures for signaling the command's completion. The *Listen* function takes one of the *NCB* structures from the array as a parameter.

The *main* function's first loop cycles through the available LANA numbers, adding the server name and posting the *NCBLISTEN* command to each LANA number, and building an array of event handles. Next, call *WaitForMultipleObjects*, which blocks until at least one of the handles becomes signaled. Once one or more of the handles in the event-handle array is in a signaled state, *WaitForMultipleObjects* completes and the code spawns a thread to read incoming messages and send them back to the client. The code creates a copy of the signaled *NCB* structure to pass into the client thread. This is because you want to reuse the original *NCB* to post another *NCBLISTEN*, which you can do by resetting the event and calling *Listen* again on that structure. Note that you don't necessarily have to copy the whole structure. In reality you need only the local session number (*ncb_lsn*) and the LANA number (*ncb_lana_num*). However, the *NCB* structure is a nice container for holding both values to pass into the single parameter of the thread. The client thread used by the event model is the same as the callback model except for the *GlobalFree* statement.

Asynchronous Server Strategies

Notice that with both servers the possibility exists of a client being denied service. Once the *NCBLISTEN* completes, there is a slight delay until either the callback function is called or the event gets signaled. The servers don't post another *NCBLISTEN* until a few statements later. If the server accepted a client on LANA 2, for example, and then another client attempted a connection before the server issued another *NCBLISTEN* on that LANA, the client would receive the error *NRC_NOCALL* (0x14), meaning that the given name had no *NCBLISTEN* posted on it. To avoid this, the server could post multiple *NCBLISTEN* commands on each LANA.

From these two server samples, you can see how easy it is to issue asynchronous commands. The *ASYNCH* flag can be applied to just about any NetBIOS command. Just remember that the *NCB* structure that you pass to *Netbios* must have a global scope.

NetBIOS Session Client

The NetBIOS client is similar in design to the asynchronous event server. The following sample contains example code for the client. The client performs the familiar routine initialization steps by name. It adds its own name to the name table of each LANA number and then issues an asynchronous connect command. The main loop waits for one of the events to be signaled. At that point, the code cycles through all the *NCB* structures that correspond to the connect commands it issued, one for each LANA. It checks the *ncb_cmd_cplt* status. If it is *NRC_PENDING*, the code cancels the asynchronous command; if the command is completed (that is, connected) and the *NCB* doesn't correspond to the *NCB* that was signaled (as specified by the return value from *WaitForMultipleObjects*), the code hangs up the connection. If the server is listening on each LANA on its side and the client attempts connections on each of its LANAs, it is possible that more than one connection can succeed. The code simply closes extra connections with the *NCBHANGUP* command—it needs to communicate over only one channel. By attempting to establish a connection using every LANA on both sides, you allow for the greatest possibility of a successful connection.

```
// Nbclient.c

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

#include "..\Common\nbcommon.h"

#define MAX_SESSIONS     254
#define MAX_NAMES        254

#define MAX_BUFFER       1024

char    szServerName[NCBNAMSZ];

//
// Function: Connect
//
// Description:
//    Post an asynchronous connect on the given LANA number to
//    the server. The NCB structure passed in already has the
//    ncb_event field set to a valid Windows event handle. Just
//    fill in the blanks and make the call.
//
int Connect(PNCB pncb, int lana, char *server, char *client)
{
    pncb->ncb_command = NCBCALL | ASYNCH;
    pncb->ncb_lana_num = lana;

    memset(pncb->ncb_name, ' ', NCBNAMSZ);
    strncpy(pncb->ncb_name, client, strlen(client));
    memset(pncb->ncb_callname, ' ', NCBNAMSZ);
    strncpy(pncb->ncb_callname, server, strlen(server));

    if (Netbios(pncb) != NRC_GOODRET)
    {
        printf("ERROR: Netbios: NCBCONNECT: %d\n",
            pncb->ncb_retcode);
        return pncb->ncb_retcode;
    }
     return NRC_GOODRET;
}

//
// Function: main
//
// Description:
//    Initialize the NetBIOS interface, allocate some resources
//    (event handles, a send buffer, and so on), and issue an
//    NCBCALL for each LANA to the given server. Once a connection
//    has been made, cancel or hang up any other outstanding
//    connections. Then send/receive the data. Finally, clean
//    things up.
//
int main(int argc, char **argv)
{
    HANDLE      *hArray;
    NCB         *pncb;
    char         szSendBuff[MAX_BUFFER];
    DWORD        dwBufferLen,
                 dwRet,
                 dwIndex,
                 dwNum;
    LANA_ENUM    lenum;
    int          i;

    if (argc != 3)
    {
        printf("usage: nbclient CLIENT-NAME SERVER-NAME\n");
        return 1;
    }
    // Enumerate all LANAs and reset each one
    //
    if (LanaEnum(&lenum) != NRC_GOODRET)
        return 1;
    if (ResetAll(&lenum, (UCHAR)MAX_SESSIONS, (UCHAR)MAX_NAMES,
            FALSE) != NRC_GOODRET)
        return 1;
    strcpy(szServerName, argv[2]);
    //
    // Allocate an array of handles to use for asynchronous events.
    // Also allocate an array of NCB structures. We need one handle
    // and one NCB for each LANA number.
    //
    hArray = (HANDLE *)GlobalAlloc(GMEM_FIXED,
        sizeof(HANDLE) * lenum.length);
    pncb   = (NCB *)GlobalAlloc(GMEM_FIXED | GMEM_ZEROINIT,
        sizeof(NCB) * lenum.length);
    //
```

```
// Create an event, assign it into the corresponding NCB
// structure, and issue an asynchronous connect (NCBCALL).
// Additionally, don't forget to add the client's name to each
// LANA it wants to connect over.
//
for(i = 0; i < lenum.length; i++)
{
    hArray[i] = CreateEvent(NULL, TRUE, FALSE, NULL);
    pncb[i].ncb_event = hArray[i];

    AddName(lenum.lana[i], argv[1], &dwNum);
    Connect(&pncb[i], lenum.lana[i], szServerName, argv[1]);
}
// Wait for at least one connection to succeed
//
dwIndex = WaitForMultipleObjects(lenum.length, hArray, FALSE,
    INFINITE);
if (dwIndex == WAIT_FAILED)
{
    printf("ERROR: WaitForMultipleObjects: %d\n",
        GetLastError());
}
else
{
    // If more than one connection succeeds, hang up the extra
    // connection. We'll use the connection that was returned
    // by WaitForMultipleObjects. Otherwise, if it's still
    // pending, cancel it.
    //
    for(i = 0; i < lenum.length; i++)
    {
        if (i != dwIndex)
        {
            if (pncb[i].ncb_cmd_cplt == NRC_PENDING)
                Cancel(&pncb[i]);
            else
                Hangup(pncb[i].ncb_lana_num, pncb[i].ncb_lsn);
        }
    }
    printf("Connected on LANA: %d\n", pncb[dwIndex].ncb_lana_num);
    //
    // Send and receive the messages
    //
    for(i = 0; i < 20; i++)
    {
         wsprintf(szSendBuff, "Test message %03d", i);
        dwRet = Send(pncb[dwIndex].ncb_lana_num,
            pncb[dwIndex].ncb_lsn, szSendBuff,
            strlen(szSendBuff));
        if (dwRet != NRC_GOODRET)
            break;
        dwBufferLen = MAX_BUFFER;
        dwRet = Recv(pncb[dwIndex].ncb_lana_num,
            pncb[dwIndex].ncb_lsn, szSendBuff, &dwBufferLen);
        if (dwRet != NRC_GOODRET)
            break;
        szSendBuff[dwBufferLen] = 0;
        printf("Read: '%s'\n", szSendBuff);
        }
        Hangup(pncb[dwIndex].ncb_lana_num, pncb[dwIndex].ncb_lsn);
    }
    // Clean things up
    //
    for(i = 0; i < lenum.length; i++)
    {
        DelName(lenum.lana[i], argv[1]);
        CloseHandle(hArray[i]);
    }
    GlobalFree(hArray);
    GlobalFree(pncb);

    return 0;
}
```

Datagram Operations

Datagrams are connectionless methods of communication. A sender merely addresses each packet with its destination NetBIOS name and sends it on its way. No checking is performed to ensure data integrity, order of arrival, or reliability.

There are three ways to send a datagram. The first is to direct the datagram at a specific (unique or group) name. This means that only the process that registered the destination name can receive that datagram. The second method is to send a datagram to a group name. Only those processes that registered the given group name will be able to receive the message. Finally, the third way to send a datagram is to broadcast it to the entire network. Any process on any workstation on the LAN can receive the datagram. Sending a datagram to either a unique or a group name uses the *NCBDGSEND* command, whereas broadcasts use the *NCBDGSENDBC* command.

Using any of the datagram send commands is a simple process. Set the *ncb_num* field to the name number returned from an *NCBADDNAME* command or using events. For each LANA, the code posts an asynchronous *NCBDGRECV* (or *NCBDGRECVBC*) and waits until one succeeds, at which point it checks all posted commands, prints the messages for those that succeed, and cancels those commands that are still pending. The following example provides functions for both directed and broadcast sends and receives. The program can be compiled into a sample application that can be configured to send or receive datagrams. The program accepts several command-line parameters that allow the user to specify the number of datagrams to send or receive, the delay between sends, the use of broadcasts instead of directed datagrams, the receipt of datagrams for any name, and so on.

```
// Nbdgram.c

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

#include "..\Common\nbcommon.h"

#define MAX_SESSIONS            254
#define MAX_NAMES               254
#define MAX_DATAGRAM_SIZE       512

BOOL   bSender = FALSE,                 // Send or receive datagrams
       bRecvAny = FALSE,                // Receive for any name
       bUniqueName = TRUE,              // Register my name as unique?
       bBroadcast = FALSE,              // Use broadcast datagrams?
       bOneLana = FALSE;                // Use all LANAs or just one?
char   szLocalName[NCBNAMSZ + 1],       // Local NetBIOS name
       szRecipientName[NCBNAMSZ + 1]; // Recipient's NetBIOS name
DWORD  dwNumDatagrams = 25,             // Number of datagrams to send
       dwOneLana,                       // If using one LANA, which one?
       dwDelay = 0;                     // Delay between datagram sends

//
// Function: ValidateArgs
```

```
//
// Description:
//    This function parses the command line arguments
//    and sets various global flags indicating the selections
//
void ValidateArgs(int argc, char **argv)
{
    int                i;

    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 'n':        // Use a unique name
                    bUniqueName = TRUE;
                    if (strlen(argv[i]) > 2)
                        strcpy(szLocalName, &argv[i][3]);
                    break;
                case 'g':        // Use a group name
                    bUniqueName = FALSE;
                    if (strlen(argv[i]) > 2)
                        strcpy(szLocalName, &argv[i][3]);
                    break;
                case 's':        // Send datagrams
                    bSender = TRUE;
                    break;
                case 'c':        // # of datagrams to send or receive
                    if (strlen(argv[i]) > 2)
                        dwNumDatagrams = atoi(&argv[i][3]);
                    break;
                case 'r':        // Recipient's name for datagrams
                    if (strlen(argv[i]) > 2)
                        strcpy(szRecipientName, &argv[i][3]);
                    break;
                case 'b':        // Use broadcast datagrams
                    bBroadcast = TRUE;
                    break;
                case 'a':        // Receive datagrams on any name
                    bRecvAny = TRUE;
                    break;
                case 'l':        // Operate on this LANA only
                    bOneLana = TRUE;
                    if (strlen(argv[i]) > 2)
                        dwOneLana = atoi(&argv[i][3]);
                    break;
                case 'd':        // Delay (millisecs) between sends
                    if (strlen(argv[i]) > 2)
                        dwDelay = atoi(&argv[i][3]);
                    break;
                default:
                    printf("usage: nbdgram ?\n");
                    break;
             }
         }
    }
    return;
}

//
// Function: DatagramSend
//
// Description:
//    Send a directed datagram to the specified recipient on the
//    specified LANA number from the given name number to the
//    specified recipient. Also specified is the data buffer and
//    the number of bytes to send.
//
int DatagramSend(int lana, int num, char *recipient,
                 char *buffer, int buflen)
{
    NCB                ncb;

    ZeroMemory(&ncb, sizeof(NCB));
    ncb.ncb_command = NCBDGSEND;
    ncb.ncb_lana_num = lana;
    ncb.ncb_num = num;
    ncb.ncb_buffer = (PUCHAR)buffer;
    ncb.ncb_length = buflen;

    memset(ncb.ncb_callname, ' ', NCBNAMSZ);
    strncpy(ncb.ncb_callname, recipient, strlen(recipient));

    if (Netbios(&ncb) != NRC_GOODRET)
    {
        printf("Netbios: NCBDGSEND failed: %d\n", ncb.ncb_retcode);
        return ncb.ncb_retcode;
    }
    return NRC_GOODRET;
}

//
// Function: DatagramSendBC
//
// Description:
//    Send a broadcast datagram on the specified LANA number from the
//    given name number. Also specified is the data buffer and the number
//    of bytes to send.
//
int DatagramSendBC(int lana, int num, char *buffer, int buflen)
{
    NCB                ncb;

    ZeroMemory(&ncb, sizeof(NCB));
    ncb.ncb_command = NCBDGSENDBC;
    ncb.ncb_lana_num = lana;
    ncb.ncb_num = num;
    ncb.ncb_buffer = (PUCHAR)buffer;
    ncb.ncb_length = buflen;

    if (Netbios(&ncb) != NRC_GOODRET)
    {
        printf("Netbios: NCBDGSENDBC failed: %d\n", ncb.ncb_retcode);
        return ncb.ncb_retcode;
    }
    return NRC_GOODRET;
}

//
// Function: DatagramRecv
//
// Description:
//    Receive a datagram on the given LANA number directed toward the
//    name represented by num. Data is copied into the supplied buffer.
//    If hEvent is not 0, the receive call is made asynchronously
//    with the supplied event handle. If num is 0xFF, listen for a
//    datagram destined for any NetBIOS name registered by the process.
//
int DatagramRecv(PNCB pncb, int lana, int num, char *buffer,
                 int buflen, HANDLE hEvent)
{
    ZeroMemory(pncb, sizeof(NCB));
    if (hEvent)
    {
        pncb->ncb_command = NCBDGRECV | ASYNCH;
        pncb->ncb_event = hEvent;
    }
    else
        pncb->ncb_command = NCBDGRECV;
    pncb->ncb_lana_num = lana;
    pncb->ncb_num = num;
    pncb->ncb_buffer = (PUCHAR)buffer;
    pncb->ncb_length = buflen;

    if (Netbios(pncb) != NRC_GOODRET)
    {
        printf("Netbos: NCBDGRECV failed: %d\n", pncb->ncb_retcode);
        return pncb->ncb_retcode;
     }
    return NRC_GOODRET;
}

//
// Function: DatagramRecvBC
//
// Description:
//    Receive a broadcast datagram on the given LANA number.
//    Data is copied into the supplied buffer. If hEvent is not 0,
//    the receive call is made asynchronously with the supplied
//    event handle.
//
int DatagramRecvBC(PNCB pncb, int lana, int num, char *buffer,
                   int buflen, HANDLE hEvent)
{
    ZeroMemory(pncb, sizeof(NCB));
    if (hEvent)
    {
        pncb->ncb_command = NCBDGRECVBC | ASYNCH;
        pncb->ncb_event = hEvent;
    }
    else
        pncb->ncb_command = NCBDGRECVBC;
    pncb->ncb_lana_num = lana;
    pncb->ncb_num = num;
    pncb->ncb_buffer = (PUCHAR)buffer;
    pncb->ncb_length = buflen;

    if (Netbios(pncb) != NRC_GOODRET)
    {
        printf("Netbios: NCBDGRECVBC failed: %d\n",
        pncb->ncb_retcode);
        return pncb->ncb_retcode;
    }
    return NRC_GOODRET;
}

//
// Function: main
//
// Description:
//    Initialize the NetBIOS interface, allocate resources, and then
//    send or receive datagrams according to the user's options
//
int main(int argc, char **argv)
{
    LANA_ENUM   lenum;
    int         i, j;
    char        szMessage[MAX_DATAGRAM_SIZE],
                szSender[NCBNAMSZ + 1];
    DWORD      *dwNum = NULL,
                dwBytesRead,
                dwErr;

    ValidateArgs(argc, argv);
    //
    // Enumerate and reset the LANA numbers
    //
    if ((dwErr = LanaEnum(&lenum)) != NRC_GOODRET)
    {
        printf("LanaEnum failed: %d\n", dwErr);
        return 1;
    }
    if ((dwErr = ResetAll(&lenum, (UCHAR)MAX_SESSIONS,
        (UCHAR)MAX_NAMES, FALSE)) != NRC_GOODRET)
    {
        printf("ResetAll failed: %d\n", dwErr);
        return 1;
    }
    //
    // This buffer holds the name number for the NetBIOS name added
    // to each LANA
    //
    dwNum = (DWORD *)GlobalAlloc(GMEM_FIXED | GMEM_ZEROINIT,
            sizeof(DWORD) * lenum.length);
    if (dwNum == NULL)
    {
        printf("out of memory\n");
        return 1;
    }
    //
    // If we're going to operate on only one LANA, register the name
    // on only that specified LANA; otherwise, register it on all
    // LANAs
    //
    if (bOneLana)
    {
        if (bUniqueName)
            AddName(dwOneLana, szLocalName, &dwNum[0]);
        else
            AddGroupName(dwOneLana, szLocalName, &dwNum[0]);
    }
    else
    {
         for(i = 0; i < lenum.length; i++)
        {
            if (bUniqueName)
                AddName(lenum.lana[i], szLocalName, &dwNum[i]);
            else
                AddGroupName(lenum.lana[i], szLocalName, &dwNum[i]);
        }
    }
    // We are sending datagrams
    //
    if (bSender)
    {
        // Broadcast sender
        //
        if (bBroadcast)
        {
            if (bOneLana)
            {
                // Broadcast the message on the one LANA only
                //
                for(j = 0; j < dwNumDatagrams; j++)
                {
                    wsprintf(szMessage,
                        "[%03d] Test broadcast datagram", j);
                    if (DatagramSendBC(dwOneLana, dwNum[0],
                        szMessage, strlen(szMessage))
                        != NRC_GOODRET)
                        return 1;
                    Sleep(dwDelay);
                }
            }
            else
            {
                // Broadcast the message on every LANA on the local
                // machine
                //
                for(j = 0; j < dwNumDatagrams; j++)
                {
                    for(i = 0; i < lenum.length; i++)
                    {
```

```
                    wsprintf(szMessage,
                        "[%03d] Test broadcast datagram", j);
                    if (DatagramSendBC(lenum.lana[i], dwNum[i],
                        szMessage, strlen(szMessage))
                        != NRC_GOODRET)
                        return 1;
                }
                Sleep(dwDelay);
            }
        }
    }
    else
    {
        if (bOneLana)
        {
            // Send a directed message to the one LANA specified
            //
            for(j = 0; j < dwNumDatagrams; j++)
            {
                wsprintf(szMessage,
                    "[%03d] Test directed datagram", j);
                if (DatagramSend(dwOneLana, dwNum[0],
                    szRecipientName, szMessage,
                    strlen(szMessage)) != NRC_GOODRET)
                    return 1;
                Sleep(dwDelay);
            }
        }
        else
        {
            // Send a directed message to each LANA on the
            // local machine
            //
            for(j = 0; j < dwNumDatagrams; j++)
            {
                for(i = 0; i < lenum.length; i++)
                {
                    wsprintf(szMessage,
                        "[%03d] Test directed datagram", j);
                    printf("count: %d.%d\n", j,i);
                    if (DatagramSend(lenum.lana[i], dwNum[i],
                        szRecipientName, szMessage,
                        strlen(szMessage)) != NRC_GOODRET)
                        return 1;
                }
                Sleep(dwDelay);
            }
        }
    }
}
else                // We are receiving datagrams
{
    NCB     *ncb=NULL;
     char    **szMessageArray = NULL;
    HANDLE  *hEvent=NULL;
    DWORD   dwRet;
```

```
// Allocate an array of NCB structure to submit to each recv
// on each LANA
//
ncb = (NCB *)GlobalAlloc(GMEM_FIXED | GMEM_ZEROINIT,
            sizeof(NCB) * lenum.length);
//
// Allocate an array of incoming data buffers
//
szMessageArray = (char **)GlobalAlloc(GMEM_FIXED,
        sizeof(char *) * lenum.length);
for(i = 0; i < lenum.length; i++)
    szMessageArray[i] = (char *)GlobalAlloc(GMEM_FIXED,
            MAX_DATAGRAM_SIZE);
//
// Allocate an array of event handles for
// asynchronous receives
//
hEvent = (HANDLE *)GlobalAlloc(GMEM_FIXED | GMEM_ZEROINIT,
        sizeof(HANDLE) * lenum.length);
for(i = 0; i < lenum.length; i++)
    hEvent[i] = CreateEvent(0, TRUE, FALSE, 0);

if (bBroadcast)
{
    if (bOneLana)
    {
        // Post synchronous broadcast receives on
        // the one LANA specified
        //
        for(j = 0; j < dwNumDatagrams; j++)
        {
            if (DatagramRecvBC(&ncb[0], dwOneLana, dwNum[0],
                szMessageArray[0], MAX_DATAGRAM_SIZE,
                NULL) != NRC_GOODRET)
                return 1;
            FormatNetbiosName(ncb[0].ncb_callname, szSender);
            printf("%03d [LANA %d] Message: '%s' "
                "received from: %s\n", j,
                ncb[0].ncb_lana_num, szMessageArray[0],
                szSender);
        }
    }
    else
    {
        // Post asynchronous broadcast receives on each LANA
        // number available. For each command that succeeded,
        // print the message; otherwise, cancel the command.
        //
        for(j = 0; j < dwNumDatagrams; j++)
        {
            for(i = 0; i < lenum.length; i++)
            {
                dwBytesRead = MAX_DATAGRAM_SIZE;
                if (DatagramRecvBC(&ncb[i], lenum.lana[i],
                    dwNum[i], szMessageArray[i],
                    MAX_DATAGRAM_SIZE, hEvent[i])
                    != NRC_GOODRET)
```

```
                        return 1;
                }
                dwRet = WaitForMultipleObjects(lenum.length,
                    hEvent, FALSE, INFINITE);
                if (dwRet == WAIT_FAILED)
                {
                    printf("WaitForMultipleObjects failed: %d\n",
                        GetLastError());
                    return 1;
                }
                for(i = 0; i < lenum.length; i++)
                {
                    if (ncb[i].ncb_cmd_cplt == NRC_PENDING)
                        Cancel(&ncb[i]);
                    else
                    {
                        ncb[i].ncb_buffer[ncb[i].ncb_length] = 0;
                        FormatNetbiosName(ncb[i].ncb_callname,
                            szSender);
                        printf("%03d [LANA %d] Message: '%s' "
                            "received from: %s\n", j,
                            ncb[i].ncb_lana_num,
                            szMessageArray[i], szSender);
                    }
                    ResetEvent(hEvent[i]);
                }
            }
        }
    }
    else
    {
        if (bOneLana)
        {
            // Make a blocking datagram receive on the specified
            // LANA number
            //
            for(j = 0; j < dwNumDatagrams; j++)
            {
                if (bRecvAny)
                {
                    // Receive data destined for any NetBIOS name
                    // in this process's name table
                    //
                    if (DatagramRecv(&ncb[0], dwOneLana, 0xFF,
                        szMessageArray[0], MAX_DATAGRAM_SIZE,
                        NULL) != NRC_GOODRET)
                        return 1;
                }
                else
                {
                    if (DatagramRecv(&ncb[0], dwOneLana,
                        dwNum[0], szMessageArray[0],
                        MAX_DATAGRAM_SIZE, NULL)
                        != NRC_GOODRET)
                        return 1;
                }
                FormatNetbiosName(ncb[0].ncb_callname, szSender);
                printf("%03d [LANA %d] Message: '%s' "
                       "received from: %s\n", j,
                       ncb[0].ncb_lana_num, szMessageArray[0],
                       szSender);
            }
        }
        else
        {
            // Post asynchronous datagram receives on each LANA
            // available. For all those commands that succeeded,
            // print the data; otherwise, cancel the command.
            //
            for(j = 0; j < dwNumDatagrams; j++)
            {
                for(i = 0; i < lenum.length; i++)
                {
                    if (bRecvAny)
                    {
                        // Receive data destined for any NetBIOS
                        // name in this process's name table
                        //
                        if (DatagramRecv(&ncb[i], lenum.lana[i],
                            0xFF, szMessageArray[i],
                            MAX_DATAGRAM_SIZE, hEvent[i])
                            != NRC_GOODRET)
                            return 1;
                    }
                    else
                    {
                        if (DatagramRecv(&ncb[i], lenum.lana[i],
                            dwNum[i], szMessageArray[i],
                            MAX_DATAGRAM_SIZE, hEvent[i])
                            != NRC_GOODRET)
                            return 1;
                    }
                }
                dwRet = WaitForMultipleObjects(lenum.length,
                    hEvent, FALSE, INFINITE);
                if (dwRet == WAIT_FAILED)
                {
                    printf("WaitForMultipleObjects failed: %d\n",
                        GetLastError());
                    return 1;
                }
                for(i = 0; i < lenum.length; i++)
                {
                    if (ncb[i].ncb_cmd_cplt == NRC_PENDING)
                        Cancel(&ncb[i]);
                    else
                    {
                        ncb[i].ncb_buffer[ncb[i].ncb_length] = 0;
                        FormatNetbiosName(ncb[i].ncb_callname,
                            szSender);
                        printf("%03d [LANA %d] Message: '%s' "
                            "from: %s\n", j, ncb[i].ncb_lana_num,
                            szMessageArray[i], szSender);
                    }
```

```
                            ResetEvent(hEvent[i]);
                        }
                    }
                }
            }
            // Clean up
            //
            for(i = 0; i < lenum.length; i++)
            {
                CloseHandle(hEvent[i]);
                GlobalFree(szMessageArray[i]);
            }
            GlobalFree(hEvent);
            GlobalFree(szMessageArray);
        }
        // Clean things up
        //
        if (bOneLana)
            DelName(dwOneLana, szLocalName);
        else
        {
            for(i = 0; i < lenum.length; i++)
                DelName(lenum.lana[i], szLocalName);
        }
        GlobalFree(dwNum);

        return 0;
}
```

