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.

17.1 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.

17.2 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.

17.2.1 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

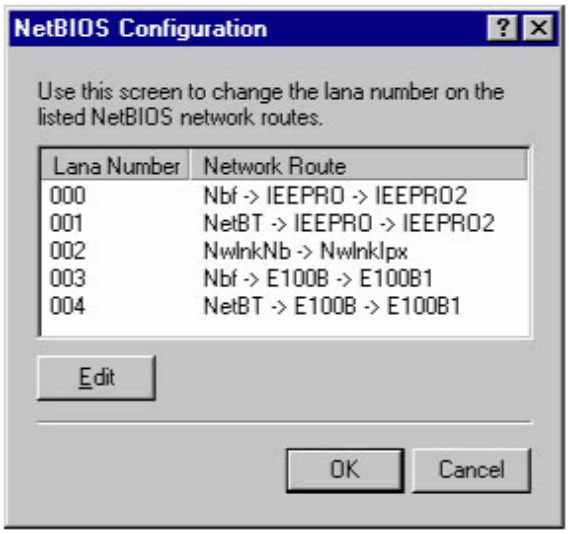

Figure 17-2 *NetBIOS Configuration dialog box. This machine is multihomed with two network cards and three transport protocols: TCP/IP (NetBT), NetBEUI (Nbf), and IPX/SPX (NwlnkNb).*

17.2.2 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.

17.2.3 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.

17.3 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 ORing 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.
ncb_lsn	Identifies the local session number that uniquely identifies a session within the 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.

17.4 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.

17.4.1 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 ORs 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.

17.4.2 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.

17.4.3 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.

17.4.4 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;
}
```

17.5 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.

17.6 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.)

17.6.1 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.

17.6.2 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.

17.7 Platform Considerations

Keep these limitations in mind when implementing NetBIOS with the following platforms.

17.7.1 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.

17.7.2 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.

17.7.3 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.

17.8 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.