Once you've compiled the example, run the following tests to get an idea of how datagrams work. For learning purposes, you should run two instances of the applications, but on separate machines. If you run them on the same machine, they'll work, but this hides some important concepts. When run on the same machine, the LANA numbers for each side correspond to the same protocol. It's more interesting when they don't. Table 17-5 lists some commands to try, and Table 17-6 lists all the command-line options available for use with the sample program.

Table 17-5 *NBDGRAM.C Commands*

Client Command	Server Command
Nbdgram /n:CLIENT01	Nbdgram /s /n:SERVER01 /r:CLIENT01
Nbdgram /n:CLIENT01 /b	Nbdgram /s /n:SERVER01 /b
Nbdgram /g:CLIENTGROUP	Nbdgram /s /r:CLIENTGROUP

Table 17-6 *Command Parameters for NBDGRAM.C*

Flag	Meaning
/n:*my-name*	Register the unique name *my-name*.
/g:*group-name*	Register the group name *group-name*.
/s	Send datagrams (by default, the sample receives datagrams).
/c:*n*	Send or receive *n* number of datagrams.
/r:*receiver*	Specify the NetBIOS name to send the datagrams to.

/b	Use broadcast datagrams.
/a	Post receives for any NetBIOS name (set *ncb_num* to 0xFF).
/l:*n*	Perform all operations on LANA *n* only (by default, all sends and receives are posted on each LANA).
/d:*n*	Wait *n* milliseconds between sends.

For the third command in Table 17-5, run several clients on various machines. This illustrates one server sending one message to a group, and each member of the group waiting for data will receive the message. Also, try various combinations of the listed commands with the /l:*x* command-line option, where *x* is a valid LANA number. This command-line option switches the program's mode from performing the commands on all LANAs to performing the commands on the listed LANA only. For example, the command Nbdgram /n:CLIENT01 /l:0 makes the application listen only for incoming datagrams on LANA 0 and ignore any data arriving on any other LANA. Additionally, option /a is meaningful only to the clients. This flag causes the receive command to pick up incoming datagrams destined for any NetBIOS name registered by the process. In our example, this isn't very meaningful because the client registers only one name, but you can at least see how this would be coded. You might want to try modifying the code so that it registers a name for every /n:*name* option in the command line. Start up the server with the recipient flag set to only one of the names that the client registered. The client will receive the data, even though the *NCBDGRECV* command does not specifically refer to a particular name.

Miscellaneous NetBIOS Commands

All of the commands discussed so far deal in some way with setting up a session, sending or receiving data through a session or a datagram, and related subjects. A few commands deal exclusively in getting information. These commands are the adapter status command (*NCBASTAT*) and the find name command (*NCBFINDNAME*), which are discussed in the following sections. The final section deals with matching LANA numbers to their protocols in a programmatic fashion. (This is not actually a NetBIOS function; we discuss it because it can gather useful NetBIOS information for you.)

Adapter Status (*NCBASTAT*)

The adapter status command is useful for obtaining information about the local computer and its LANA numbers. Using this command is also the only way to programmatically find the machine's MAC address from Windows 95 and Windows NT 4.0. With the advent of the IP Helper functions for Windows 2000 and Windows 98 (discussed in Chapter 22), there is a more generic interface for finding the Media Access Control (MAC) address; however, for the other Windows platforms, using the adapter status command is your only valid option.

The command and its syntax are fairly easy to understand, but two ways of calling the function affect what data is returned. The adapter status command returns an *ADAPTER_STATUS* structure followed by a number of *NAME_BUFFER* structures. The structures are defined as follows:

```
typedef struct _ADAPTER_STATUS {
    UCHAR   adapter_address[6];
    UCHAR   rev_major;
    UCHAR   reserved0;
    UCHAR   adapter_type;
    UCHAR   rev_minor;
    WORD    duration;
    WORD    frmr_recv;
    WORD    frmr_xmit;
    WORD    iframe_recv_err;
    WORD    xmit_aborts;
    DWORD   xmit_success;
    DWORD   recv_success;
    WORD    iframe_xmit_err;
    WORD    recv_buff_unavail;
    WORD    t1_timeouts;
    WORD    ti_timeouts;
    DWORD   reserved1;
    WORD    free_ncbs;
    WORD    max_cfg_ncbs;
    WORD    max_ncbs;
    WORD    xmit_buf_unavail;
    WORD    max_dgram_size;
    WORD    pending_sess;
    WORD    max_cfg_sess;
    WORD    max_sess;
    WORD    max_sess_pkt_size;
    WORD    name_count;
} ADAPTER_STATUS, *PADAPTER_STATUS;
typedef struct _NAME_BUFFER {
    UCHAR   name[NCBNAMSZ];
    UCHAR   name_num;
    UCHAR   name_flags;
} NAME_BUFFER, *PNAME_BUFFER;
```

The fields of most interest are MAC address (*adapter_address*), maximum datagram size (*max_dgram_size*), and maximum number of sessions (*max_sess*). Also, the *name_count* field tells you how many *NAME_BUFFER* structures were returned. The maximum number of NetBIOS names per LANA is 254, so you have a choice of providing a buffer large enough for all names or calling the adapter status command once with *ncb_length* equal to 0. When the *Netbios* function returns, it provides the necessary buffer size.

```
     UCHAR   unique_group;
} FIND_NAME_HEADER, *PFIND_NAME_HEADER;

typedef struct _FIND_NAME_BUFFER {
    UCHAR   length;
    UCHAR   access_control;
    UCHAR   frame_control;
    UCHAR   destination_addr[6];
    UCHAR   source_addr[6];
    UCHAR   routing_info[18];
} FIND_NAME_BUFFER, *PFIND_NAME_BUFFER;
```

As with the adapter status command, if the *NCBFINDNAME* command is executed with a buffer length of 0, the *Netbios* function returns the required length with the error *NRC_BUFLEN*.

The *FIND_NAME_HEADER* structure that a successful query returns indicates whether the name is registered as a unique name or a group name. If the field *unique_group* is 0, it is a unique name. The value 1 indicates a group name. The *node_count* field indicates how many *FIND_NAME_BUFFER* structures were returned. The *FIND_NAME_BUFFER* structure returns quite a bit of information, most of which is useful at the protocol level. However, we're interested in the fields *destination_addr* and *source_addr*. The *source_addr* field contains the MAC address of the network adapter that has registered the name, and the *destination_addr* field contains the MAC address of the adapter that performed the query.

A find name query can be issued on any LANA number on the local machine. The data returned should be identical on all valid LANA numbers for the local network. (For example, you can execute a find name command on a RAS connection to determine whether a name is registered on the remote network.) Using Windows NT 4.0, you will find the following bug: when a find name query is executed over TCP/IP, *Netbios* returns bogus information. Therefore, if you plan to use this query with Windows NT 4.0, be sure to pick a LANA corresponding to a transport other than TCP/IP.

Matching Transports to LANA Numbers

This last section discusses matching transport protocols such as TCP/IP and NetBEUI to their LANA numbers. Because there are different potential problems to deal with depending on which transport your application is using, it's nice to be able to find these transports programmatically. This isn't possible with a native NetBIOS call, but it is possible with Winsock 2 under Windows NT 4.0 and Windows 2000. The Winsock 2 function *WSAEnumProtocols* returns information about available transport protocols. (See Chapters 5 and 6 for more information about *WSAEnumProtocols*.) Although Winsock 2 is available on Windows 95 and by default on Windows 98, the protocol information stored on these platforms does not contain any NetBIOS information, which is what we're looking for.

We won't discuss Winsock 2 in great detail, as this was the subject of Part II of this book. The basic steps involved are loading Winsock 2 through the *WSAStartup* function, calling *WSAEnumProtocols*, and inspecting the *WSAPROTOCOL_INFO* structures returned from the call. The sample file NBPROTO.C on this book's companion CD contains code for performing this query.

The *WSAEnumProtocols* function takes a buffer to a block of data and a buffer-length parameter. First call the function with a null buffer address and 0 for the length. The call will fail, but the buffer-length parameter will contain the size of the buffer required. Once you have the proper size, call the function again. *WSAEnumProtocols* returns the number of protocols it found. The *WSAPROTOCOL_INFO* structure is large and contains a lot of fields, but the ones we're interested in are *szProtocol*, *iAddressFamily*, and *iProtocol*. If *iAddressFamily* is equal to *AF_NETBIOS*, the absolute value of *iProtocol* is the LANA number for the protocol given in the string *szProtocol*. In addition, the *ProviderId* GUID can be used to match the returned protocol to certain predefined GUIDs for protocols.

There is only one “gotcha” with this method. Under Windows NT and Windows 2000, the *iProtocol* field for any protocol installed on LANA 0 is the value 0x80000000 because protocol 0 is reserved for special use. Any protocol assigned LANA 0 will always have the value 0x80000000, so it is a matter of simply checking for this value.

Platform Considerations

Keep these limitations in mind when implementing NetBIOS with the following platforms.

Windows CE

The NetBIOS interface is not available on Windows CE. Although the redirector supports NetBIOS names and name resolution, there is no programming interface support.

Windows 95 and Windows 98

There are several bugs to watch out for in Windows 95 and Windows 98. On either of these two platforms, you must reset all LANA numbers before adding any NetBIOS name to any LANA. This is because resetting one LANA corrupts the name tables of the others; therefore, you want to avoid code similar to the following:

```
 LANA_ENUM    lenum;
// Enumerate the LANAs
for(i = 0; i < lenum.length; i++)
{
    Reset(lenum.lana[i]);
    AddName(lenum.lana[i], MY_NETBIOS_NAME);
}
```

In addition, with Windows 95, do not attempt to perform an asynchronous *NCBRESET* command on the LANA corresponding to the TCP/IP protocol. To begin with, you shouldn't issue this command asynchronously because a reset has to complete before you can do anything with that LANA anyway. If you do decide to execute an *NCBRESET* command asynchronously, your application will cause a fatal error in the NetBIOS TCP/IP virtual device driver (VXD), and you will have to reboot your computer.

General

When performing session-oriented communications, one side can send as much data as it wants; however, the sender really buffers the data it sends until the receiver acknowledges receiving the data by posting a receive command. The NetBIOS commands *NCBSENDNA* and *NCBCHAINSENDNA* are the “no acknowledgment required” versions of the send commands. You can use these commands if you specifically don't want your send commands to wait for acknowledgment from the receiver. Because TCP/IP provides its own acknowledgment scheme in the underlying protocol, these versions of the send commands (versions that don't require acknowledgment from the receiver) behave exactly like the versions that do require acknowledgment.

Conclusion

The NetBIOS interface is a powerful but outdated application interface. One of its strengths is its protocol independence—applications can run over TCP/IP, NetBEUI, and IPX/SPX. NetBIOS offers both connection-oriented and connectionless communication. One major advantage the NetBIOS interface has over the Winsock interface is a unified name resolution and registration method. That is, a NetBIOS application only needs a NetBIOS name to operate, whereas a Winsock application that utilizes different protocols needs to be aware of each protocol's addressing scheme (as you learned in Part II of this book). Chapter 18 introduces the redirector, which is an integral part of mailslots and named pipes, which you'll learn about in Chapters 19 and 20.

Chapter 18

The Redirector

Microsoft Windows offers applications the capability to communicate over a network using built-in file system services. This is sometimes referred to as the network operating system (NOS) capability. This chapter explores these networking capabilities using Windows file system components available in Windows 95, Windows 98, Windows Me, Windows NT, and Windows CE. The purpose of this chapter is to provide an understanding of these capabilities as they relate to the mailslot and named pipe networking technologies, which are covered in greater detail in Chapters 19 and 20, respectively.

When applications want to access files on a local system, they rely on the operating system to service I/O requests. This is typically referred to as *local I/O*. For example, when an application opens or closes a file, the operating system determines how to access a device that contains the contents of the specified file. Once the device is found, the I/O request is forwarded to a local device driver. The same operating principle is also available for accessing devices over a network. However, the I/O request must be forwarded over a network to the remote device. This is referred to as *I/O redirection*. For example, Windows allows you to map or redirect a local disk identifier—such as E:—to a directory share point on a remote computer. When applications reference E:, the operating system redirects the I/O to a device called a *redirector*. The redirector forms a communication channel to a remote computer to access the desired remote directory. This functionality allows applications to use common file system API functions, such as *ReadFile* and *WriteFile*, to access remote files across a network.

This chapter discusses how the redirector is used to redirect I/O requests to remote devices. This is important information—it is the foundation for communication in the mailslot and named pipe technologies. First we cover how files can be referenced over a network with the UNC using the MUP resource locator. This is followed by an explanation of how MUP calls a network provider, which exposes a redirector to form communications among computers using the SMB protocol. Finally, we describe network security considerations when accessing files over a network using basic file I/O operations.

Universal Naming Convention

UNC paths provide a standardized way of accessing files and devices over a network without specifying or referencing a local drive letter that has been mapped to a remote file system. This is important because it allows applications to become drive-letter–independent and work seamlessly in a network environment. UNC names are better than names that reference a local drive letter because you don't have to worry about running out of drive letters when forming connections to access server shares. Drive letters also operate on a per-user basis—processes that are not running in your user context cannot access your drive mappings.

UNC names are specified as follows:

```
\\[server]\[share]\[path]
```

The first portion, \\[*server*], starts with two backslashes followed by a server name. The server name represents a remote server in which an application wants to reference a remote file. The second portion, \[*share*], represents a share point on the remote server. A *share point* is simply a directory in a file system that is identified on a network as shared for network user access. The third portion, \[*path*], represents a directory path to a file in a file system. For example, suppose you have a server named Myserver that contains a directory on a local drive named D:\Myfiles\CoolMusic that is shared out as Myshare. Let's also assume the shared directory contains a file named SAMPLE.MP3. If you would like to reference SAMPLE.MP3 from a remote machine, simply specify the UNC name \\Myserver\Myshare\Sample.mp3. As you can see, it's much easier to reference a file across a network than it is to map a local drive to the shared directory Myshare.

Referencing files over a network using UNC names hides the details of forming a connection over a network from an application. This is great—a system can easily locate network server directory shares and file paths with UNC names, even over a modem connection. All of the network communication details are handled by a network provider's redirector, which we discuss later in this chapter. As discussed in Chapters 19 and 20, the mailslot and named pipe technologies depend solely on the use of UNC names for identification.

Figure 18-1 illustrates the common components that form UNC connections on the NOS in Windows. The figure also shows how the data flows among client and server NOS components. Using the UNC path \\Myserver\ Myshare\Sample.mp3 described earlier, the remainder of this chapter describes each component and demonstrates what happens when we open this file across a network.

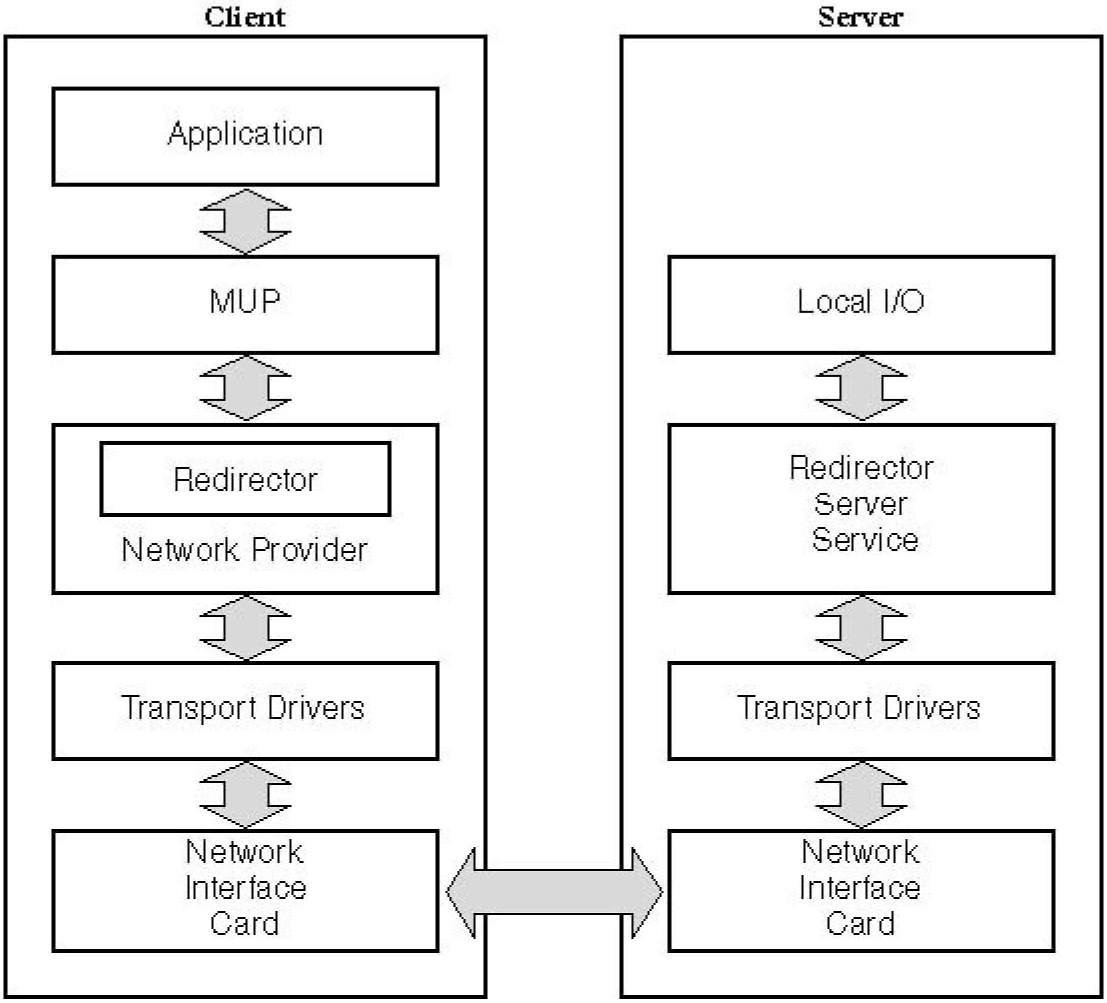


Figure 18-1 *Redirector components*

Multiple UNC Provider

MUP is a resource locator responsible for selecting a network provider to service UNC connections. A *network provider* is a service that can use network hardware to access resources—such as files and printers—located on a remote computer. MUP uses a network provider to form communications on all UNC-name–based I/O requests for files and printers. We discuss the details of a network provider later in this chapter.

Windows 95, Windows 98, and Windows NT platforms are all capable of having multiple network providers installed. For example, Windows platforms provide a network provider named Client for Microsoft Networks. It is also possible to install other third-party network providers, such as Novell's Novell Client version 3.01 for Windows. Thus, more than one network provider might be able to service a single UNC request at a time. On the other hand, Windows CE can have only one network provider, Client for Microsoft Networks.

The primary role of MUP is to decide which network provider should service a UNC request. MUP makes this decision by sending the UNC names in the request to each installed provider (in parallel). If a network provider indicates that it can service a request involving the UNC names, MUP sends the rest of the request to the provider. If more than one provider is capable of servicing a UNC request, MUP chooses the network provider with the most priority. Network provider priority is determined by the order in which providers are installed on your system. In Windows 95, Windows 98, and Windows NT, the priority can be managed by modifying the registry key ProviderOrder in the following directory in the Windows registry:

```
\HKEY_LOCAL_MACHINE
    \SYSTEM
        \CurrentControlSet
            \Control
                \NetworkProvider
                    \Order
```

The installed providers are listed first to last in order of priority. Because Windows CE can have only one provider, it does not use MUP to resolve UNC names. Instead, the UNC requests go directly to its single provider.

Network Providers

As mentioned earlier, a network provider is a service that uses network hardware to access files and printers located on a remote computer. This is considered to be the core function of a NOS. One of a provider's main capabilities is redirecting a local disk identifier—such as E:—to a disk directory located on a remote computer. Providers must also be able to service UNC connection requests. In Windows, network providers do this by exposing a redirector to the operating system.

Windows features a network provider named Client for Microsoft Networks, formally known as the Microsoft Networking Provider (MSNP). The MSNP enables communications among Windows 95, Windows 98, Windows NT, and Windows CE platforms. Windows CE, however, does not support multiple network providers and provides only built-in client-side support for the MSNP.

Redirector

A redirector is a component exposed by a network provider to an operating system that accepts and processes remote I/O service requests by formulating service request messages and sending them to a remote computer's redirector server service. The remote computer's redirector server service receives the request and services it by making local I/O requests. Because a redirector provides I/O services to higher level services such as MUP, a redirector hides the details of the network layer from applications so that applications don't have to supply protocol-specific parameters to a redirector. Thus, a network provider is protocol-independent: applications can operate in almost any network configuration.

The MSNP provides a redirector that works directly with the networking transport layer and NetBIOS to form communication between a client and a server. The NetBIOS API discussed in Chapter 17 provides a programming interface with these same transports. This redirector provided by MSNP is often referred to as the LAN manager redirector because it is designed around the old Microsoft LAN manager software that provided NOS capability to MS-DOS applications in the past. (For more detailed information about the NetBIOS programming interface, see Chapter 17.) The NetBIOS interface is capable of communicating over numerous network protocols. This makes the MSNP redirector protocol-independent: your application does not have to concern itself with the specific details of a network protocol. When your application uses the MSNP redirector, it can communicate over TCP/IP, NetBEUI, or even IPX/SPX. This is a helpful feature because it allows applications to communicate no matter what the physical network comprises. However, one important detail needs to be considered. For two applications to communicate with each other over the network, the two workstations must have at least one transport protocol in common. For example, if Workstation A has only TCP/IP installed and Workstation B has only IPX installed, the MSNP redirector will not be able to establish communication between the two workstations over a network.

The MSNP redirector communicates with other workstations by sending messages to a remote workstation's redirector server service. These messages are set up in a well-defined structure known as SMB. The actual protocol for how the redirector sends and receives messages to a remote workstation is known as the Server Message Block File Sharing Protocol, or simply the SMB protocol.

Server Message Block

The SMB protocol was originally developed by Microsoft and Intel in the late 1980s to allow remote file systems to be transparently accessed by MS-DOS applications. Today this protocol simply allows a Windows MSNP redirector to communicate with a remote workstation's MSNP server service using an SMB data structure. An SMB data structure contains three basic components: a command code, command-specific parameters, and user data.

The SMB protocol is centered on a simple client request and server response messaging model. An MSNP redirector client creates an SMB structure with a specific request indicated in the command code field. If the command requires sending data, such as an SMB Write instruction, data accompanies the request. The SMB structure is then sent over a transport protocol such as TCP/IP to a remote workstation's server service. This server service processes the client's request and transmits an SMB response data structure back to the client.

Now that we've covered the basics of the components used in forming communication through the MSNP redirector, let's follow how each component communicates when we try to open \\Myserver\Myshare\Sample.mp3 across a network. It does so by following these steps:

1. An application submits a request to the local operating system to open \\Myserver\Myshare\Sample.mp3 using the *CreateFile* API function.
2. The local operating system's file system determines that the I/O request is destined for a remote machine named \\Myserver based on the UNC path description, so it passes the request to MUP.
3. MUP determines that this I/O request is destined for the MSNP provider because the MSNP provider finds \\Myserver on the network using NetBIOS name resolution.
4. The I/O request is passed to the MSNP provider's redirector.
5. The redirector formats the I/O request as an SMB message to open the file SAMPLE.MP3 that is contained in the remote \Myshare directory.
6. The formatted SMB message is transmitted over a network transport protocol.
7. The server named \\Myserver receives the SMB request from the network and passes the request to the server's MSNP redirector server service.
8. The server's redirector server service submits a local I/O request to open the SAMPLE.MP3 file that is located on the \Myshare share point.
9. The server's redirector server service formats an SMB response message regarding the success or failure of the local file open I/O request.
10. The server's SMB response message is sent back to the client over a network transport protocol.
11. The MSNP redirector receives the server's SMB response and passes a return code back to the local operating system.
12. The local operating system returns the return code to the application *CreateFile* API request.

As you can see, the MSNP redirector must go through quite a few steps to grant applications access to remote resources. The MSNP redirector also provides access control to resources on a network as a form of network security.

Security

Our discussion of security focuses on accessing resources over a network. However, before we can discuss how security is enforced on resources over a network, we need to discuss security basics on a local machine. Windows NT platforms provide the capability to locally and remotely control access to system resources such as files and directories. These resources are considered securable objects. When an application attempts to access a securable object, the operating system checks whether an application has access rights to that object. The three basic access types are read, write, and execute privileges. Windows NT systems accomplish access control through security descriptors and access tokens.

Security Descriptors

All securable objects contain a *security descriptor* that defines their access control information. A security descriptor consists of a *SECURITY_DESCRIPTOR* structure and its associated security information, which includes the following items:

- Owner Security Identifier (SID). Represents the owner of the object.
- Group SID. Represents the primary group owner of the object.
- Discretionary Access Control List (DACL). Specifies who has what type of access to the object. Access types include read, write, and execute privileges.
- System Access Control List (SACL). Specifies the types of access attempts that generate audit records for the object.

Applications cannot directly manipulate the contents of a security descriptor structure. A descriptor can, however, be manipulated indirectly through Windows security APIs that provide functions for setting and retrieving the security information. We demonstrate this at the end of this chapter.

Access Control Lists and Access Control Entities

The DACL and SACL fields of a security descriptor are access control lists (ACLs) that contain zero or more access control entities (ACEs). Each ACE controls or monitors access to an object by a specified user or group. An ACE contains the following types of access control information:

- A SID that identifies the user or the group that the ACE applies to
- A mask that specifies access rights such as read, write, and execute privileges
- A flag that indicates ACE type (allow-access, deny-access, or system-audit)

Note that system audit ACE types are used only in SACLs, whereas allow-access and deny-access ACE types are used in DACLs. Figure 18-2 shows a file object with an associated DACL.

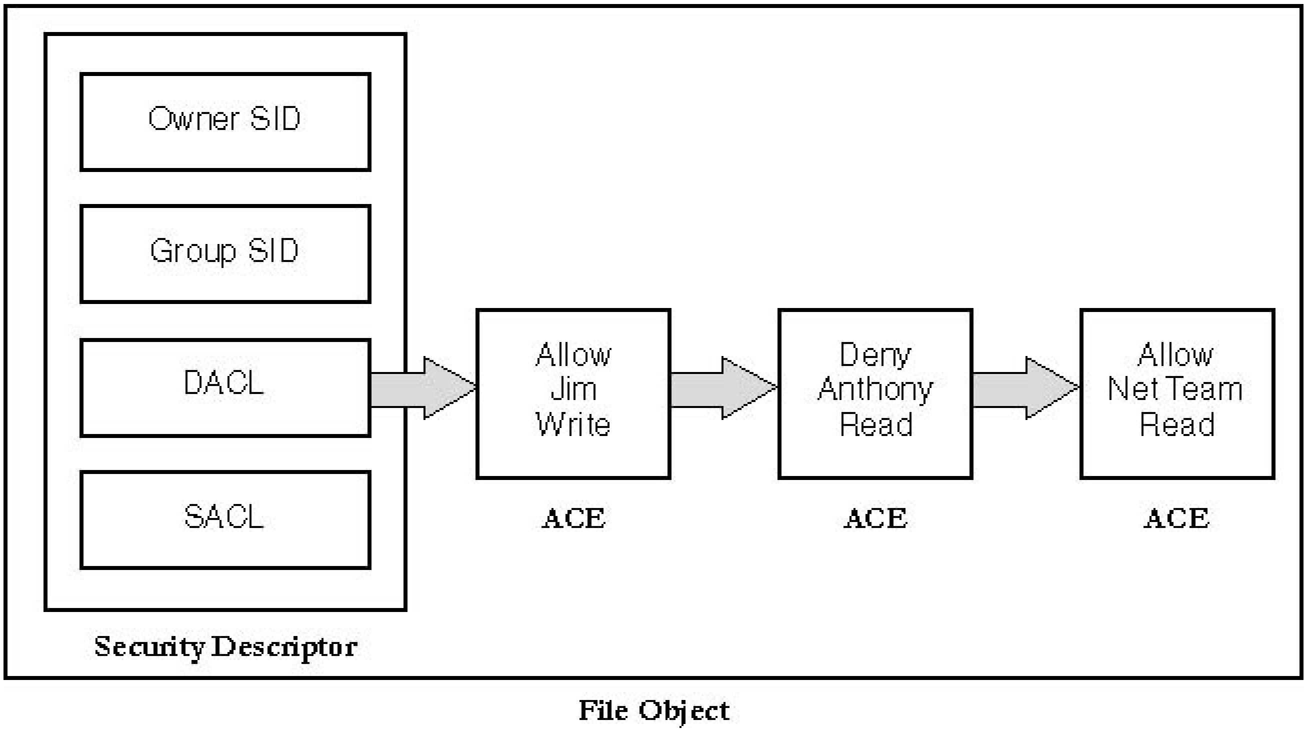


***Figure 18-2** File object with an associated DACL*

If a secured object does not have a DACL (its DACL has been set to a null value using the *SetSecurityDescriptorDacl* API function), the system allows everyone full access. If an object has a DACL, the system allows only the access that is explicitly allowed by the ACEs in the DACL. If there are no ACEs in the DACL, the system does not allow access to anyone. Similarly, if a DACL has some allow-access ACEs, the system implicitly denies access to all users and groups not included in the ACEs.

In most cases, you need to specify only allow-access ACEs, with the following exception: if you include an allow-access ACE for a group, you might have to use deny-access ACEs to exclude members of that particular group. To do this, you must place a user's deny-access ACE ahead of a group's allow-access ACE. Note that the order of the ACL is important because the system reads the ACEs in sequence until access is granted or denied. The user's access-denied ACE must appear first; otherwise, when the system reads the group's access-allowed ACE, it will grant access to the restricted user.

Figure 18-2 shows how to set up a DACL that grants read access to a group named Net Team. Let's assume that the Net Team group consists of Anthony, Jim, and Gary, and we want to grant read access to everyone in the group except Anthony. To do this, we must have a deny-access ACE for Anthony set before the allow-access ACE for the Net Team. Figure 18-2 also includes an allow-access ACE to grant Jim write access. Remember that applications do not directly manipulate ACLs; instead, they use security APIs to perform these transactions.

Security Identifiers

We have noted that security descriptors and ACEs for securable objects include a SID, which is a unique value used to identify a user account, group account, or logon session. A security authority, such as a Windows NT server domain, maintains SID information in a security account database.

When a user logs on, the system retrieves the user's SID from the database and places it in a user's access token. The system then uses the SID in the user's access token to identify the user in all subsequent interactions with Windows NT security.

Access Tokens

When a user logs on to a Windows NT system, the system authenticates the user's account name and password, which are known together as *login credentials*. If a user logs on successfully, the system creates an *access token* and assigns it the user's SID. Every process executed on behalf of this user will have a copy of this access token. When a process attempts to access a secured object, the SID in the access token is compared with access rights assigned to SIDs in DACLs.

Network Security

Now that we've briefly explained how security is enforced on a local machine, we are ready to look at security when accessing secured objects over a network. As we saw earlier, the MSNP redirector is responsible for accessing resources among computers. The MSNP redirector is also responsible for establishing a secure link between a client and a server by creating user session credentials.

Session Credentials

There are two types of user credentials: primary login and session credentials. When a user sitting in front of a workstation logs on to the machine, the user name and the password presented by the user become the primary set of credentials and are stored in an access token. Only one set of primary credentials exists at any given time. When a user attempts to establish a connection (either mapping a drive or connecting through UNC names) to a remote resource, the user's primary credentials are used to validate access to the remote resource. Note that with Windows NT systems, the user has the option of supplying a different set of credentials to be used in validating with the remote resource. If the user's credentials are valid, the MSNP redirector establishes a session between the user's computer and the remote resource. The redirector associates the session with session credentials, which consist of a copy of the credentials the user's computer used to validate the connection with the remote resource. Only one set of session credentials can be established at a time between a user's computer and a remote server. If Machine B has two share points, \Hack and \Slash, and if the user of Machine A maps \Hack to G and \Slash to H, both sessions share the same session credentials because they both refer to the same remote server.

The MSNP redirector server service handles security access control on a remote server. When the MSNP redirector server attempts to access a secured object, it uses the session credentials to create a remote access token. From there, security is managed as if the access were made locally. Figure 18-3 demonstrates how the MSNP redirector establishes security credentials using Windows NT domain security.

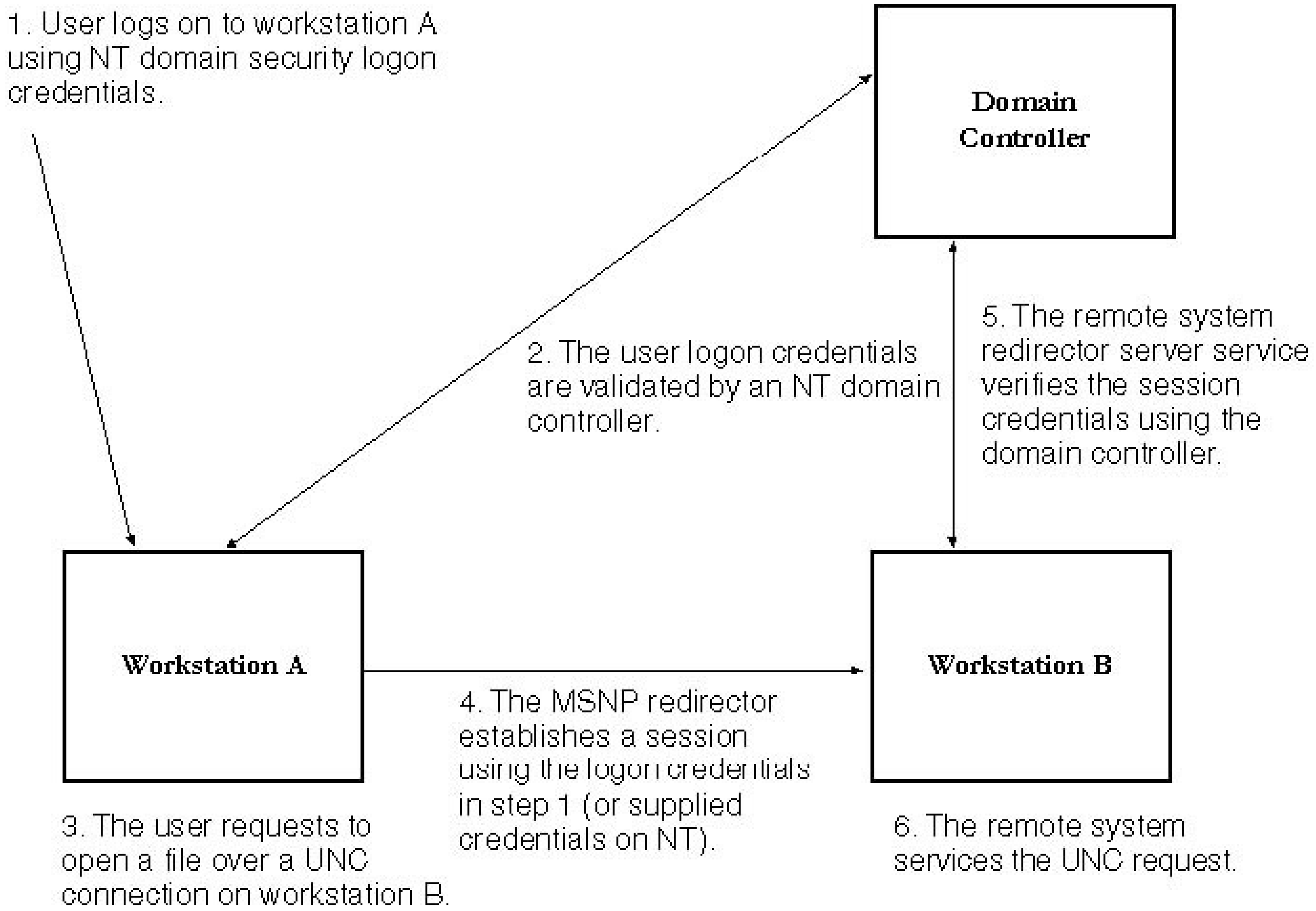

1. User logs on to workstation A using NT domain security logon credentials.
Domain Controller
2. The user logon credentials are validated by an NT domain controller.
5. The remote system redirector server service verifies the session credentials using the domain controller.
Workstation A
Workstation B
4. The MSNP redirector establishes a session using the logon credentials in step 1 (or supplied credentials on NT).
3. The user requests to open a file over a UNC connection on workstation B.
6. The remote system services the UNC request.

Figure 18-3. *Security credentials demonstration*

A Practical Example

Windows applications can use the *CreateFile*, *ReadFile*, and *WriteFile* API functions to create, access, and modify files over a network using the MSNP redirector. Windows NT systems are the only platforms that support Windows security. The following sample demonstrates how to write a simple application that will create a file over a UNC connection. You will find a file with this code called FILEIO.CPP on the companion CD.

```
#include <windows.h>
#include <stdio.h>

void main(void)
{
    HANDLE FileHandle;
    DWORD BytesWritten;

    // Open a handle to file \\Myserver\Myshare\Sample.txt
    if ((FileHandle = CreateFile("\\\\Myserver\\Myshare\\Sample.txt",
        GENERIC_WRITE | GENERIC_READ,
        FILE_SHARE_READ | FILE_SHARE_WRITE, NULL,
        CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL))
        == INVALID_HANDLE_VALUE)
    {
        printf("CreateFile failed with error %d\n", GetLastError());
        return;
    }

    // Write 14 bytes to our new file
    if (WriteFile(FileHandle, "This is a test", 14,
        &BytesWritten, NULL) == 0)
    {
        printf("WriteFile failed with error %d\n", GetLastError());
        return;
    }

    if (CloseHandle(FileHandle) == 0)
    {
        printf("CloseHandle failed with error %d\n", GetLastError());
        return;
    }
}
```

Conclusion

This chapter introduced you to the Windows redirector, which enables an application to access Windows file system resources over a network. We explained the fundamental way in which the redirector communicates over a network, followed by a discussion of security features offered by Windows NT systems when applications use the redirector. The next two chapters cover the mailslot and named pipe technologies, which depend solely on the redirector for all network communications.

Chapter 19

Mailslots

Microsoft Windows NT, Windows 95, Windows 98, and Windows Me platforms (but not Windows CE) include a simple one-way interprocess communication (IPC) mechanism known as *mailslots*. In simplest terms, mailslots allow a client process to transmit or broadcast messages to one or more server processes. Mailslots can assist transmission of messages among processes on the same computer or among processes on different computers across a network. Developing applications using mailslots is simple, requiring no formal knowledge of underlying network transport protocols such as TCP/IP or IPX. Because mailslots are designed around a broadcast architecture, you can't expect reliable data transmissions using mailslots. They can be useful, nevertheless, in certain types of network programming situations in which delivery of data isn't mission-critical.

One possible scenario for using mailslots is developing a messaging system that includes everyone in your office. Imagine that your office environment has a large number of workstations. The office is suffering from a soda shortage, and every workstation user in your office is interested in knowing every few minutes how many sodas are available in the vending machine. Mailslots lend themselves well to this type of situation. You can easily implement a mailslot client application that monitors the soda count and broadcasts to every interested workstation user the total number of available sodas at five-minute intervals. Because mailslots don't guarantee delivery of a broadcast message, some workstation users might not receive all updates. A few transmission failures won't be a problem in this case because messages sent at five-minute intervals with occasional misses are still frequent enough to keep the workstation users well informed.

The major limitation of mailslots is that they permit only unreliable one-way data communication from a client to a server. The biggest advantage of mailslots is that they allow a client application to easily send broadcast messages to one or more server applications.

This chapter explains how to develop a mailslot client/server application. We'll describe mailslot naming conventions before we address the message sizing considerations that control the overall behavior of mailslots. Next we'll show you the details of developing a basic client/server application. At the end of this chapter, we'll tell you about known problems and limitations of mailslots and offer workaround solutions.

Mailslot Implementation Details

Mailslots are designed around the Windows file system interface. Client and server applications use standard Win32 file system I/O functions, such as *ReadFile* and *WriteFile*, to send and receive data on a mailslot and take advantage of Win32 file system naming conventions. Mailslots rely on the Windows redirector to create and identify mailslots using a file system named the Mailslot File System (MSFS). Chapter 18 described the Windows redirector in greater detail.

Mailslot Names

Mailslots use the following naming convention for identification:

```
\\server\Mailslot\[path]name
```

This string is divided into three portions: \\server, *Mailslot*, and \[*path*]*name*. The first string portion, \\server, represents the name of the server on which a mailslot is created and on which a server application is running. The second portion, *Mailslot*, is a hard-coded mandatory string for notifying the system that this filename belongs to MSFS. The third portion, \[*path*]*name*, allows applications to uniquely define and identify a mailslot name; the path portion might specify multiple levels of directories. For example, the following types of names are legal for identifying a mailslot:

```
\\Oreo\Mailslot\Mymailslot
\\Testserver\Mailslot\Cooldirectory\Funtest\Anothermailslot
\\.\Mailslot\Easymailslot
\\*\Mailslot\Myslot
```

The server string portion can be represented as a dot (.), an asterisk (*), a domain name, or a server name. A domain is simply a group of workstations and servers that share a common group name. We'll examine mailslot names in greater detail later in this chapter, when we cover implementation details of a simple client.

Because mailslots rely on the Windows file system services for creation and transferring data over a network, the interface protocol is independent. When creating your application, you don't have to worry about the details of underlying network transport protocols to form communications among processes across a network. When mailslots communicate remotely to computers across a network, the Windows file system services rely on the Windows redirector to send data from a client to a server using the SMB protocol. Messages are typically sent via connectionless transfers, but you can force the Windows redirector to use connection-oriented transfers on the Windows NT platform, depending on the size of your message.

Message Sizing

Mailslots normally use *datagrams* to transmit messages over a network. Datagrams are small packets of data that are transmitted over a network in a connectionless manner. Connectionless transmission means that each data packet is sent to a recipient without packet acknowledgment. This is unreliable data transmission, so you cannot guarantee message delivery. However, connectionless transmission does give you the capability to broadcast a message from one client to many servers. The exception to this occurs on Windows NT platforms when messages exceed 424 bytes.

On Windows NT platforms, messages larger than 426 bytes are transferred using a connection-oriented protocol over an SMB session instead of using datagrams. This allows large messages to be transferred reliably and efficiently. However, you lose the ability to broadcast a message from a client to many servers. Connection-oriented transfers are limited to one-to-one communication: one client to one server. Connection-oriented transfers normally provide reliable guaranteed delivery of data between processes, but the mailslot interface on Windows NT platforms does not guarantee that a message will actually be written to a mailslot. For example, if you send a large message from a client to a server that does not exist on a network, the mailslot interface does not tell your client application that it failed to submit data to the server. Because Windows NT platforms change their transmission method based on message size, an interoperability problem occurs when you send large messages between a machine running Windows NT and a machine running Windows 95, Windows 98, or Windows Me.

Windows 95, Windows 98, and Windows Me platforms deliver messages using datagrams only, regardless of message size. If a client running one of these operating systems attempts to send a message larger than 424 bytes to a Windows NT platform, Windows NT will accept the first 424 bytes and truncate the remaining data. Windows NT expects larger messages to be sent over a connection-oriented SMB session. A similar problem exists in transferring messages from a Windows NT client to a Windows 95, Windows 98, or Windows Me server. Remember that Windows 95, Windows 98, and Windows Me receive data via datagrams only. Because Windows NT transfers data via datagrams for messages 426 bytes or smaller, Windows 95, Windows 98, and Windows Me cannot receive messages larger than 426 bytes from such clients. Table 19-1 outlines these message size limitations in detail.

Windows CE was intentionally left out of Table 19-1 because the mailslot-programming interface is not available. Also note that messages sized 425 to 426 bytes are not listed i this table due to a Windows NT redirector limitation.

Table 19-1 *Mailslot Message Size Limitations*

Transfer Direction	Connectionless Transfer Using Datagrams	Connection-Oriented Transfer
Windows 95, Windows 98, Windows Me -> Windows 95, Windows 98, Windows Me	Message size up to 64 KB.	Not supported.

Windows NT -> Windows NT	Messages must be 424 bytes or less.	Messages must be greater than 426 bytes.
Windows NT -> Windows 95, Windows 98, Windows Me	Messages must be 424 bytes or less.	Not supported.
Windows 95, Windows 98, Windows Me -> Windows NT	Messages must be 424 bytes or less; otherwise, the message is truncated.	Not supported.

Another limitation of Windows NT platforms is worth discussion because it affects datagram data transmissions. The Windows NT redirector cannot send or receive a complete datagram message of 425 or 426 bytes. For example, if you send out a message from a Windows NT client to a Windows 95, Windows 98, Windows Me, or Windows NT server, the Windows NT redirector truncates the message to 424 bytes before sending it to the destination server.

To accomplish total interoperability among all Windows platforms, we strongly recommend limiting message sizes to 424 bytes or less. If you are looking for connection-oriented transfers, consider using named pipes instead of mailslots. Named pipes are covered in Chapter 20.

Compiling Applications

When you build a mailslot client or server application using Microsoft Visual C++, your application must include the WINBASE.H include file in your program files. If you include WINDOWS.H (as most applications do) you can omit WINBASE.H. Your application is also responsible for linking with KERNEL32.LIB, which is typically configured with the Visual C++ linker flags.

Error Codes

All Win32 API functions that are used in developing mailslot client and server applications (except for *CreateFile* and *CreateMailslot*) return the value 0 when they fail. The *CreateFile* and *CreateMailslot* API functions return *INVALID_HANDLE_VALUE*. When these API functions fail, applications should call the *GetLastError* function to retrieve specific information about the failure. For a complete list of error codes, see the standard Windows error codes in Chapter 21 or consult the header file WINERROR.H.

Basic Client/Server

As we mentioned earlier, mailslots feature a simple client/server design architecture in which data can flow only from a client to a server. The data communication model is one-way, or unidirectional. The server is responsible for creating a mailslot and is the only process that can read data from it. Mailslot clients are processes that open instances of mailslots and are the only processes that can write data to them.

Mailslot Server Details

Implementing a mailslot requires developing a server application to create a mailslot. The following steps describe how to write a basic server application:

1. Create a mailslot handle using the *CreateMailslot* API function.
2. Receive data from any client by calling the *ReadFile* API function using the mailslot handle.
3. Close the mailslot handle using the *CloseHandle* API function.

As you can see, very few API calls are needed to develop a mailslot server application.

Server processes create mailslots using the *CreateMailslot* API call, which is defined as follows:

```
HANDLE CreateMailslot(
    LPCTSTR lpName,
    DWORD nMaxMessageSize,
    DWORD lReadTimeout,
    LPSECURITY_ATTRIBUTES lpSecurityAttributes
);
```

The first parameter, *lpName*, specifies the name of the mailslot. The name must have the following form:

```
\\.\Mailslot\[path]name
```

Notice that the server name is represented as a dot, which represents the local machine. This is required because you cannot create a mailslot on a remote computer. In the *lpName* parameter, *name* must represent a unique name. This might simply be a name, or a full directory path might precede it.

The *nMaxMessageSize* parameter defines the maximum size—in bytes—of a message that can be written to a mailslot. If a client writes more than *nMaxMessageSize* bytes, the server doesn't see the message. Specifying the value 0 allows the server to accept a message of any size.

Read operations can operate in blocking or nonblocking mode on a mailslot, depending on the *lReadTimeout* parameter, which determines the amount of time in milliseconds that read operations wait for incoming messages. Specifying the value *MAILSLOT_WAIT_FOREVER* allows read operations to block and wait indefinitely until incoming data is available to be read. If you specify 0, read operations return immediately. We discuss details of reading later in this chapter. The *lpSecurityAttributes* parameter determines access control rights to a mailslot. Using Windows 95, Windows 98, or Windows Me, this parameter must be *NULL* because you cannot apply security to objects. Using the Windows NT platform, this parameter is only partially implemented, so you should also specify a *NULL* parameter. The only security that you can enforce on a mailslot is for local I/O, in which a client attempts to open a mailslot with a dot (.) for the server name. A client can get around this security by specifying the server's actual name instead of a dot (.), as when making a remote I/O call. The *lpSecurityAttributes* parameter is not implemented for remote I/O on the Windows NT platform because of the extreme inefficiency of forming an authenticated session between the client and the server every time a message is sent. Mailslots, therefore, only partially follow the Windows NT security model found in the standard file systems. As a consequence, any mailslot client on your network can send data to your server.

After a mailslot is created with a valid handle, you can begin reading data. The server is the only process that can read data from a mailslot. The server should use the Win32 *ReadFile* function to accomplish this. *ReadFile* is defined as follows:

```
BOOL ReadFile(
    HANDLE hFile,
    LPVOID lpBuffer,
    DWORD nNumberOfBytesToRead,
    LPDWORD lpNumberOfBytesRead,
    LPOVERLAPPED lpOverlapped
);
```

CreateMailslot returns the handle *hFile*. The *lpBuffer* and *nNumberOfBytesToRead* parameters determine how much data can be read off a mailslot. It is important to make the size of this buffer greater than the *nMaxMessageSize* parameter from the *CreateMailslot* API call. Additionally, the buffer must be larger than incoming messages on the mailslot; if it is not larger, *ReadFile* will fail with the error *ERROR_INSUFFICIENT_BUFFER*. The *lpNumberOfBytesRead* parameter reports the actual number of bytes read when the *ReadFile* operation completes.

The *lpOverlapped* parameter provides a way to read data asynchronously off a mailslot. This parameter uses the Win32 overlapped I/O mechanism, which we describe in greater detail in Chapter 20. By default, the *ReadFile* operation blocks (waits) on I/O until data is available for reading. Overlapped I/O can be accomplished only on the Windows NT platform; you should specify *NULL* for this parameter when using Windows 95, Windows 98, or Windows Me. The following code further demonstrates how to write a simple mailslot server application.

```
// Server1.cpp

#include <windows.h>
#include <stdio.h>

void main(void)
{
    HANDLE Mailslot;
    char buffer[256];
    DWORD NumberOfBytesRead;
    // Create the mailslot
    if ((Mailslot = CreateMailslot("\\\\.\\Mailslot\\Myslot", 0,
        MAILSLOT_WAIT_FOREVER, NULL)) == INVALID_HANDLE_VALUE)
    {
        printf("Failed to create a mailslot %d\n", GetLastError());
        return;
    }

    // Read data from the mailslot forever!
    while(ReadFile(Mailslot, buffer, 256, &NumberOfBytesRead,
        NULL) != 0)
    {
        printf("%.*s\n", NumberOfBytesRead, buffer);
    }
}
```

Mailslot Client Details

Implementing a client requires developing an application to reference and write to an existing mailslot. The following steps describe how to write a basic client application:

1. Open a reference handle to the mailslot we want to send data to using the *CreateFile* API.
2. Write data to the mailslot by calling the *WriteFile* API.
3. Once you are finished writing data, close the mailslot handle using the *CloseHandle* API.

As we described earlier, mailslot clients communicate to mailslot servers in a connectionless manner. When a client opens a reference handle to a mailslot, the client does not form a connection to the mailslot server. Mailslots are referenced using the *CreateFile* API call, which is defined as follows:

```
HANDLE CreateFile(
    LPCTSTR lpFileName,
    DWORD dwDesiredAccess,
    DWORD dwShareMode,
    LPSECURITY_ATTRIBUTES lpSecurityAttributes,
    DWORD dwCreationDisposition,
    DWORD dwFlagsAndAttributes,
    HANDLE hTemplateFile
);
```

The *lpFileName* parameter describes one or more mailslots that can be written to using the mailslot name format described earlier. Table 19-2 describes mailslot naming conventions in greater detail. The *dwDesiredAccess* parameter must be set to *GENERIC_WRITE* because a client can only write data to the server. The *dwShareMode* parameter must be set to *FILE_SHARE_READ*, allowing the server to open and perform read operations on the mailslot. The *lpSecurityAttributes* parameter has no effect on mailslots and should be set to *NULL*. The *dwCreationDisposition* flag should be set to *OPEN_EXISTING*. This setting is useful when a client and a server are operating on the same machine: If the server has not created the mailslot, the *CreateFile* API function fails. The *dwCreationDisposition* parameter has no effect if the server is operating remotely. The *dwFlagsAndAttributes* parameter should be defined as *FILE_ATTRIBUTE_NORMAL*. The *hTemplateFile* parameter should be set to *NULL*.

Table 19-2 *Mailslot Name Types*

Name Format	Description
\\.\mailslot\name	Identifies a local mailslot on the same machine
\\servername\mailslot\name	Identifies a remote mailslot server named servername
\\domainname\mailslot\name	Identifies all mailslots of a particular name in the specified domain
*\mailslot\name	Identifies all mailslots of a particular name in the system's primary domain

After a handle has been successfully created, you can begin writing data to a mailslot. Remember, a client can only write data to the mailslot. This can be accomplished using the Win32 *WriteFile* function, defined as follows:

```
BOOL WriteFile(
    HANDLE hFile,
    LPCVOID lpBuffer,
    DWORD nNumberOfBytesToWrite,
    LPDWORD lpNumberOfBytesWritten,
    LPOVERLAPPED lpOverlapped
);
```

The *hFile* parameter is the reference handle that *CreateFile* returns. The *lpBuffer* and *nNumberOfBytesToWrite* parameters determine how many bytes will be sent from the client to the server. The maximum size of a message is 64 KB. If the mailslot handle was created using a domain or asterisk format, the message size is limited to 424 bytes on Windows NT and 64 KB on Windows 95, Windows 98, and Windows Me. If a client attempts to send a message that exceeds those limits, the *WriteFile* function fails and the *GetLastError* function returns *ERROR_BAD_NETPATH*. This happens because the message is sent as a broadcast datagram to all servers on the network. The *lpNumberOfBytesWritten* parameter returns the number of bytes sent to a server when the *WriteFile* operation completes.

The *lpOverlapped* parameter provides a way to write data asynchronously to a mailslot. Because mailslots feature connectionless data transfer, the *WriteFile* function is not subject to blocking on I/O calls. This parameter should be set to *NULL* on the client. The following code further demonstrates how to write a simple mailslot client application.

```
// Client.cpp

#include <windows.h>
#include <stdio.h>

void main(int argc, char *argv[])
{
    HANDLE Mailslot;
    DWORD BytesWritten;
    CHAR ServerName[256];

    // Accept a command line argument for the server to send
    // a message to
    if (argc < 2)
    {
        printf("Usage: client <server name>\n");
        return;
    }
    sprintf(ServerName, "\\\\%s\\Mailslot\\Myslot", argv[1]);

    if ((Mailslot = CreateFile(ServerName, GENERIC_WRITE,
        FILE_SHARE_READ, NULL, OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL,
        NULL)) == INVALID_HANDLE_VALUE)
    {
        printf("CreateFile failed with error %d\n", GetLastError());
        return;
    }

    if (WriteFile(Mailslot, "This is a test", 14, &BytesWritten,
        NULL) == 0)
    {
        printf("WriteFile failed with error %d\n", GetLastError());
        return;
    }

    printf("Wrote %d bytes\n", BytesWritten);

    CloseHandle(Mailslot);
}
```

Additional Mailslot APIs

A mailslot server application can use two additional API functions to interact with a mailslot: *GetMailslotInfo* and *SetMailslotInfo*. The *GetMailslotInfo* function retrieves message sizing information when messages become available on a mailslot. Applications can use this to dynamically adjust their buffers for incoming messages of varying length. *GetMailslotInfo* can also be used to poll for incoming data. *GetMailslotInfo* is defined as follows:

```
BOOL GetMailslotInfo(
    HANDLE hMailslot,
    LPDWORD lpMaxMessageSize,
    LPDWORD lpNextSize,
    LPDWORD lpMessageCount,
    LPDWORD lpReadTimeout
);
```

The *hMailslot* parameter identifies a mailslot returned from the *CreateMailslot* API call. The *lpMaxMessageSize* parameter points to how large a message (in bytes) can be written to the mailslot. The *lpNextSize* parameter points to the size in bytes of the next message. *GetMailslotInfo* might return the value *MAILSLOT_NO_MESSAGE*, indicating that no message is currently waiting to be received on the mailslot. A server can potentially use this parameter to poll the mailslot for incoming data, preventing your application from blocking on a *ReadFile* function call. Polling for data using this mechanism is not a good programming approach. Your application will continuously use the computer's CPU to check for incoming data—even when no messages are being processed—resulting in a slower overall performance by the computer. If you want to prevent the *ReadFile* function from blocking, we recommend using Win32 overlapped I/O. The *lpMesssageCount* parameter points to a buffer that receives the total number of messages waiting to be read. You can use this parameter for polling purposes, too. The *lpReadTimeout* parameter points to a buffer that returns the amount of time in milliseconds that a read operation can wait for a message to be written to the mailslot before a timeout occurs.

The *SetMailslotInfo* API function sets the timeout values on a mailslot for how long read operations wait for incoming messages. Thus the application has the ability to change the read behavior from blocking to nonblocking mode or vice versa. *SetMailslotInfo* is defined as follows:

```
BOOL SetMailslotInfo(
    HANDLE hMailslot,
    DWORD lReadTimeout
);
```

The *hMailslot* parameter identifies a mailslot that is returned from the *CreateMailslot* API call. The *lReadTimeout* parameter specifies the amount of time in milliseconds that a read operation can wait for a message to be written to the mailslot before a timeout occurs. If you specify 0, read operations will return immediately if no message is present. If you specify *MAILSLOT_WAIT_FOREVER*, read operations will wait forever.

Platform and Performance Considerations

Mailslots on Windows 95, Windows 98, and Windows Me platforms have three limitations that you should be aware of: 8.3-character name limits, inability to cancel blocking I/O requests, and timeout memory leaks.

8.3-Character Name Limits

Windows 95, Windows 98, and Windows Me platforms silently limit mailslot names to an 8.3-character name format. This causes interoperability problems between Windows 95, Windows 98, Windows Me, and Windows NT. For example, if you create or open a mailslot with the name \\.\Mailslot\Mymailslot, Windows 95, Windows 98, and Windows Me will actually create and reference the mailslot as \\.\Mailslot\Mymailsl. The *CreateMailslot* and *CreateFile* functions succeed even though name truncation occurs. If a message is sent from Windows NT to Windows 95, Windows 98, or Windows Me, or vice versa, the message will not be received because the mailslot names do not match. If both the client and the server are running on Windows 95, Windows 98, or Windows Me machines, there isn't a problem—the name is truncated on both the client and the server. An easy way to prevent interoperability problems is to limit mailslot names to eight characters or less.

Inability to Cancel Blocking I/O Requests

Windows 95, Windows 98, and Windows Me platforms also have a problem with canceling blocking I/O requests. Mailslot servers use the *ReadFile* function to receive data. If a mailslot is created with the *MAILSLOT_WAIT_FOREVER* flag, read requests block indefinitely until data is available. If a server application is terminated when there is an outstanding *ReadFile* request, the application hangs forever. The only way to cancel the application is to reboot Windows. A possible solution is to have the server open a handle to its own mailslot in a separate thread and send data to break the blocking read request. The following code demonstrates this solution in detail:

```
// Server2.cpp

#include <windows.h>
#include <stdio.h>
#include <conio.h>

BOOL StopProcessing;

DWORD WINAPI ServeMailslot(LPVOID lpParameter);
void SendMessageToMailslot(void);

void main(void) {

    DWORD ThreadId;
    HANDLE MailslotThread;
    StopProcessing = FALSE;
    MailslotThread = CreateThread(NULL, 0, ServeMailslot, NULL,
        0, &ThreadId);

    printf("Press a key to stop the server\n");
    _getch();

    // Mark the StopProcessing flag to TRUE so that when ReadFile
    // breaks, our server thread will end
    StopProcessing = TRUE;

    // Send a message to our mailslot to break the ReadFile call
    // in our server
     SendMessageToMailslot();

    // Wait for our server thread to complete
    if (WaitForSingleObject(MailslotThread, INFINITE) == WAIT_FAILED)
    {
        printf("WaitForSingleObject failed with error %d\n",
            GetLastError());
        return;
    }
}

//
// Function: ServeMailslot
//
// Description:
//     This function is the mailslot server worker function to
//     process all incoming mailslot I/O
//
DWORD WINAPI ServeMailslot(LPVOID lpParameter)
{
    char buffer[2048];
    DWORD NumberOfBytesRead;
    DWORD Ret;
    HANDLE Mailslot;

    if ((Mailslot = CreateMailslot("\\\\.\\mailslot\\myslot", 2048,
        MAILSLOT_WAIT_FOREVER, NULL)) == INVALID_HANDLE_VALUE)
    {
        printf("Failed to create a MailSlot %d\n", GetLastError());
        return 0;
    }

    while((Ret = ReadFile(Mailslot, buffer, 2048,
        &NumberOfBytesRead, NULL)) != 0)
    {
        if (StopProcessing)
            break;

        printf("Received %d bytes\n", NumberOfBytesRead);
    }

    CloseHandle(Mailslot);

    return 0;
}
```

```
//
// Function: SendMessageToMailslot
//
// Description:
//     The SendMessageToMailslot function is designed to send a
//     simple message to our server so we can break the blocking
//     ReadFile API call
//
void SendMessageToMailslot(void)
{
    HANDLE Mailslot;
    DWORD BytesWritten;

    if ((Mailslot = CreateFile("\\\\.\\mailslot\\myslot",
        GENERIC_WRITE, FILE_SHARE_READ, NULL, OPEN_EXISTING,
        FILE_ATTRIBUTE_NORMAL, NULL)) == INVALID_HANDLE_VALUE)
    {
        printf("CreateFile failed with error %d\n", GetLastError());
        return;
    }

    if (WriteFile(Mailslot, "STOP", 4, &BytesWritten, NULL) == 0)
    {
        printf("WriteFile failed with error %d\n", GetLastError());
        return;
    }

    CloseHandle(Mailslot);
}
```

Timeout Memory Leaks

The final problem with Windows 95, Windows 98, and Windows Me platforms worth mentioning is memory leaks, which can occur when you're using timeout values on mailslots. When you create a mailslot using the *CreateMailslot* function with a timeout value greater than 0, the *ReadFile* function leaks memory when the timeout expires and the function returns *FALSE*. After many calls to the *ReadFile* function, the system becomes unstable and subsequent *ReadFile* calls with timers that expire start returning *TRUE*. As a result, the system is no longer able to execute other MS-DOS applications. To work around this, create the mailslot with a timeout value of either 0 or *MAILSLOT_WAIT_FOREVER*. This prevents an application from using the timeout mechanism, which causes the actual memory leak.

The Microsoft knowledge base documents the following problems and limitations. You can access the knowledge base at http://support.microsoft.com/support/search. We briefly describe each issue here.

- Q139715 *ReadFile* Returns Wrong Error Code for Mailslots

  If a server opens a mailslot using *CreateMailslot*, specifies a timeout, and then uses *ReadFile* to receive data, the *ReadFile* fails if no data is available. *GetLastError* returns an error code of 5 (access denied).

- Q192276 *GetMailslotInfo* Returns Incorrect *lpNextSize* Value

If you call the API function *GetMailslotInfo* under Windows 95 OEM Service Release 2 (OSR2) or Windows 98 without a network client component installed, you receive an incorrect value (usually in the millions) or a negative number for the *lpNextSize* parameter. If you repeatedly call the function, it usually returns the correct value.

- Q170581 *Mailslot* Created on Win95 Allows Only 4093 Bytes

  If you call the *WriteFile* API function to write more than 4093 bytes to a mailslot that has been created on a Windows 95 workstation, it fails.

- Q131493 *CreateFile* and Mailslots

  The documentation for the *CreateFile* API function incorrectly describes the possible values that *CreateFile* returns when opening a client end of a mailslot.

Conclusion

This chapter introduced the mailslot networking technology, which provides an application with simple one-way interprocess data communication using the Windows redirector. One of the most useful features of mailslots is that they allow you to broadcast a message to one or more computers over a network. However, because of the broadcast capability, mailslots do not provide reliable data transmission. If you want reliable data communication using the Windows redirector, consider using named pipes—the focus of our next chapter.

Chapter 20

Named Pipes

Named pipes are a simple interprocess communication (IPC) mechanism included in Microsoft Windows NT, and Windows 95, Windows 98, and Windows Me platforms (but not Windows CE). Named pipes provide reliable one-way and two-way data communications among processes on the same computer or among processes on different computers across a network. Developing applications using named pipes is actually quite simple and requires no formal knowledge of underlying network transport protocols (such as TCP/IP or IPX). This is because named pipes use the Microsoft Network Provider (MSNP) redirector to form communication among processes over a network, thus hiding network protocol details from the application. One of the best reasons for using named pipes as a networking communication solution is that they take advantage of security features built into the Windows NT platform.

One possible scenario for using named pipes is developing a data management system that allows only a select group of people to perform transactions. Imagine an office setting in which you have a computer that contains company secrets. You need to have these secrets accessed and maintained by management personnel only. Let's say every employee can see the computer on the network from his or her workstation. However, you do not want regular employees to obtain access to the confidential records. Named pipes work well in this situation because you can develop a server application that, based on requests from clients, safely performs transactions on the company secrets. The server can easily limit client access to management personnel by using security features of the Windows NT platform.

What's important to remember when using named pipes as a network programming solution is that they feature a simple client/server data communication architecture that reliably transmits data. This chapter explains how to develop named pipe client and server applications. We start by explaining named pipe naming conventions, followed by basic pipe types. We'll then show how to implement a basic server application, followed by advanced server programming details. Next we discuss how to develop a basic client application. By the chapter's end, we uncover the known problems and limitations of named pipes.

Named Pipe Implementation Details

Named pipes are designed around the Windows file system using the Named Pipe File System (NPFS) interface. As a result, client and server applications use standard Windows file system API functions such as *ReadFile* and *WriteFile* to send and receive data. Using these API functions allows applications to take advantage of Windows file system naming conventions and Windows NT file system security. NPFS relies on the MSNP redirector to send and receive named pipe data over a network. This makes the interface protocol-independent: when developing an application that uses named pipes to form communications among processes across a network, so a programmer does not have to worry about the details of underlying network transport protocols, such as TCP and IPX. Named pipes are identified to NPFS using the Universal Naming Convention. Chapter 18 describes the UNC, the Windows redirector, and security in greater detail.

Named Pipe Naming Conventions

Named pipes are identified using the following UNC format:

```
\\server\Pipe\[path]name
```

This string is divided into three parts: \\server, *Pipe*, and \[*path*]*name*. The first string part, \\server, represents the server name in which a named pipe is created and the server that listens for incoming connections. The second part, *Pipe*, is a hard-coded mandatory string requirement for identifying that this filename belongs to NPFS. The third part, \[*path*]*name*, allows applications to uniquely define and identify a named pipe name, and it can have multiple levels of directories. For example, the following name types are legal for identifying a named pipe:

```
\\myserver\PIPE\mypipe
\\Testserver\pipe\cooldirectory\funtest\jim
\\.\Pipe\Easynamedpipe
```

The server string portion can be represented as a dot (.) or a server name.

Byte Mode and Message Mode

Named pipes offer two basic communication modes: byte mode and message mode. In byte mode, messages travel as a continuous stream of bytes between the client and the server. This means that a client application and a server application do not know precisely how many bytes are being read from or written to a pipe at any given moment. Therefore a write on one side will not always result in a same-size read on the other. This allows a client and a server to transfer data without regard to the contents of the data. In message mode, the client and the server send and receive data in discrete units. Every time a message is sent on the pipe, it must be read as a complete message. Figure 20-1 compares the two pipe modes.

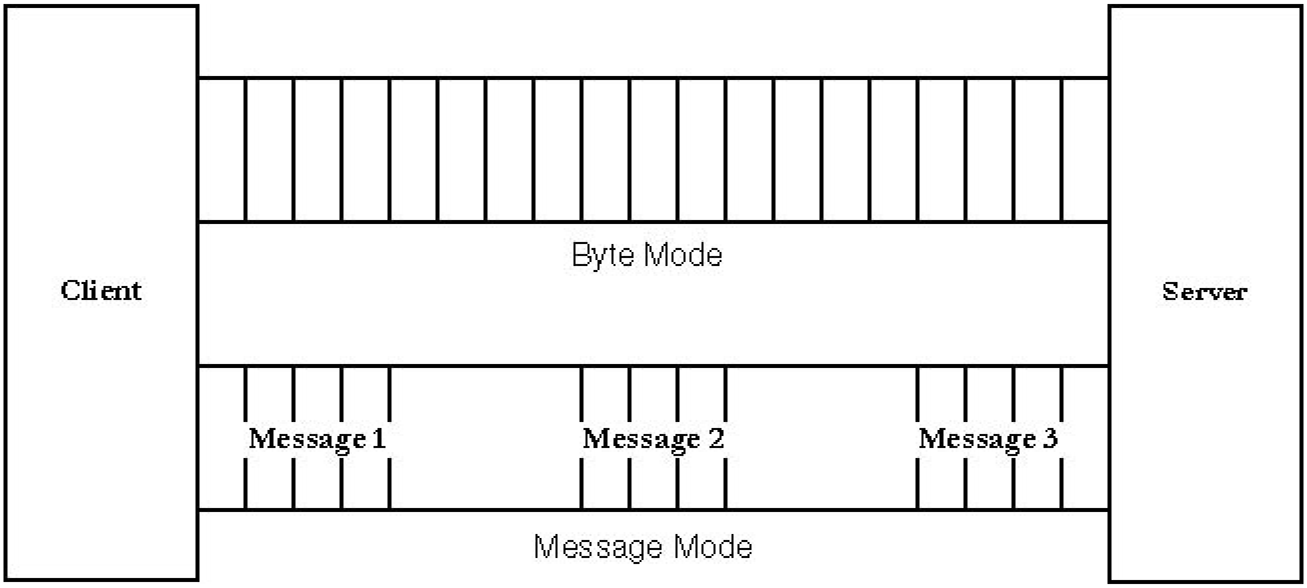


***Figure 20-1** Byte mode and message mode*

Compiling Applications

When you build a named pipe client or server application using Microsoft Visual C++, your application must include the WINBASE.H file in your program files. If your application includes WINDOWS.H—as most do—you can omit WINBASE.H. Your application is also responsible for linking with KERNEL32.LIB, which typically is configured with the Visual C++ linker flags.

Error Codes

All Windows API functions (except *CreateFile* and *CreateNamedPipe*) that are used in developing named pipe client and server applications return the value 0 when they fail. *CreateFile* and *CreateNamedPipe* return *INVALID_HANDLE_VALUE*. When either of these functions fails, applications should call the *GetLastError* function to retrieve specific information about the failure. For a complete list of error codes, consult the header file WINERROR.H.

Basic Server and Client

Named pipes feature a simple client/server design architecture in which data can flow in both a unidirectional and a bidirectional manner between a client and server. This is useful because it allows you to send and receive data whether your application is a client or a server. The main difference between a named pipe server and a client application is that a named pipe server is the only process capable of creating a named pipe and accepting pipe client connections. A client application is capable only of connecting to an existing named pipe server. Once a connection is formed between a client application and a server application, both processes are capable of reading and writing data on a pipe using standard Windows functions such as *ReadFile* and *WriteFile*. Note that a named pipe server application can operate only on the Windows NT platform—Windows 95, Windows 98, and Windows Me systems do not permit applications to create a named pipe. This limitation makes it impossible to form communications directly between two Windows 95, Windows 98, or Windows Me computers. However, Windows 95, Windows 98, and Windows Me clients can form connections to Windows NT–based computers.

Server Details

Implementing a named pipe server requires developing an application to create one or more named pipe instances, which can be accessed by clients. To a server, a pipe instance is nothing more than a handle used to accept a connection from a local or remote client application. The following steps describe how to write a basic server application:

1. Create a named pipe instance handle using the *CreateNamedPipe* API function.
2. Use the *ConnectNamedPipe* API function to listen for a client connection on the named pipe instance.
3. Receive data from and send data to the client using the *ReadFile* and *WriteFile* API functions.
4. Close down the named pipe connection using the *DisconnectNamed Pipe* API function.
5. Close the named pipe instance handle using the *CloseHandle* API function.

First, your server process needs to create a named pipe instance using the *CreateNamedPipe* API call, which is defined as follows:

```
HANDLE CreateNamedPipe(
    LPCTSTR lpName,
    DWORD dwOpenMode,
    DWORD dwPipeMode,
    DWORD nMaxInstances,
    DWORD nOutBufferSize,
    DWORD nInBufferSize,
    DWORD nDefaultTimeOut,
    LPSECURITY_ATTRIBUTES lpSecurityAttributes
);
```

The first parameter, *lpName*, specifies the name of a named pipe. The name must have the following UNC form:

```
\\.\Pipe\[path]name
```

Notice that the server name is represented as a dot, which represents the local machine. You cannot create a named pipe on a remote computer. The [*path*]*name* part of the parameter must represent a unique name. This might simply be a filename, or it might be a full directory path followed by a filename.

The *dwOpenMode* parameter describes the directional, I/O control, and security modes of a pipe when it is created. Table 20-1 describes all the available flags that can be used. A pipe can be created using a combination of these flags by *OR*ing them together.

Table 20-1 *Named Pipe Open Mode Flags*

Open Mode	Flags	Description
Directional	*PIPE_ACCESS_DUPLEX*	The pipe is bidirectional: Both the server and client processes can read from and write data to the pipe.
	PIPE_ACCESS_OUTBOUND	The flow of data in the pipe goes from server to client only.
	PIPE_ACCESS_INBOUND	The flow of data in the pipe goes from client to server only.
I/O Control	*FILE_FLAG_WRITE_THROUGH*	Works only for byte-mode pipes. Functions writing to a named pipe do not return until the data written is transmitted across the network and is in the pipe's buffer on the remote computer.
I/O control	*FILE_FLAG_OVERLAPPED*	Allows functions that perform read, write, and connect operations to use overlapped I/O.
Security	*WRITE_DAC*	Allows your application to have write access to the named pipe's DACL.
Security	*ACCESS_SYSTEM_SECURITY*	Allows your application to have write access to the named pipe's SACL.
	WRITE_OWNER	Allows your application to have write access to the named pipe's owner and group SID.

The *PIPE_ACCESS_* flags determine flow direction on a pipe between a client and a server. A pipe can be opened as bidirectional (two-way) using the *PIPE_ACCESS_DUPLEX* flag: Data can flow in both directions between the client and the server. In addition, you can also control the direction of data flow by opening the pipe as unidirectional (one-way) using the flag *PIPE_ACCESS_INBOUND* or *PIPE_ACCESS_OUTBOUND*: data can flow only one way from the client to the server or vice versa. Figure 20-2 describes the flag combinations further and shows the flow of data between a client and a server.

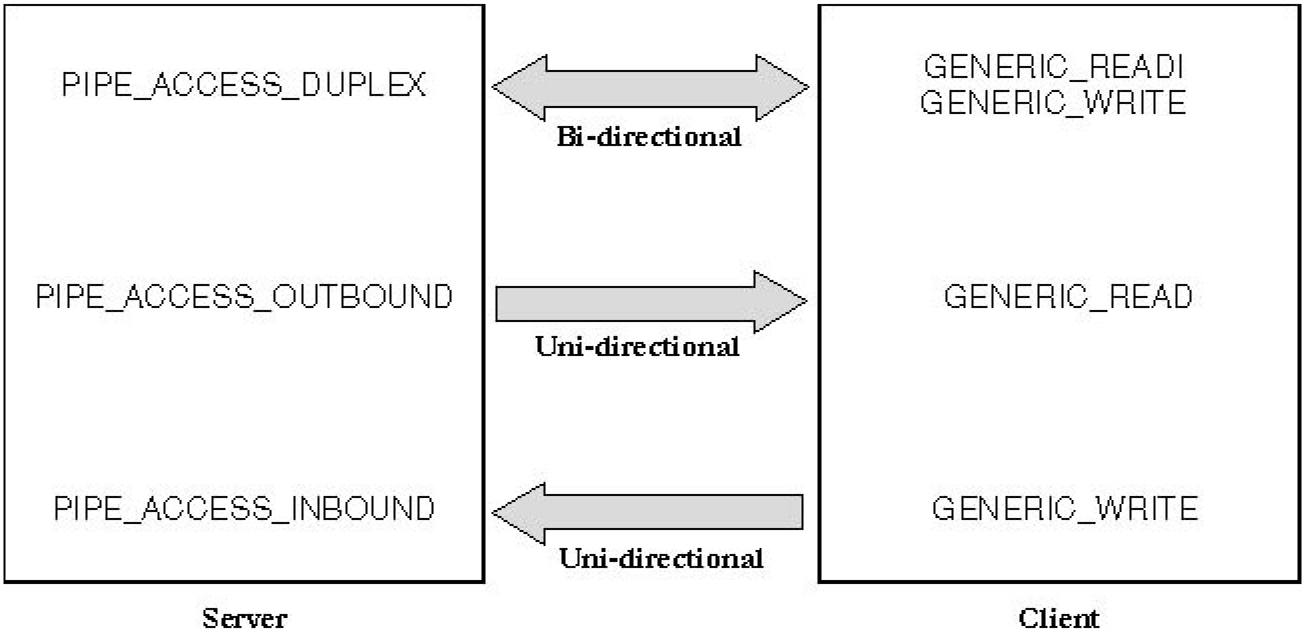


Figure 20-2 *Mode flags and flow direction*

The next set of *dwOpenMode* flags controls I/O behavior on a named pipe from the server's perspective. The *FILE_FLAG_WRITE_THROUGH* flag controls the write operations so that functions writing to a named pipe do not return until the data written is transmitted across the network and is in the pipe's buffer on the remote computer. This flag works only for byte-mode named pipes when the client and the server are on different computers. The *FILE_FLAG_OVERLAPPED* flag allows functions performing read, write, and connect operations to return immediately, even if those functions take significant time to complete. We discuss the details of overlapped I/O when we develop an advanced server later in this chapter.

The last set of *dwOpenMode* flags described in Table 20-1 controls the server's ability to access the security descriptor that is created by a named pipe. If your application needs to modify or update the pipe's security descriptor after the pipe is created, you should set these flags accordingly to permit access. The *WRITE_DAC* flag allows your application to update the pipe's DACL, whereas *ACCESS_SYSTEM_SECURITY* allows access to the pipe's SACL. The *WRITE_OWNER* flag allows you to change the pipe's owner and group SID. For example, if you want to deny access to a particular user who has access rights to your pipe, you can modify the pipe's DACL using security API functions. Chapter 18 discusses DACLs, SACLs, and SIDs in greater detail.

CreateNamedPipe's *dwPipeMode* parameter specifies the read, write, and wait operating modes of a pipe. Table 20-2 describes all the available mode flags that can be used. The flags can be issued by *OR*ing one flag from each mode category. If a pipe is opened as byte-oriented using the *PIPE_READMODE_BYTE | PIPE_TYPE_BYTE* mode flags, data can be read and written only as a stream of bytes. This means that when you read and write data to a pipe, you do not have to balance each read and write because your data does not have any message boundaries. For example, if a sender writes 500 bytes to a pipe, a receiver might want to read 100 bytes at a time until it receives all of the data. To establish clear boundaries around messages, place the pipe in message-oriented mode using the flags *PIPE_READMODE_MESSAGE | PIPE_TYPE_MESSAGE*, meaning each read and write must be balanced. For example, if a sender writes a 500-byte message to a pipe, the receiver must provide the *ReadFile* function a 500-byte or larger buffer when reading data. If the receiver fails to do so, *ReadFile* will fail with error *ERROR_MORE_DATA*. You can also combine

PIPE_TYPE_MESSAGE with *PIPE_READMODE_BYTE*, allowing a sender to write messages to a pipe and the receiver to read an arbitrary amount of bytes at a time. The message delimiters will be ignored in the data stream. You cannot mix the *PIPE_TYPE_BYTE* flag with the *PIPE_READMODE_MESSAGE* flag. Doing so will cause the *CreateNamedPipe* function to fail with the error *ERROR_INVALID_PARAMETER* because no message delimiters are in the I/O stream when data is written into the pipe as bytes. The *PIPE_WAIT* or *PIPE_NOWAIT* flag can also be combined with read and write mode flags. The *PIPE_WAIT* flag places a pipe in blocking mode and the *PIPE_NOWAIT* flag places a pipe in nonblocking mode. In blocking mode, I/O operations such as *ReadFile* block until the I/O request is complete. This is the default behavior if you do not specify any flags. The nonblocking mode flag *PIPE_NOWAIT* is designed to allow I/O operations to return immediately. However, it should not be used to achieve asynchronous I/O in Windows applications. It is included to provide backward compatibility with older Microsoft LAN Manager 2.0 applications. The *ReadFile* and *WriteFile* functions allow applications to accomplish asynchronous I/O using Windows overlapped I/O, which is demonstrated later in this chapter.

Table 20-2 *Named Pipe Read/Write Mode Flags*

Mode	Flags	Description
Write	*PIPE_TYPE_BYTE*	Data is written to the pipe as a stream of bytes.
	PIPE_TYPE_MESSAGE	Data is written to the pipe as a stream of messages.
Read	*PIPE_READMODE_BYTE*	Data is read from the pipe as a stream of bytes.
	PIPE_READMODE_MESSAGE	Data is read from the pipe as a stream of messages.
Wait	*PIPE_WAIT*	Blocking mode is enabled.
	PIPE_NOWAIT	Nonblocking mode is enabled.

The *PIPE_NOWAIT* flag is obsolete and should not be used in Windows environments to accomplish asynchronous I/O. It is included in this book to provide backward compatibility with older Microsoft LAN Manager 2.0 software.

The *nMaxInstances* parameter specifies how many instances or pipe handles can be created for a named pipe. A *pipe instance* is a connection from a local or remote client application to a server application that created the named pipe. Acceptable values are in the range 1 through *PIPE_UNLIMITED_INSTANCES*. For example, if you want to develop a server that can service only five client connections at a time, set this parameter to 5. If you set this parameter to *PIPE_UNLIMITED_INSTANCES*, the number of pipe instances that can be created is limited only by the availability of system resources.

CreateNamedPipe's *nOutBufferSize* and *nInBufferSize* parameters represent the number of bytes to reserve for internal input and output buffer sizes. These sizes are advisory in that every time a named pipe instance is created, the system sets up inbound and/or outbound buffers using the nonpaged pool (the physical memory used by the operating system). The buffer size specified should be reasonable (not too large) so that your system will not run out of nonpaged pool memory, but it should also be large enough to accommodate typical I/O requests. If an application attempts to write data that is larger than the buffer sizes specified, the system tries to automatically expand the buffers to accommodate the data using nonpaged pool memory. For practical purposes, applications should size these internal buffers to match the size of the application's send and receive buffers used when calling *ReadFile* and *WriteFile*.

The *nDefaultTimeOut* parameter specifies the default timeout value (how long a client will wait to connect to a named pipe) in milliseconds. This affects only client applications that use the *WaitNamedPipe* function to determine when an instance of a named pipe is available to accept connections. We discuss this concept in greater detail later in this chapter, when we develop a named pipe client application.

The *lpSecurityAttributes* parameter allows the application to specify a security descriptor for a named pipe and determines whether a child process can inherit the newly created handle. If this parameter is specified as *NULL*, the named pipe gets a default security descriptor and the handle cannot be inherited. A default security descriptor grants the named pipe the same security limits and access controls as the process that created it following the Windows NT platform security model described in Chapter 18. An application can apply access control restrictions to a pipe by setting access privileges for particular users and groups in a *SECURITY_DESCRIPTOR* structure using security API functions. If a server wants to open access to any client, you should assign a null DACL to the *SECURITY_DESCRIPTOR* structure.

After you successfully receive a handle from *CreateNamedPipe*, which is known as a pipe instance, you have to wait for a connection from a named pipe client. This connection can be made through the *ConnectNamedPipe* API function, which is defined as follows:

```
BOOL ConnectNamedPipe(
    HANDLE hNamedPipe,
    LPOVERLAPPED lpOverlapped
);
```

The *hNamedPipe* parameter represents the pipe instance handle returned from *CreateNamedPipe*. The *lpOverlapped* parameter allows this API function to operate asynchronously, or in nonblocking mode, if the pipe was created using the *FILE_FLAG_OVERLAPPED* flag, which is known as Windows overlapped I/O. If this parameter is specified as *NULL*, *ConnectNamedPipe* blocks until a client forms a connection to the server. We discuss overlapped I/O in greater detail when you learn to create a more advanced named pipe server later in this chapter.

Once a named pipe client successfully connects to your server, the *ConnectNamedPipe* API call completes. The server is then free to send data to a client using the *WriteFile* API function and to receive data from the client using *ReadFile*. Once the server has finished communicating with a client, it should call *DisconnectNamedPipe* to close the communication session. The following sample demonstrates how to write a simple server application that can communicate with one client.

```
// Server.cpp

#include <windows.h>
#include <stdio.h>
```

```
void main(void)
{
    HANDLE PipeHandle;
    DWORD BytesRead;
    CHAR buffer[256];
    if ((PipeHandle = CreateNamedPipe("\\\\.\\Pipe\\Jim",
        PIPE_ACCESS_DUPLEX, PIPE_TYPE_BYTE | PIPE_READMODE_BYTE, 1,e
        0, 0, 1000, NULL)) == INVALID_HANDLE_VALUE)
    {
        printf("CreateNamedPipe failed with error %d\n",
            GetLastError());
        return;
    }

    printf("Server is now running\n");

    if (ConnectNamedPipe(PipeHandle, NULL) == 0)
    {
        printf("ConnectNamedPipe failed with error %d\n",
            GetLastError());
        CloseHandle(PipeHandle);
        return;
    }

    if (ReadFile(PipeHandle, buffer, sizeof(buffer),
        &BytesRead,  NULL) <= 0)
    {
        printf("ReadFile failed with error %d\n", GetLastError());
        CloseHandle(PipeHandle);
        return;
    }

    printf("%.*s\n", BytesRead, buffer);

    if (DisconnectNamedPipe(PipeHandle) == 0)
    {
        printf("DisconnectNamedPipe failed with error %d\n",
            GetLastError());
        return;
    }

    CloseHandle(PipeHandle);
}
```

Building Null DACLs

When applications create securable objects such as files and named pipes on the Windows NT platform using Windows API functions, the operating system grants the applications the ability to set up access control rights by specifying a *SECURITY_ATTRIBUTES* structure, defined as follows:

```
typedef struct _SECURITY_ATTRIBUTES {
    DWORD   nLength;
    LPVOID  lpSecurityDescriptor;
    BOOL    bInheritHandle
} SECURITY_ATTRIBUTES;
```

The *lpSecurityDescriptor* field defines the access rights for an object in a *SECURITY_DESCRIPTOR* structure. A *SECURITY_DESCRIPTOR* structure contains a DACL field that defines which users and groups can access the object. If you set this field to *NULL*, any user or group can access your resource.

Applications cannot directly access a *SECURITY_DESCRIPTOR* structure and must use Windows security API functions to do so. If you want to assign a null DACL to a *SECURITY_DESCRIPTOR* structure, you must do the following:

1. Create and initialize a *SECURITY_DESCRIPTOR* structure by calling the *InitializeSecurityDescriptor* API function.
2. Assign a null DACL to the *SECURITY_DESCRIPTOR* structure by calling the *SetSecurityDescriptorDacl* API function.

After you successfully build a new *SECURITY_DESCRIPTOR* structure, you must assign it to the *SECURITY_ATTRIBUTES* structure. Now you are ready to begin calling Windows functions such as *CreateNamedPipe* with your new *SECURITY_ATTRIBUTES* structure, which contains a null DACL. The following code fragment demonstrates how to call the security API functions needed to accomplish this:

```
// Create new SECURITY_ATTRIBUTES and SECURITY_DESCRIPTOR
// structure objects
SECURITY_ATTRIBUTES sa;
SECURITY_DESCRIPTOR sd;

// Initialize the new SECURITY_DESCRIPTOR object to empty values
if (InitializeSecurityDescriptor(&sd, SECURITY_DESCRIPTOR_REVISION)
    == 0)
{
    printf("InitializeSecurityDescriptor failed with error %d\n",
        GetLastError());
    return;
}

// Set the DACL field in the SECURITY_DESCRIPTOR object to NULL
if (SetSecurityDescriptorDacl(&sd, TRUE, NULL, FALSE) == 0)
{
    printf("SetSecurityDescriptorDacl failed with error %d\n",
        GetLastError());
    return;
}

// Assign the new SECURITY_DESCRIPTOR object to the
// SECURITY_ATTRIBUTES object
sa.nLength = sizeof(SECURITY_ATTRIBUTES);
sa.lpSecurityDescriptor = &sd;
sa.bInheritHandle = TRUE;
```

Advanced Server

The previous sample demonstrates how to develop a named pipe server application that handles only a single pipe instance. All of the API calls operate in a synchronous mode in which each call waits until an I/O request is complete. A named pipe server is also capable of having multiple pipe instances so that clients can form two or more connections to the server; the number of pipe instances is limited by the number specified in the *nMaxInstances* parameter of the *CreateNamedPipe* API call. To handle more than one pipe instance, a server must consider using multiple threads or asynchronous Windows I/O mechanisms—such as overlapped I/O and completion ports—to service each pipe instance. Asynchronous I/O mechanisms allow a server to service all pipe instances simultaneously from a single application thread. Our discussion demonstrates how to develop advanced servers using threads and overlapped I/O. See Chapter 5 for more information on completion ports as they apply to Windows sockets.

Threads

Developing an advanced server that can support more than one pipe instance using threads is simple. All you need to do is create one thread for each pipe instance and service each instance using the techniques we described earlier for the simple server. The following sample demonstrates a server that is capable of serving five pipe instances. The application is an echo server that reads data from a client and echoes the data back.

```
// Threads.cpp

#include <windows.h>
#include <stdio.h>
#include <conio.h>

#define NUM_PIPES 5

DWORD WINAPI PipeInstanceProc(LPVOID lpParameter);

void main(void)
{
    HANDLE ThreadHandle;
    INT i;
    DWORD ThreadId;

    for(i = 0; i < NUM_PIPES; i++)
    {
        // Create a thread to serve each pipe instance
        if ((ThreadHandle = CreateThread(NULL, 0, PipeInstanceProc,
            NULL, 0, &ThreadId)) == NULL)
        {
            printf("CreateThread failed with error %\n",
                GetLastError());
            return;
        }
        CloseHandle(ThreadHandle);
    }

    printf("Press a key to stop the server\n");
    _getch();
}
```

```
//
// Function: PipeInstanceProc
//
// Description:
//     This function handles the communication details of a single
//     named pipe instance
//
DWORD WINAPI PipeInstanceProc(LPVOID lpParameter)
{
    HANDLE PipeHandle;
    DWORD BytesRead;
    DWORD BytesWritten;
    CHAR Buffer[256];

    if ((PipeHandle = CreateNamedPipe("\\\\.\\PIPE\\jim",
        PIPE_ACCESS_DUPLEX, PIPE_TYPE_BYTE | PIPE_READMODE_BYTE,
        NUM_PIPES, 0, 0, 1000, NULL)) == INVALID_HANDLE_VALUE)
    {
        printf("CreateNamedPipe failed with error %d\n",
            GetLastError());
        return 0;
    }

    // Serve client connections forever
    while(1)
    {
        if (ConnectNamedPipe(PipeHandle, NULL) == 0)
        {
            printf("ConnectNamedPipe failed with error %d\n",
                GetLastError());
            break;
        }

        // Read data from and echo data to the client until
        // the client is ready to stop
        while(ReadFile(PipeHandle, Buffer, sizeof(Buffer),
            &BytesRead,  NULL) > 0)
        {
            printf("Echo %d bytes to client\n", BytesRead);

            if (WriteFile(PipeHandle, Buffer, BytesRead,
                &BytesWritten, NULL) == 0)
            {
                printf("WriteFile failed with error %d\n",
                    GetLastError());
                break;
            }
        }

        if (DisconnectNamedPipe(PipeHandle) == 0)
        {
            printf("DisconnectNamedPipe failed with error %d\n",
                GetLastError());
            break;
        }
    }
    CloseHandle(PipeHandle);
    return 0;
}
```

To develop your server to handle five pipe instances, start by calling the *CreateThread* API function. *CreateThread* starts five execution threads, all of which execute the *PipeInstanceProc* function simultaneously. The *PipeInstanceProc* function operates exactly like the basic server application (the previous sample) except that it reuses a named pipe handle by calling the *DisconnectNamedPipe* API function, which closes a client's session to the server. Once an application calls *DisconnectNamedPipe*, it is free to service another client by calling the *ConnectNamedPipe* function with the same pipe instance handle.

Overlapped I/O

Overlapped I/O is a mechanism that allows Windows API functions such as *ReadFile* and *WriteFile* to operate asynchronously when I/O requests are made. This is accomplished by passing an *OVERLAPPED* structure to these API functions and later retrieving the results of an I/O request through the original *OVERLAPPED* structure using the *GetOverlappedResult* API function. When a Windows API function is invoked with an overlapped structure, the call returns immediately.

To develop an advanced named pipe server that can manage more than one named pipe instance using overlapped I/O, you need to call *CreateNamedPipe* with the *nMaxInstances* parameter set to a value greater than 1. You also must set the *dwOpenMode* flag to *FILE_FLAG_OVERLAPPED*. The next sample demonstrates how to develop this advanced named pipe server. The application is an echo server that reads data from a client and writes the data back.

```
// Overlap.cpp

#include <windows.h>
#include <stdio.h>

#define NUM_PIPES 5
#define BUFFER_SIZE 256

void main(void)
{
    HANDLE PipeHandles[NUM_PIPES];
    DWORD BytesTransferred;
    CHAR Buffer[NUM_PIPES][BUFFER_SIZE];
    INT i;
    OVERLAPPED Ovlap[NUM_PIPES];
    HANDLE Event[NUM_PIPES];

    // For each pipe handle instance, the code must maintain the
    // pipes' current state, which determines if a ReadFile or
    // WriteFile is posted on the named pipe. This is done using
    // the DataRead variable array. By knowing each pipe's
    // current state, the code can determine what the next I/O
    // operation should be.
    BOOL DataRead[NUM_PIPES];
```

```
DWORD Ret;
DWORD Pipe;

for(i = 0; i < NUM_PIPES; i++)
{
    // Create a named pipe instance
    if ((PipeHandles[i] = CreateNamedPipe("\\\\.\\PIPE\\jim",
        PIPE_ACCESS_DUPLEX | FILE_FLAG_OVERLAPPED,
        PIPE_TYPE_BYTE | PIPE_READMODE_BYTE, NUM_PIPES,
        0, 0, 1000, NULL)) == INVALID_HANDLE_VALUE)
    {
        printf("CreateNamedPipe for pipe %d failed "
            "with error %d\n", i, GetLastError());
        return;
    }

    // Create an event handle for each pipe instance. This
    // will be used to monitor overlapped I/O activity on
    // each pipe.
    if ((Event[i] = CreateEvent(NULL, TRUE, FALSE, NULL))
        == NULL)
    {
        printf("CreateEvent for pipe %d failed with error %d\n",
            i, GetLastError());
        continue;
    }

    // Maintain a state flag for each pipe to determine when data
    // is to be read from or written to the pipe
    DataRead[i] = FALSE;

    ZeroMemory(&Ovlap[i], sizeof(OVERLAPPED));
    Ovlap[i].hEvent = Event[i];

    // Listen for client connections using ConnectNamedPipe()
    if (ConnectNamedPipe(PipeHandles[i], &Ovlap[i]) == 0)
    {
        if (GetLastError() != ERROR_IO_PENDING)
        {
            printf("ConnectNamedPipe for pipe %d failed with"
                " error %d\n", i, GetLastError());
            CloseHandle(PipeHandles[i]);
            return;
        }
    }
}

printf("Server is now running\n");


// Read and echo data back to Named Pipe clients forever
while(1)
{
    if ((Ret = WaitForMultipleObjects(NUM_PIPES, Event,
        FALSE, INFINITE)) == WAIT_FAILED)
    {
        printf("WaitForMultipleObjects failed with error %d\n",
            GetLastError());
        return;
    }

    Pipe = Ret - WAIT_OBJECT_0;

    ResetEvent(Event[Pipe]);

    // Check overlapped results, and if they fail, reestablish
    // communication for a new client; otherwise, process read
    // and write operations with the client

    if (GetOverlappedResult(PipeHandles[Pipe], &Ovlap[Pipe],
        &BytesTransferred, TRUE) == 0)
    {
        printf("GetOverlapped result failed %d start over\n",
            GetLastError());

        if (DisconnectNamedPipe(PipeHandles[Pipe]) == 0)
        {
            printf("DisconnectNamedPipe failed with error %d\n",
                GetLastError());
            return;
        }

        if (ConnectNamedPipe(PipeHandles[Pipe],
            &Ovlap[Pipe]) == 0)
        {
            if (GetLastError() != ERROR_IO_PENDING)
            {
                // Severe error on pipe. Close this
                // handle forever.
                printf("ConnectNamedPipe for pipe %d failed with"
                    " error %d\n", i, GetLastError());
                CloseHandle(PipeHandles[Pipe]);
            }
        }

        DataRead[Pipe] = FALSE;
    }
    else
    {
        // Check the state of the pipe. If DataRead equals
        // FALSE, post a read on the pipe for incoming data.
        // If DataRead equals TRUE, then prepare to echo data
        // back to the client.

        if (DataRead[Pipe] == FALSE)
        {
            // Prepare to read data from a client by posting a
            // ReadFile operation

            ZeroMemory(&Ovlap[Pipe], sizeof(OVERLAPPED));
            Ovlap[Pipe].hEvent = Event[Pipe];

            if (ReadFile(PipeHandles[Pipe], Buffer[Pipe],
                BUFFER_SIZE, NULL, &Ovlap[Pipe]) == 0)
            {
                if (GetLastError() != ERROR_IO_PENDING)
```

```
                    {
                        printf("ReadFile failed with error %d\n",
                        GetLastError());
                    }
                }

                DataRead[Pipe] = TRUE;
            }
            else
            {
                // Write received data back to the client by
                // posting a WriteFile operation
                printf("Received %d bytes, echo bytes back\n",
                    BytesTransferred);

                ZeroMemory(&Ovlap[Pipe], sizeof(OVERLAPPED));
                Ovlap[Pipe].hEvent = Event[Pipe];

                if (WriteFile(PipeHandles[Pipe], Buffer[Pipe],
                    BytesTransferred, NULL, &Ovlap[Pipe]) == 0)
                 {
                    if (GetLastError() != ERROR_IO_PENDING)
                    {
                        printf("WriteFile failed with error %d\n",
                        GetLastError());
                    }
                }

                DataRead[Pipe] = FALSE;
            }
        }
    }
}
```

For the server application to service five pipe instances at a time, it must call *CreateNamedPipe* five times to retrieve an instance handle for each pipe. After the server retrieves all the instance handles, it begins to listen for clients by calling *ConnectNamedPipe* asynchronously five times using an overlapped I/O structure for each pipe. As clients form connections to the server, all I/O is processed asynchronously. When clients disconnect, the server reuses each pipe instance handle by calling *DisconnectNamedPipe* and reissuing a *ConnectNamedPipe* call.

Security Impersonation

One of the best reasons for using named pipes as a network programming solution is that they rely on Windows NT platform security features to control access when clients attempt to form communication to a server. Windows NT security offers security impersonation, which allows a named pipe server application to execute in the security context of a client. When a named pipe server executes, it normally operates at the security context permission level of the process that starts the application. For example, if a person with administrator privileges starts up a named pipe server, the server has the ability to access almost every resource on a Windows NT system. Such security access for a named pipe server is bad if the *SECURITY_DESCRIPTOR* structure specified in *CreateNamedPipe* allows all users to access your named pipe.

When a server accepts a client connection using the *ConnectNamedPipe* function, it can make its execution thread operate in the security context of the client by calling the *ImpersonateNamedPipeClient* API function, which is defined as follows:

```
BOOL ImpersonateNamedPipeClient(
    HANDLE hNamedPipe
);
```

The *hNamedPipe* parameter represents the pipe instance handle that is returned from *CreateNamedPipe*. When this function is called, the operating system changes the thread security context of the server to the security context of the client. This is quite handy: If your server is designed to access resources such as files, it will do so using the client's access rights, thereby allowing your server to preserve access control to resources regardless of who started the process.

When a server thread executes in a client's security context, it does so through a security impersonation level. There are four basic impersonation levels: anonymous, identification, impersonation, and delegation. Security impersonation levels govern the degree to which a server can act on behalf of a client. We discuss these impersonation levels in greater detail when we develop a client application later in this chapter. After the server finishes processing a client's session, it should call *RevertToSelf* to return to its original thread execution security context. The *RevertToSelfAPI* function is defined as follows:

```
BOOL RevertToSelf(VOID);
```

This function does not have any parameters.

Client Details

Implementing a named pipe client requires developing an application that forms a connection to a named pipe server. Clients cannot create named pipe instances. However, clients do open handles to preexisting instances from a server. The following steps describe how to write a basic client application:

1. Wait for a named pipe instance to become available using the *WaitNamedPipe* API function.
2. Connect to the named pipe using the *CreateFile* API function.
3. Send data to and receive data from the server using the *WriteFile* and *ReadFile* API functions.
4. Close the named pipe session using the *CloseHandle* API function.

Before forming a connection, clients need to check for the existence of a named pipe instance using the *WaitNamedPipe* function, which is defined as follows:

```
BOOL WaitNamedPipe(
    LPCTSTR lpNamedPipeName,
    DWORD nTimeOut
);
```

The *lpNamedPipeName* parameter represents the named pipe you are trying to connect to. The *nTimeOut* parameter represents how long a client is willing to wait for a pipe's server process to have a pending *ConnectNamedPipe* operation on the pipe.

After *WaitNamedPipe* successfully completes, the client needs to open a handle to the server's named pipe instance using the *CreateFile* API function. *CreateFile* is defined as follows:

```
HANDLE CreateFile(
    LPCTSTR lpFileName,
    DWORD dwDesiredAccess,
    DWORD dwShareMode,
    LPSECURITY_ATTRIBUTES lpSecurityAttributes,
    DWORD dwCreationDisposition,
    DWORD dwFlagsAndAttributes,
    HANDLE hTemplateFile
);
```

The *lpFileName* parameter is the name of the pipe you are trying to open; the name must conform to the named pipe naming conventions mentioned earlier in this chapter.

The *dwDesiredAccess* parameter defines the access mode and should be set to *GENERIC_READ* for reading data off the pipe and *GENERIC_WRITE* for writing data to the pipe. These flags can also be specified together by *OR*ing both flags. The access mode must be compatible with how the pipe was created in the server. Match the mode specified in the *dwOpenMode* parameter of *CreateNamedPipe*, as described earlier. For example, if the server creates a pipe with *PIPE_ACCESS_INBOUND*, the client should specify *GENERIC_WRITE*.

The *dwShareMode* parameter should be set to 0 because only one client is capable of accessing a pipe instance at a time. The *lpSecurityAttributes* parameter should be set to *NULL* unless you need a child process to inherit the client's handle. This parameter is incapable of specifying security controls because *CreateFile* is not capable of creating named pipe instances. The *dwCreationDisposition* parameter should be set to *OPEN_EXISTING*, which means that the *CreateFile* function will fail if the named pipe does not exist.

The *dwFlagsAndAttributes* parameter should always be set to *FILE_ATTRIBUTE_NORMAL*. Optionally, you can specify the *FILE_FLAG_WRITE_THROUGH, FILE_FLAG_OVERLAPPED*, and *SECURITY_SQOS_PRESENT* flags by *OR*ing them with the *FILE_ATTRIBUTE_NORMAL* flag. The *FILE_FLAG_WRITE_THROUGH* and *FILE_FLAG_OVERLAPPED* flags behave like the server's mode flags described earlier in this chapter. The *SECURITY_SQOS_PRESENT* flag controls client impersonation security levels in a named pipe server. Security impersonation levels govern the degree to which a server process can act on behalf of a client process. A client can specify this information when it connects to a server. When the client specifies the *SECURITY_SQOS_PRESENT* flag, it must use one or more of the following security flags:

- *SECURITY_ANONYMOUS.* Specifies to impersonate the client at the anonymous impersonation security level. The server process cannot obtain identification information about the client, and it cannot execute in the security context of the client.

- *SECURITY_IDENTIFICATION*. Specifies to impersonate the client at the identification impersonation security level. The server process can obtain information about the client, such as security identifiers and privileges, but it cannot execute in the security context of the client. This is useful for named pipe clients that want to allow the server to identify the client but not to act as the client.
- *SECURITY_IMPERSONATION*. Specifies to impersonate the client at the impersonation security level. The client wants to allow the server process to obtain information about the client and execute in the client's security context on the local system. Using this flag, the client allows the server to access any local resource on the server as the client. The server, however, cannot impersonate the client on remote systems.
- *SECURITY_DELEGATION*. Specifies to impersonate the client at the delegation impersonation security level. The server process can obtain information about the client and execute in the client's security context on its local system and on remote systems.

> *SECURITY_DELEGATION* works only if the server process is running on Windows 2000 and Windows XP. Windows NT 4.0 does not implement security delegation.

- *SECURITY_CONTEXT_TRACKING*. Specifies that the security-tracking mode is dynamic. If this flag is not specified, security-tracking mode is static.
- *SECURTIY_EFFECTIVE_ONLY*. Specifies that only the enabled aspects of the client's security context are available to the server. If you do not specify this flag, all aspects of the client's security context are available.

Named pipe security impersonation is described earlier in this chapter in the section entitled “Server Details.”

The final parameter of *CreateFile*, *hTemplateFile*, does not apply to named pipes and should be specified as *NULL*. If *CreateFile* completes without an error, the client application can begin to send and receive data on the named pipe using the *ReadFile* and *WriteFile* functions. Once the application is finished processing data, it can close down the connection using the *CloseHandle* function.

The next program listing is a simple named pipe client that demonstrates the API calls needed to successfully develop a basic named pipe client application. When this application successfully connects to a named pipe, it writes the message “This is a test” to the server.

```
// Client.cpp

#include <windows.h>
#include <stdio.h>

#define PIPE_NAME "\\\\.\\Pipe\\jim"

void main(void)
{
    HANDLE PipeHandle;
    DWORD BytesWritten;

    if (WaitNamedPipe(PIPE_NAME, NMPWAIT_WAIT_FOREVER) == 0)
    {
        printf("WaitNamedPipe failed with error %d\n",
            GetLastError());
        return;
    }

    // Create the named pipe file handle
    if ((PipeHandle = CreateFile(PIPE_NAME,
        GENERIC_READ | GENERIC_WRITE, 0,
        (LPSECURITY_ATTRIBUTES) NULL, OPEN_EXISTING,
        FILE_ATTRIBUTE_NORMAL,
        (HANDLE) NULL)) == INVALID_HANDLE_VALUE)
    {
        printf("CreateFile failed with error %d\n", GetLastError());
        return;
    }

    if (WriteFile(PipeHandle, "This is a test", 14, &BytesWritten,
        NULL) == 0)
    {
        printf("WriteFile failed with error %d\n", GetLastError());
        CloseHandle(PipeHandle);
        return;
    }

    printf("Wrote %d bytes", BytesWritten);

    CloseHandle(PipeHandle);
}
```

Other API Calls

There are several additional named pipe functions that we haven't touched on yet. The first set of these API functions—*CallNamedPipe* and *TransactNamedPipe*—is designed to reduce coding complexity in an application. Both functions perform a write and read operation in one call. The *CallNamedPipe* function allows a client application to connect to a message-type pipe (and waits if an instance of the pipe is not available), writes to and reads from the pipe, and then closes the pipe. This is practically an entire client application written in one call. *CallNamedPipe* is defined as follows:

```
BOOL CallNamedPipe(
  LPCTSTR lpNamedPipeName,
  LPVOID lpInBuffer,
  DWORD nInBufferSize,
  LPVOID lpOutBuffer,
  DWORD nOutBufferSize,
  LPDWORD lpBytesRead,
  DWORD nTimeOut
);
```

The *lpNamedPipeName* parameter is a string that represents the named pipe in UNC form. The *lpInBuffer* and *nInBufferSize* parameters represent the address and the size of the buffer that the application uses to write data to the server. The *lpOutBuffer* and *nOutBufferSize* parameters represent the address and the size of the buffer that the application uses to retrieve data from the server. The *lpBytesRead* parameter receives the number of bytes read from the pipe. The *nTimeOut* parameter specifies how many milliseconds to wait for the named pipe to be available.

The *TransactNamedPipe* function can be used in both a client and a server application. It is designed to combine read and write operations in one API call, thus optimizing network I/O by reducing send and receive transactions in the MSNP redirector. *TransactNamedPipe* is defined as follows:

```
BOOL TransactNamedPipe(
  HANDLE hNamedPipe,
  LPVOID lpInBuffer,
  DWORD nInBufferSize,
  LPVOID lpOutBuffer,
  DWORD nOutBufferSize,
  LPDWORD lpBytesRead,
  LPOVERLAPPED lpOverlapped
);
```

The *hNamedPipe* parameter identifies the named pipe returned by the *CreateNamedPipe* or *CreateFile* API functions. The *lpInBuffer* and *nInBufferSize* parameters represent the address and the size of the buffer that the application uses to write data to the pipe. The *lpOutBuffer* and *nOutBufferSize* parameters represent the address and the size of the buffer that the application uses to retrieve data from the pipe. The *lpBytesRead* parameter receives the number of bytes read from the pipe. The *lpOverlapped* parameter allows this *TransactNamedPipe* to operate asynchronously using overlapped I/O.

The next set of functions—*GetNamedPipeHandleState*, *SetNamedPipeHandleState*, and *GetNamedPipeInfo*—are designed to make named pipe client and server communication more flexible at run time. For example, you can use these functions to change the operating mode of a pipe at run time from message mode to byte mode and vice versa. *GetNamedPipeHandleState* retrieves information such as the operating mode (message mode and byte mode), pipe instance count, and buffer caching information about a specified named pipe. The information that *GetNamedPipeHandleState* returns can vary during the lifetime of an instance of the named pipe. *GetNamedPipeHandleState* is defined as follows:

```
BOOL GetNamedPipeHandleState(
    HANDLE hNamedPipe,
    LPDWORD lpState,
    LPDWORD lpCurInstances,
    LPDWORD lpMaxCollectionCount,
    LPDWORD lpCollectDataTimeout,
    LPTSTR lpUserName,
    DWORD nMaxUserNameSize
);
```

The *hNamedPipe* parameter identifies the named pipe returned by the *CreateNamedPipe* or *CreateFile* function. The *lpState* parameter is a pointer to a variable that receives the current operating mode of the pipe handle. The *lpState* parameter can return the value *PIPE_NOWAIT* or the value *PIPE_READMODE_MESSAGE*. The *lpCurInstances* parameter is a pointer to a variable that receives the number of current pipe instances. The *lpMaxCollectionCount* parameter receives the maximum number of bytes to be collected on the client's computer before transmission to the server. The *lpCollectDataTimeout* parameter receives the maximum time in milliseconds that can pass before a remote named pipe transfers information over a network. The *lpUserName* and *nMaxUserNameSize* parameters represent a buffer that receives a null-terminated string containing the user name string of the client application.

The *SetNamedPipeHandleState* function allows you to change the pipe characteristics retrieved with *GetNamedPipeHandleState*. *SetNamedPipeHandleState* is defined as follows:

```
BOOL SetNamedPipeHandleState(
    HANDLE hNamedPipe,
    LPDWORD lpMode,
    LPDWORD lpMaxCollectionCount,
    LPDWORD lpCollectDataTimeout
);
```

The *hNamedPipe* parameter identifies the named pipe returned by *CreateNamedPipe* or *CreateFile*. The *lpMode* parameter sets the operating mode of a pipe. The *lpMaxCollectionCount* parameter specifies the maximum number of bytes collected on the client computer before data is transmitted to the server. The *lpCollectDataTimeout* parameter specifies the maximum time in milliseconds that can pass before a remote named pipe client transfers information over the network.

The *GetNamedPipeInfo* API function is used to retrieve buffer size and maximum pipe instance information. *GetNamedPipeInfo* is defined as follows:

```
BOOL GetNamedPipeInfo(
  HANDLE hNamedPipe,
  LPDWORD lpFlags,
  LPDWORD lpOutBufferSize,
  LPDWORD lpInBufferSize,
  LPDWORD lpMaxInstances
);
```

The *hNamedPipe* parameter identifies the named pipe returned by *Create-NamedPipe* or *CreateFile*. The *lpFlags* parameter retrieves the type of the named pipe and determines whether it is a server or a client and whether the pipe is in byte mode or message mode. The *lpOutBufferSize* parameter determines the size in bytes of the internal buffer for outgoing data. The *lpInBufferSize* parameter receives the size of the internal buffer for incoming data. The *lpMaxInstance* parameter receives the maximum number of pipe instances that can be created.

The final API function, *PeekNamedPipe*, allows an application to look at the data in a named pipe without removing it from the pipe's internal buffer. This function is useful if an application wants to poll for incoming data to avoid blocking on the *ReadFile* API call. The function can also be useful for applications that need to examine data before they actually receive it. For example, an application might want to adjust its application buffers based on the size of incoming messages. *PeekNamedPipe* is defined as follows:

```
BOOL PeekNamedPipe(
  HANDLE hNamedPipe,
  LPVOID lpBuffer,
  DWORD nBufferSize,
  LPDWORD lpBytesRead,
  LPDWORD lpTotalBytesAvail,
  LPDWORD lpBytesLeftThisMessage
);
```

The *hNamedPipe* parameter identifies the named pipe returned by *Create-NamedPipe* or *CreateFile*. The *lpBuffer* and *nBufferSize* parameters represent the receiving buffer along with the receiving buffer size to retrieve data from the pipe. The *lpBytesRead* parameter receives the number of bytes read from the pipe into the *lpBuffer* parameter. The *lpTotalBytesAvail* parameter receives the total number of bytes that are available to be read from the pipe. The *lpBytesLeftThisMessage* parameter receives the number of bytes remaining in a message if a pipe is opened in message mode. If a message cannot fit in the *lpBuffer* parameter, the remaining bytes in a message are returned. This parameter always returns 0 for byte-mode named pipes.

Platform and Performance Considerations

The Microsoft Knowledge Base documents the following problems and limitations. You can access the Knowledge Base at http://support.microsoft.com/support. The following are brief descriptions of each issue.

- Q100291 Restriction on Named-Pipe Names

  If a pipe named \\.\Pipe\Mypipes is created, it is not possible to subsequently create a pipe named \\.\Pipe\Mypipes\Pipe1, because \\.\Pipe\Mypipes is already a pipe name and cannot be used as a subdirectory.

- Q119218 Named Pipe Write Limited to 64K

  The *WriteFile* API function returns *FALSE* and *GetLastError* returns *ERROR_MORE_DATA* when *WriteFile* writes to a message-mode named pipe using a buffer greater than 64 KB.

- Q110148 *ERROR_INVALID_PARAMETER* from *WriteFile* or *ReadFile*

  The *WriteFile* or *ReadFile* function call can fail with the error *ERROR_INVALID_PARAMETER* if you are operating on a named pipe and using overlapped I/O. A possible cause for the failure is that the *Offset* and *OffsetHigh* members of the *OVERLAPPED* structure are not set to 0.

- Q180222 *WaitNamedPipe* and Error 253 in Windows 95

  In Windows 95, when *WaitNamedPipe* fails because of an invalid pipe name passed as the first parameter, *GetLastError* returns Error 253, which is not listed as a possible error code for this function. When you run the same code on Windows NT 4, the error code 161 (*ERROR_BAD_PATHNAME*) appears. To work around the problem, resolve Error 253 the same way as Error 161, *ERROR_BAD_PATHNAME*.

- Q141709 Limit of 49 Named Pipe Connections from a Single Workstation

  If a named pipe server creates more than 49 distinctly named pipes, a single client on a remote computer cannot connect more than 49 pipes on the named pipe server.

- Q126645 Access Denied When Opening a Named Pipe from a Service

  If a service running in the Local System account attempts to open a named pipe on a computer running Windows NT, the operation can fail with an Access Denied error (Error 5).

Conclusion

This chapter introduced the named pipe networking technology, which provides a simple client/server data-communication architecture that reliably transmits data. The interface relies on the Windows redirector to transmit data over a network. A major benefit of named pipes is that it takes advantage of Windows NT platform security features—an advantage offered by no other networking technology described in this book.

Chapter 21

Winsock Error Codes

This chapter lists Winsock error codes by error number. The list does not include the Winsock errors marked BSD-specific or undocumented. In addition, the Winsock errors that map directly to Windows errors appear toward the end of the chapter.

10004—*WSAEINTR*

Interrupted function call. This error indicates that a blocking call was interrupted by a call to *WSACancelBlockingCall*.

10009—*WSAEBADF*

Bad file handle. This error means that the supplied file handle is invalid. Under Microsoft Windows CE, it is possible for the *socket* function to return this error, which indicates that the shared serial port is busy.

10013—*WSAEACCES*

Permission denied. An attempt was made to manipulate the socket, which is forbidden. This error most commonly occurs when attempting to use a broadcast address in *sendto* or *WSASendTo*, in which broadcast permission has not been set with *setsockopt* and the *SO_BROADCAST* option.

10014—*WSAEFAULT*

Invalid address. The pointer address passed into the Winsock function is invalid. This error is also generated when the specified buffer is too small.

10022—*WSAEINVAL*

Invalid argument. An invalid argument was specified. For example, specifying an invalid control code to *WSAIoctl* generates this error. This code can also indicate an error with the current state of a socket—for example, calling *accept* or *WSAAccept* on a socket that is not listening.

10024—*WSAEMFILE*

Too many open files. Too many sockets are open. Typically, Microsoft providers are limited only by the amount of resources available on the system.

10035—*WSAEWOULDBLOCK*

Resource temporarily unavailable. This error is most commonly returned on nonblocking sockets in which the requested operation cannot complete immediately. For example, calling *connect* on a nonblocking socket returns this error because the connection request cannot be completed immediately.

10036—*WSAEINPROGRESS*

Operation now in progress. A blocking operation is currently executing. Typically, you do not see this error unless you are developing 16-bit Winsock applications.

10037—*WSAEALREADY*

Operation already in progress. This error typically occurs when an operation that is already in progress is attempted on a nonblocking socket—for example, calling *connect* or *WSAConnect* a second time on a nonblocking socket already in the process of connecting. This error can also occur when a service provider is in the process of executing a callback function (for those Winsock functions that support callback routines).

10038—*WSAENOTSOCK*

Socket operation on an invalid socket. This error can be returned from any Winsock function that takes a *SOCKET* handle as a parameter. This error indicates that the supplied socket handle is not valid.

10039—*WSAEDESTADDRREQ*

Destination address required. This error indicates that the supplied address was omitted. For instance, calling *sendto* with the destination address *INADDR_ANY* returns this error.

10040—*WSAEMSGSIZE*

Message too long. This error can mean a number of things. If a message is sent on a datagram socket that is too large for the internal buffer, this error occurs. It also occurs if the message is too large because of a network limitation. Finally, if on receiving a datagram the buffer is too small to receive the message, this error is generated.

10041—*WSAEPROTOTYPE*

Wrong protocol type for socket. A protocol was specified in a call to *socket* or *WSASocket* that does not support the semantics of the given socket type. For example, requesting the creation of an IP socket of type *SOCK_STREAM* and protocol *IPPROTO_UDP* generates this error.

10042—*WSAENOPROTOOPT*

Bad protocol option. An unknown, unsupported, or invalid socket option or level was specified in a call to *getsockopt* or *setsockopt*.

10043—*WSAEPROTONOSUPPORT*

Protocol not supported. Either the requested protocol is not installed on the system or no implementation exists for it. For example, if TCP/IP is not installed on the system, attempting to create either a TCP or UDP socket generates this error.

10044—*WSAESOCKTNOSUPPORT*

Socket type not supported. Support for the specified socket type does not exist for the given address family. For example, requesting a socket of type *SOCK_RAW* for a protocol that does not support raw sockets generates this error.

10045—*WSAEOPNOTSUPP*

Operation not supported. The attempted operation is not supported for the referenced object. Typically, this occurs when trying to call a Winsock function on a socket that does not support that operation. For example, calling *accept* or *WSAAccept* on a datagram socket causes this error.

10046—*WSAEPFNOSUPPORT*

Protocol family not supported. The requested protocol family does not exist or is not installed on the system. In most cases, this error is interchangeable with *WSAEAFNOSUPPORT*, which occurs more often.

10047—*WSAEAFNOSUPPORT*

Address family does not support requested operation. This error occurs when attempting to perform an operation that is not supported by the socket type. For example, trying to call *sendto* or *WSASendTo* with a socket of type *SOCK_STREAM* generates this error. This error can also occur when calling *socket* or *WSASocket* and requesting an invalid combination of address family, socket type, and protocol.

10048—*WSAEADDRINUSE*

Address already in use. Under normal circumstances, only one socket is permitted to use each socket address. (For example, an IP socket address consists of the local IP address and port number.) This error is usually associated with the *bind*, *connect*, and *WSAConnect* functions. The socket option *SO_REUSEADDR* can be set with the *setsockopt* function to allow multiple sockets access to the same local IP address and port. (For more information, see Chapter 7.)

10049—*WSAEADDRNOTAVAIL*

Cannot assign requested address. This error occurs when the address specified in an API call is not valid for that function. For example, specifying an IP address in *bind* that does not correspond to a local IP interface generates this error. This error can also occur when specifying port 0 for the remote machine to connect to with *connect*, *WSAConnect*, *sendto*, *WSASendTo*, and *WSAJoinLeaf*.

10050—*WSAENETDOWN*

Network is down. The operation encountered a dead network. This could indicate the failure of the network stack, the network interface, or the local network.

10051—*WSAENETUNREACH*

Network is unreachable. An operation was attempted to an unreachable network. This indicates that the local host does not know how to reach the remote host—in other words, no known route to the destination exists.

10052—*WSAENETRESET*

Network dropped the connection on reset. The connection has been broken because keepalives have detected a failure. This error can also occur when attempting to set the *SO_KEEPALIVE* option with *setsockopt* on a connection that has already failed.

10053—*WSAECONNABORTED*

Software caused the connection to abort. An established connection was aborted due to a software error. Typically, this means the connection was aborted due to a protocol or timeout error.

10054—*WSAECONNRESET*

Connection reset by peer. The remote host forcibly closed an established connection. This error can occur if the remote process is abnormally terminated (as in memory violation or hardware failure) or if a hard close was performed on the socket. A socket can be configured for a hard close using the *SO_LINGER* socket option and *setsockopt*. (For more information, see Chapter 7.)

10055—*WSAENOBUFS*

No buffer space available. The requested operation could not be performed because the system lacked sufficient buffer space.

10056—*WSAEISCONN*

Socket is already connected. A connection is being attempted on a socket that is already connected. This can occur on both datagram and stream sockets. When using datagram sockets, if *connect* or *WSAConnect* has been called to associate an endpoint's address for datagram communication, attempting to call either *sendto* or *WSASendTo* generates this error.

10057—*WSAENOTCONN*

Socket is not connected. This error occurs when a request is made to send or receive data on a connection-oriented socket that is not currently connected.

10058—*WSAESHUTDOWN*

Cannot send after socket shutdown. The socket has already been partially closed by a call to *shutdown*, and either a send or a receive operation is being requested. Note that this occurs only on the data-flow direction that has been shut down. For example, after calling *shutdown* on sends, any call to send data generates this error.

10060—*WSAETIMEDOUT*

Connection timed out. This error occurs when a connection request has been made and the remote computer fails to properly respond (or doesn't respond at all) after a specified length of time. This error is typically seen when the socket options *SO_SNDTIMEO* and *SO_RCVTIMEO* are set on a socket as well as when the *connect* and *WSAConnect* functions are called. For more information on setting *SO_SNDTIMEO* and *SO_RCVTIMEO* on a socket, see Chapter 7.

10061—*WSAECONNREFUSED*

Connection refused. The connection could not be established because the target machine refused it. This error usually occurs because no application on the remote machine is servicing connections on that address.

10064—*WSAEHOSTDOWN*

Host is down. This error indicates that the operation has failed because the destination host is down; however, an application is more likely to receive the error *WSAETIMEDOUT* because it typically occurs when attempting to establish a connection.

10065—*WSAEHOSTUNREACH*

No route to host. An operation was attempted to an unreachable host. This error is similar to *WSAENETUNREACH*.

10067—*WSAEPROCLIM*

Too many processes. Some Winsock service providers set a limit on the number of processes that can simultaneously access them.

10091—*WSASYSNOTREADY*

Network subsystem is unavailable. This error is returned when calling *WSAStartup*, and the provider cannot function because the underlying system that provides services is unavailable.

10092—*WSAVERNOTSUPPORTED*

Winsock.dll version out of range. The requested version of the Winsock provider is not supported.

10093—*WSANOTINITIALISED*

Winsock has not been initialized. A successful call to *WSAStartup* has not yet been performed.

10101—*WSAEDISCON*

Graceful shutdown in progress. *WSARecv* and *WSARecvFrom* return this error to indicate that the remote party has initiated a graceful shutdown. This error occurs on message-oriented protocols such as ATM.

10102—*WSAENOMORE*

No more records found. *WSALookupServiceNext* returns this record to indicate that no additional records are left. This error is interchangeable with *WSA_E_NO_MORE*. Applications should check for both this error and *WSA_E_NO_MORE*.

10103—*WSAECANCELLED*

Operation canceled. This error indicates that a call to *WSALookupServiceEnd* was made while a call to *WSALookupServiceNext* was still processing. *WSALookupServiceNext* returns this error. This code is interchangeable with *WSA_E_CANCELLED*. Applications should check for both this error and *WSA_E_CANCELLED*.

10104—*WSAEINVALIDPROCTABLE*

The procedure call table is invalid. A service provider typically returns this error when the procedure table contains invalid entries. For more information on service providers, see Chapter 12.

10105—*WSAEINVALIDPROVIDER*

Invalid service provider. This error is associated with service providers, and it occurs when the provider cannot establish the correct Winsock version needed to function correctly.

10106—*WSAEPROVIDERFAILEDINIT*

The provider failed to initialize. This error is associated with service providers, and it is typically seen when the provider cannot load the necessary DLLs.

10107—*WSASYSCALLFAILURE*

System call failure. A system call that should never fail has failed.

10108—*WSASERVICE_NOT_FOUND*

No such service found. This error is normally associated with registration and name resolution functions when querying for services. (See Chapter 8 for more information about these functions.) This error indicates that the requested service could not be found in the given namespace.

10109—*WSATYPE_NOT_FOUND*

Class type not found. This error is also associated with the registration and name resolution functions when manipulating service classes. When an instance of a service is registered, it must reference a service class that was previously installed with *WSAInstallServiceClass*.

10110—*WSA_E_NO_MORE*

No more records found. This error is returned from *WSALookupServiceNext* to indicate that no additional records are left. It is interchangeable with *WSAENOMORE*. Applications should check for both this error and *WSAENOMORE*.

10111—*WSA_E_CANCELLED*

Operation canceled. This error indicates that a call to *WSALookupServiceEnd* was made while a call to *WSALookupServiceNext* was still processing. *WSALookupServiceNext* returns this error. This code is interchangeable with *WSAECANCELLED*. Applications should check for both this error and

WSAECANCELLED.

10112—*WSAEREFUSED*

Query refused. A database query failed because it was actively refused.

11001—*WSAHOST_NOT_FOUND*

Host not found. This error occurs with *gethostbyname* and *gethostbyaddr* to indicate that an authoritative answer host was not found.

11002—*WSATRY_AGAIN*

Nonauthoritative host not found. This error is also associated with *gethostbyname* and *gethostbyaddr*, and it indicates that either the nonauthoritative host was not found or a server failure occurred.

11003—*WSANO_RECOVERY*

A nonrecoverable error occurred. This error is also associated with *gethostbyname* and *gethostbyaddr*. It indicates that a nonrecoverable error has occurred and the operation should be tried again.

11004—*WSANO_DATA*

No data record of the requested type found. This error is also associated with *gethostbyname* and *gethostbyaddr*. It indicates that the supplied name was valid but no data record of the requested type was found with it.

11005—*WSA_QOS_RECEIVERS*

At least one reserve message has arrived. This value is associated with IP Quality of Service (QOS) and is not an error per se. (See Chapter 10 for more on QOS.) It indicates that at least one process on the network is interested in receiving QOS traffic.

11006—*WSA_QOS_SENDERS*

At least one path message has arrived. This value is associated with QOS and is more of a status message. This value indicates that at least one process on the network is interested in sending QOS traffic.

11007—*WSA_QOS_NO_SENDERS*

No QOS senders. This value is associated with QOS and indicates that there are no longer any processes interested in sending QOS data. See Chapter 10 for a more complete description of when this error occurs.

11008—*WSA_QOS_NO_RECEIVERS*

No QOS receivers. This value is associated with QOS and indicates that there are no longer any processes interested in receiving QOS data. See Chapter 10 for a more complete description of this error.

11009—*WSA_QOS_REQUEST_CONFIRMED*

Reservation request has been confirmed. QOS applications can request that they be notified when their reservation request for network bandwidth has been approved. When such a request is made, this is the message generated. See Chapter 10 for a more complete description.

11010—*WSA_QOS_ADMISSION_FAILURE*

Error due to lack of resources. Insufficient resources were available to satisfy the QOS bandwidth request.

11011—*WSA_QOS_POLICY_FAILURE*

Invalid credentials. Either the user did not possess the correct privileges or the supplied credentials were invalid when making a QOS reservation request.

11012—*WSA_QOS_BAD_STYLE*

Unknown or conflicting style. QOS applications can establish different filter styles for a given session. This error indicates either unknown or conflicting style types. See Chapter 10 for a description of filter styles.

11013—*WSA_QOS_BAD_OBJECT*

Invalid FILTERSPEC structure or provider-specific object. This error occurs if either the *FLOWSPEC* structures or the provider-specific buffers of a QOS object are invalid. See Chapter 10 for more details.

11014—*WSA_QOS_TRAFFIC_CTRL_ERROR*

Problem with a FLOWSPEC. This error occurs if the traffic control component has a problem with the supplied *FLOWSPEC* parameters that are passed as a member of a QOS object.

11015—*WSA_QOS_GENERIC_ERROR*

General QOS error. This is a catch-all error that is returned when the other QOS errors do not apply.

6—*WSA_INVALID_HANDLE*

Specified event object invalid. This Windows error is seen when using Winsock functions that map to Win32 functions. This particular error occurs when a handle passed to *WSAWaitForMultipleEvents* is invalid.

8—*WSA_NOT_ENOUGH_MEMORY*

Insufficient memory available. This Windows error indicates that insufficient memory is available to complete the operation.

87—*WSA_INVALID_PARAMETER*

One or more parameters are invalid. This Windows error indicates that a parameter passed into the function is invalid. This error also occurs with *WSAWaitForMultipleEvents* when the event count parameter is not valid.

258—*WSA_WAIT_TIMEOUT*

Operation timed out. This Windows error indicates that the overlapped operation did not complete in the specified time.

995—*WSA_OPERATION_ABORTED*

Overlapped operation aborted. This Windows error indicates that an overlapped I/O operation was canceled because of the closure of a socket. In addition, this error can occur when executing the *SIO_FLUSH* ioctl command.

996—*WSA_IO_INCOMPLETE*

Overlapped I/O event object is not in a signaled state. This Windows error is also associated with overlapped I/O. It is seen when calling *WSAGetOverlappedResults* and indicates that the overlapped I/O operation has not yet completed.

997—*WSA_IO_PENDING*

Overlapped operations will complete later. When making an overlapped I/O call with a Winsock function, this Windows error is returned to indicate that the operation is pending and will complete later. See Chapter 5 for a discussion of overlapped I/O.

Chapter 22

NetBIOS Command Reference

This chapter lists and describes the valid commands for the *ncb_command* field of the *NCB* structure that you must pass to the *Netbios* function. Each command description includes a table that indicates which fields of the *NCB* structure you must set for that command and which fields the *Netbios* function sets prior to returning. Each table contains two columns. The first column indicates whether the given field of the *NCB* structure is an input or output parameter. The second column indicates whether the field must be set when making a NetBIOS call. If this column is marked with an X, a value must be provided. Otherwise, if the field is an input parameter and no X is present, providing a value is optional. Please refer to Chapter 17 for an in-depth discussion of the *Netbios* function.

NCBADDGRNAME

This command adds a group name to the local name table. This name cannot collide with a unique name, but anyone else can use it as a group name. Group names are most often used as recipients of datagrams. A name number is returned in the *ncb_num* field that is used in datagram operations. Table 22-1 describes the characteristics of the *NCBADDGRNAME* command.

Table 22-1 *NCBADDGRNAME*

Field	In/Out	Required
ncb_command	In	X
ncb_retcode	Out	
ncb_lsn		
ncb_num	Out	
ncb_buffer		
ncb_length		
ncb_callname		
ncb_name	In	X
ncb_rto		
ncb_sto		
ncb_post	In	
ncb_lana_num	In	X
ncb_cmd_cplt	Out	
ncb_event	In	

NCBADDNAME

This command adds a unique name to the local name table. This name must be unique across the network, or an error is returned. A name number is returned in the *ncb_num* field that is used in datagram operations. Table 22-2 describes the characteristics of the *NCBADDNAME* command.

Table 22-2 *NCBADDNAME*

Field	In/Out	Required
ncb_command	In	X
ncb_retcode	Out	
ncb_lsn		
ncb_num	Out	
ncb_buffer		
ncb_length		
ncb_callname		
ncb_name	In	X
ncb_rto		
ncb_sto		
ncb_post	In	
ncb_lana_num	In	X
ncb_cmd_cplt	Out	
ncb_event	In	

NCBASTAT

This command retrieves the status of a local or remote adapter. When you call this command, set *ncb_buffer* to point to a buffer that has an *ADAPTER_STATUS* structure followed by an array of *NAME_BUFFER* structures. Table 22-3 describes the characteristics of the *NCBASTAT* command.

Table 22-3 *NCBASTAT*

Field	In/Out	Required
ncb_command	In	X
ncb_retcode	Out	
ncb_lsn		
ncb_num		
ncb_buffer	In/Out	X
ncb_length	In/Out	X
ncb_callname	In	X
ncb_name		
ncb_rto		
ncb_sto		
ncb_post	In	
ncb_lana_num	In	X
ncb_cmd_cplt	Out	
ncb_event	In	

NCBCALL

This command connects (opens) a session to another process that you indicate in the *ncb_name* field. Table 22-4 describes the characteristics of the *NCBCALL* command.

Table 22-4 *NCBCALL*

Field	In/Out	Required
ncb_command	In	X
ncb_retcode	Out	
ncb_lsn	Out	
ncb_num		
ncb_buffer		
ncb_length		
ncb_callname	In	X
ncb_name	In	X
ncb_rto	In	
ncb_sto	In	
ncb_post	In	
ncb_lana_num	In	X
ncb_cmd_cplt	Out	
ncb_event	In	

NCBCANCEL

This command cancels a previous outstanding command. The *ncb_buffer* field points to the *NCB* structure with the operation that you want canceled. Canceling an *NCBSEND* or *NCBCHAINSEND* command aborts the session; however, aborting their no-ack variants does not cancel their respective sessions. The following commands cannot be canceled: *NCBADDGRNAME*, *NCBADDNAME*, *NCBCANCEL*, *NCBDELNAME*, *NCBRESET*, *NCBDGSEND*, *NCBDGSENDBC*, and *NCBSSTAT*. Table 22-5 describes the characteristics of the *NCBCANCEL* command.

Table 22-5 *NCBCANCEL*

Field	In/Out	Required
ncb_command	In	X
ncb_retcode	Out	
ncb_lsn		
ncb_num		
ncb_buffer	In	X
ncb_length		
ncb_callname		
ncb_name		
ncb_rto		
ncb_sto		
ncb_post		
ncb_lana_num	In	X
ncb_cmd_cplt	Out	
ncb_event		

NCBCHAINSEND

This command sends the contents of two buffers to the specified receiver. The maximum amount of data that can be sent is 128 KB (a maximum of 64 KB in each buffer). Use *ncb_buffer* and *ncb_length* to point to the first buffer and specify its length. Use bytes 0–1 of *ncb_callname* to specify the length of the second buffer, and use bytes 2–5 to point to it. Table 22-6 describes the characteristics of the *NCBCHAINSEND* command.

Table 22-6 *NCBCHAINSEND*

Field	In/Out	Required
ncb_command	In	X
ncb_retcode	Out	
ncb_lsn	In	X
ncb_num		
ncb_buffer	In	X
ncb_length	In	X
ncb_callname	In	X
ncb_name		
ncb_rto		
ncb_sto		
ncb_post	In	
ncb_lana_num	In	X
ncb_cmd_cplt	Out	
ncb_event	In	

NCBCHAINSENDNA

This command sends the contents of two buffers to the specified receiver and does not wait for any acknowledgment from the receiver. The maximum amount of data that can be sent is 128 KB (a maximum of 64 KB in each buffer). Specify the first buffer and its length in *ncb_buffer* and *ncb_length*, respectively. Use bytes 0–1 of *ncb_callname* to specify the length of the second buffer, and use bytes 2–5 to point to it. Table 22-7 describes the characteristics of the *NCBCHAINSENDNA* command.

Table 22-7 *NCBCHAINSENDNA*

Field	In/Out	Required
ncb_command	In	X
ncb_retcode	Out	
ncb_lsn	In	X
ncb_num		
ncb_buffer	In	X
ncb_length	In	X
ncb_callname	In	X
ncb_name		
ncb_rto		
ncb_sto		
ncb_post	In	
ncb_lana_num	In	X
ncb_cmd_cplt	Out	
ncb_event	In	

NCBDELNAME

This command deletes a name from the local name table. If the name to be deleted is associated with active sessions, the error *NRC_ACTSES* (0x0F) is returned. If any nonactive session commands are outstanding, they receive the error *NRC_NAMERR* (0x17). Table 22-8 describes the characteristics of the *NCBDELNAME* command.

Table 22-8 *NCBDELNAME*

Field	In/Out	Required
ncb_command	In	X
ncb_retcode	Out	
ncb_lsn		
ncb_num		
ncb_buffer		
ncb_length		
ncb_callname		
ncb_name	In	X
ncb_rto		
ncb_sto		
ncb_post	In	
ncb_lana_num	In	X
ncb_cmd_cplt	Out	
ncb_event	In	

NCBDGRECV

This command receives a datagram directed to the local name associated with the *ncb_num* value. If *ncb_num* is 0xFF, this command receives datagrams directed to any local name. The local name can be either a group name or a unique name. If no receive datagram command is pending when a datagram is sent, the data is lost. If the supplied buffer is too small, an "incomplete error" message, *NRC_INCOMP* (0x06), occurs and the data is truncated to fill the buffer. Table 22-9 describes the characteristics of the *NCBDGRECV* command.

Table 22-9 *NCBDGRECV*

Field	In/Out	Required
ncb_command	In	X
ncb_retcode	Out	
ncb_lsn		
ncb_num	In	X
ncb_buffer	In	X
ncb_length	In/Out	X
ncb_callname	Out	
ncb_name		
ncb_rto		
ncb_sto		
ncb_post	In	
ncb_lana_num	In	X
ncb_cmd_cplt	Out	
ncb_event	In	

NCBDGRECVBC

This command receives a broadcast datagram from any name issuing a command to send broadcast datagrams. An "incomplete error" message, *NRC_INCOMP* (0x06), occurs if the supplied buffer is not large enough, and the data is truncated to fill the buffer. Table 22-10 describes the characteristics of the *NCBDGRECVBC* command.

Table 22-10 *NCBDGRECVBC*

Field	In/Out	Required
ncb_command	In	X
ncb_retcode	Out	
ncb_lsn		
ncb_num	In	X
ncb_buffer	In	X
ncb_length	In/Out	X
ncb_callname	Out	
ncb_name		
ncb_rto		
ncb_sto		
ncb_post	In	
ncb_lana_num	In	X
ncb_cmd_cplt	Out	
ncb_event	In	

NCBDGSEND

This command sends a datagram to a specified name. The name can be either a unique name or a group name. If an adapter has a pending receive datagram command for the same name, the adapter receives its own message. The maximum datagram size depends on the underlying protocol. To find the maximum datagram size, you can perform a local *NCBASTAT* command. The *ADAPTER_STATUS* structure that is returned gives the maximum datagram size for the underlying transport protocol. Table 22-11 describes the characteristics of the *NCBDGSEND* command.

Table 22-11 *NCBDGSEND*

Field	In/Out	Required
ncb_command	In	X
ncb_retcode	Out	
ncb_lsn		
ncb_num	In	X
ncb_buffer	In	X
ncb_length	In	X
ncb_callname	In	X
ncb_name		
ncb_rto		
ncb_sto		
ncb_post	In	
ncb_lana_num	In	X
ncb_cmd_cplt	Out	
ncb_event	In	

NCBDGSENDBC

This command sends a broadcast datagram to every host on the LAN. Only those machines with an outstanding receive datagram command get the message. Also, if the local adapter has a pending receive datagram command, it receives its own message. Broadcast datagrams have the same size limitation mentioned in the *NCBDGSEND* entry. Table 22-12 describes the characteristics of the *NCBDGSENDBC* command.

Table 22-12 *NCBDGSENDBC*

Field	In/Out	Required
ncb_command	In	X
ncb_retcode	Out	
ncb_lsn		
ncb_num	In	X
ncb_buffer	In	X
ncb_length	In	X
ncb_callname		
ncb_name		
ncb_rto		
ncb_sto		
ncb_post	In	
ncb_lana_num	In	X
ncb_cmd_cplt	Out	
ncb_event	In	

NCBENUM

This command enumerates LANA numbers. When you issue this command, set *ncb_buffer* to a *LANA_ENUM* structure. On return, the *length* field of *LANA_ENUM* returns the number of LANA numbers on the local machine. The *lana* field of *LANA_ENUM* is filled with the LANA numbers. Table 22-13 describes the characteristics of the *NCBENUM* command.

Table 22-13 *NCBENUM*

Field	In/Out	Required
ncb_command	In	X
ncb_retcode	Out	
ncb_lsn		
ncb_num		
ncb_buffer	In	X
ncb_length	In	X
ncb_callname		
ncb_name		
ncb_rto		
ncb_sto		
ncb_post		
ncb_lana_num	In	X
ncb_cmd_cplt	Out	
ncb_event		

NCBFINDNAME

This command finds the location (machine name) of a name on the network. When this command is issued, *ncb_buffer* is filled with a *FIND_NAME_HEADER* structure, followed by one or more *FIND_NAME_BUFFER* structures. This command is Microsoft Windows NT–specific and is not supported on any other Windows platforms. Table 22-14 describes the characteristics of the *NCBFINDNAME* command.

Table 22-14 *NCBFINDNAME*

Field	In/Out	Required
ncb_command	In	X
ncb_retcode	Out	
ncb_lsn		
ncb_num		
ncb_buffer	In/Out	X
ncb_length	In	X
ncb_callname	In	X
ncb_name		
ncb_rto		
ncb_sto		
ncb_post	In	
ncb_lana_num	In	X
ncb_cmd_cplt	Out	
ncb_event	In	

NCBHANGUP

This command closes a specified connected session. All pending receive commands for the session are terminated and return the "session closed" error message, *NRC_SCLOSED* (0x0A). If either send or chain send commands are outstanding, the hang up command delays until the command completes. This delay occurs whether the commands are transferring data or waiting for the remote side to issue a receive command. Additionally, if multiple outstanding *NCBRECVANY* commands exist, only one of them returns an error code when the session is closed. For any other receive command, each outstanding receive returns an error. Table 22-15 describes the characteristics of the *NCBHANGUP* command.

Table 22-15 *NCBHANGUP*

Field	In/Out	Required
ncb_command	In	X
ncb_retcode	Out	
ncb_lsn	In	X
ncb_num		
ncb_buffer		
ncb_length		
ncb_callname		
ncb_name		
ncb_rto		
ncb_sto		
ncb_post	In	
ncb_lana_num	In	X
ncb_cmd_cplt	Out	
ncb_event	In	

NCBLANSTALERT

This is a Windows NT–only command that notifies the user of LAN failures that last for more than one minute. However, in testing, this command did nothing in response to several common LAN failures, such as disconnected network cables. Table 22-16 describes the characteristics of the *NCBLANSTALERT* command.

Table 22-16 *NCBLANSTALERT*

Field	In/Out	Required
ncb_command	In	X
ncb_retcode	Out	
ncb_lsn		
ncb_num		
ncb_buffer		
ncb_length		
ncb_callname		
ncb_name		
ncb_rto		
ncb_sto		
ncb_post		
ncb_lana_num	In	X
ncb_cmd_cplt	Out	
ncb_event		

NCBLISTEN

This command listens for a connection from another process, local or remote. If the first character of *ncb_callname* is an asterisk (*), a session is established with any network adapter that issues an *NCBCALL* to the local name. The name making the *NCBCALL* is returned in the *ncb_callname* field. If either a send or receive timeout is specified, these timeout values are applied to all send and receive calls made on the new session. Table 22-17 describes the characteristics of the *NCBLISTEN* command.

Table 22-17 *NCBLISTEN*

Field	In/Out	Required
ncb_command	In	X
ncb_retcode	Out	
ncb_lsn	Out	
ncb_num		
ncb_buffer		
ncb_length		
ncb_callname	In/Out	X
ncb_name	In	X
ncb_rto	In	
ncb_sto	In	
ncb_post	In	
ncb_lana_num	In	X
ncb_cmd_cplt	Out	
ncb_event	In	

NCBRECV

This command receives data from the specified session name. If more than one command capable of receiving data is pending, they are processed in the following order:

1. Receive (*NCBRECV*)
2. Receive-any for a specified name (*NCBRECVANY*)
3. Receive-any for any name (*NCBRECVANY*)

All commands with the same precedence are processed in first-in, first-out (FIFO) order. If the buffer passed is not large enough to hold the data, the error *NRC_INCOMP* (0x06) is returned. If this occurs, issue another receive command with a larger buffer unless the send command was issued with either a timeout that expired or a no-ack—in which case the data is lost. The *ncb_length* field is set to the amount of data actually read on return. Table 22-18 describes the characteristics of the *NCBRECV* command.

Table 22-18 *NCBRECV*

Field	In/Out	Required
ncb_command	In	X
ncb_retcode	Out	
ncb_lsn	In	X
ncb_num	In	X
ncb_buffer	In	X
ncb_length	In/Out	X
ncb_callname		
ncb_name		
ncb_rto		
ncb_sto		
ncb_post	In	
ncb_lana_num	In	X
ncb_cmd_cplt	Out	
ncb_event	In	

NCBRECVANY

This command receives data from any session corresponding to the specified name. This command can also be used to receive data destined for any local name by setting the *ncb_num* field to 0xFF. Otherwise, simply set *ncb_num* to the network number returned from adding a name to the local name table. Then any data pending for that particular name will be picked up by this command. Also, a precedence order exists for when multiple receive commands are outstanding. See the entry for *NCBRECV* for more details.

When a session is closed by a local session close command, by the remote side closing the session, or by a session abort command, any outstanding *NRCRECVANY* commands for the specified name complete with the error *NRC_SCLOSED* (0x0A); the *ncb_lsn* field of the *NCB* structure is set to the local session number that was terminated. If no *NCBRECVANY* commands for that closed session are pending for the specified name and an outstanding *NCBRECVANY* command exists for any session (*ncb_num* is 0xFF), that command completes with the error *NRC_SCLOSED* and with the *ncb_lsn* field set to the corresponding session number. Table 22-19 describes the characteristics of the *NCBRECVANY* command.

Table 22-19 *NCBRECVANY*

Field	In/Out	Required
ncb_command	In	X
ncb_retcode	Out	
ncb_lsn	Out	
ncb_num	In/Out	X
ncb_buffer	In	X
ncb_length	In/Out	X
ncb_callname		
ncb_name		
ncb_rto		
ncb_sto		
ncb_post	In	
ncb_lana_num	In	X
ncb_cmd_cplt	Out	
ncb_event	In	

NCBRESET

This command resets the specified LANA number and affects certain environment resources as follows:

- If *ncb_lsn* is not 0, all resources associated with *ncb_lana_num* are freed.
- If *ncb_lsn* is 0, all resources associated with *ncb_lana_num* are freed and new resources are allocated. The *ncb_callname[0]* byte specifies the maximum number of sessions, the *ncb_callname[2]* byte specifies the maximum number of names, and the *ncb_callname[3]* byte requests that the application use the computer's name (which has the name number 1).

Table 22-20 describes the characteristics of the *NCBRESET* command.

Table 22-20 *NCBRESET*

Field	In/Out	Required
ncb_command	In	X
ncb_retcode	Out	
ncb_lsn	In	X
ncb_num	In	X
ncb_buffer		
ncb_length		
ncb_callname		
ncb_name		
ncb_rto		
ncb_sto		
ncb_post		
ncb_lana_num	In	X
ncb_cmd_cplt	Out	
ncb_event		

NCBSEND

This command sends data to the specified session partner. The maximum data size that can be transmitted is 65,536 bytes (64 KB). If the remote side issues a hang up command, all pending sends return the "session closed" error, *NRC_SCLOSED* (0x0A). If more than one send command is pending, they are processed in FIFO order. Table 22-21 describes the characteristics of the *NCBSEND* command.

Table 22-21 *NCBSEND*

Field	In/Out	Required
ncb_command	In	X
ncb_retcode	Out	
ncb_lsn	In	X
ncb_num		
ncb_buffer	In	X
ncb_length	In	X
ncb_callname		
ncb_name		
ncb_rto		
ncb_sto		
ncb_post	In	
ncb_lana_num	In	X
ncb_cmd_cplt	Out	
ncb_event	In	

NCBSENDNA

This command sends data to a specified session and does not wait for acknowledgment from the session partner. Otherwise, the behavior of this command is the same as that of *NCBSEND*. Table 22-22 describes the characteristics of the *NCBSENDNA* command.

Table 22-22 *NCBSENDNA*

Field	In/Out	Required
ncb_command	In	
ncb_retcode	Out	
ncb_lsn	In	X
ncb_num		
ncb_buffer	In	X
ncb_length	In	X
ncb_callname		
ncb_name		
ncb_rto		
ncb_sto		
ncb_post	In	
ncb_lana_num	In	X
ncb_cmd_cplt	Out	
ncb_event	In	

NCBSSTAT

This command retrieves the status of a session. When calling this command, *ncb_buffer* is set to a block of memory that is filled with a *SESSION_HEADER* structure followed by one or more *SESSION_BUFFER* structures. If the first byte of *ncb_name* is an asterisk (*), this command obtains the status for all sessions associated with all names in the local name table. If the supplied buffer is too small, the error *NRC_INCOMP* (0x06) is returned. If the buffer length is less than 4, the error returned is *NRC_BUFLEN* (0x01). Table 22-23 describes the characteristics of the *NCBSSTAT* command.

Table 22-23 *NCBSSTAT*

Field	In/Out	Required
ncb_command	In	X
ncb_retcode	Out	
ncb_lsn		
ncb_num	Out	
ncb_buffer	In	X
ncb_length	In	X
ncb_callname		
ncb_name	In	X
ncb_rto		
ncb_sto		
ncb_post	In	
ncb_lana_num	In	X
ncb_cmd_cplt	Out	
ncb_event	In	

NCBUNLINK

This command unlinks the adapter and is provided for compatibility with earlier versions of NetBIOS. It has no effect on Windows platforms.

About the Authors

Anthony Jones

Anthony Jones was born in San Antonio, Texas, and graduated with honors from the University of Texas at San Antonio in 1996 with a bachelor's degree in computer science. His undergraduate thesis was based upon optimizing the Icon compiler.

After graduation, Anthony worked for Southwest Research Institute, a nonprofit contract research company in San Antonio. There he worked on a variety of projects, including real-time embedded control systems and visualization and simulation software, for customers ranging from the U.S. Air Force to the Weather Channel. In 1997, he moved to Washington State to work for Microsoft Developer Support on the NetAPI team. Anthony recently moved to the Windows 2000 Core Networking department, where he is a tester on the Winsock team.

In his spare time, Anthony enjoys mountain biking, skiing, hiking, photography, and watching *Futurama* and *The X-Files*.

Jim Ohlund

Jim Ohlund works as a software design engineer for Microsoft's Internet Security and Acceleration (ISA) Server test team in Redmond, Washington. He has worked in many areas of the computer industry, from systems programming to developer support to software testing.

In 1990, Jim received a bachelor's degree in computer science from the University of Texas at San Antonio. Jim began his computer career while still in college by developing personnel systems for the United States Department of Defense. He expanded his working knowledge of computer networks and network programming in 1994 by developing terminal emulation software for Windows platforms. In 1996, Jim joined Microsoft's Developer Support Networking API team, helping software developers use many of the networking APIs described in this book.

When Jim is not working with computers, he likes to ski, snowboard, bicycle, and hike in the beautiful Pacific Northwest.