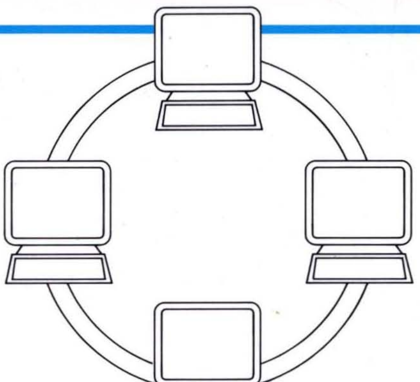
A C Programmer's Guide to the IBM Token Ring



William H. Roetzheim

A C Programmer's Guide to the IBM Token Ring

William H. Roetzheim

Prentice Hall, Englewood Cliffs, New Jersey 07632

Library of Congress Cataloging-in-Publication Data

```
Roetzheim, William H.
   A C programmer's guide to the IBM Token Ring / William H.
  Roetzheim.
      p.   cm.
    Includes bibliographical references (p.    ) and index.
    ISBN 0-13-723768-5
   .1. IBM Token-Ring Network (Local area network system)  2. C
  (Computer program language)
  TK5195.8.I24R64  1991
  005.265--dc20                                          90-14199
                                                              CIP
```

Editorial/production supervision: *Brendan M. Stewart*
Manufacturing buyer: *Kelly Behr* and *Susan Brunke*

Printed in the United States of America
10 9 8 7 6 5 4 3 2 1

ISBN 0-13-723768-5

Prentice-Hall International (UK) Limited, *London*
Prentice-Hall of Australia Pty. Limited, *Sydney*
Prentice-Hall Canada Inc., *Toronto*
Prentice-Hall Hispanoamericana, S.A., *Mexico*
Prentice-Hall of India Private Limited, *New Delhi*
Prentice-Hall of Japan, Inc., *Tokyo*
Simon & Schuster Asia Pte. Ltd., *Singapore*
Editora Prentice-Hall do Brasil, Ltda., *Rio de Janeiro*

To my wife, Marianne, for being so incredibly fun.

Contents

Preface

If you write application programs for the MS-DOS arena, you *must* consider the operation of your software over the IBM Token Ring Network. In the past, this requirement could only be met by wading through dozens of highly technical, difficult to understand reference documents. This book distills the key elements from these documents and presents it in an easy to understand, concise fashion. For most programmers, everything they ever need to know about the IBM Token Ring Network can now be found in one convenient volume. Programming for this environment is covered at the BIOS redirector, NetBIOS, DLC, register direct, and APPC level. The token ring hardware is described, with a particular emphasis on the interaction between the hardware and your application program. Dozens of tables and charts provide a convenient reference to all interrupts, functions, and return codes. Each concept is illustrated with complete C functions which serve both as examples and form the basis of a working library to be used over and over. For advanced users developing network oriented system software, detailed and highly specific references are included to simplify the search for additional details.

Acknowledgements

I would like to thank Dean Ostergaard for his valuable inputs regarding the sample code in this book. Many functions reflect improvements suggested by Dean.

I would also like to thank the Computer Support Corporation for permission to use Arts and Letters^{tm}, which was used to create the figures and illustrations in this book. I would also like to thank IBM for providing technical reference materials used in the book's preparation, and Borland International for providing an advance copy of their latest Turbo C compiler used to write and test all code examples.

Companion Disk Offer

The author has created a set of companion disks to *A C Programmer's Guide to the IBM Token Ring* to save you time typing and proofreading the large number of token ring functions provided in this book. The complete source code for all functions contained in this book is available in 5.25 inch IBM format. If you have questions about the files on these companion disks, you can contact the author through Compuserve at [7653,1365].

The companion disks to *A C Programmer's Guide to the IBM Token Ring* are available only from William H. Roetzheim. To order, send $14 (plus $2.50 per set for domestic postage and handling, $8 per set for foreign orders) to: William H. Roetzheim & Associates, Attn: Companion Disk Offer, 13518 Jamul Drive, Jamul, CA 92035. Payment must be in U.S. funds. You may pay by check or money order (payable to William H. Roetzheim & Associates). Sorry, no COD or purchase orders are accepted.

1. Local Area Network Overview

More and more large companies are installing local area networks, or LANs, in their offices. These networks are normally installed to facilitate the sharing of data and equipment between personal computers within the offices. If you write software which will be used by large or midsized corporations, you *must* address the issue of network compatibility for your program to be successful. For many applications, the end users will not be satisfied if you claim that your software is *usable on* their network; they want software that is designed to take *advantage of* the capabilities offered by the network. This book assumes that you are interested in developing application software that might potentially operate on computers attached to a LAN. This book provides sufficient information for you to develop LAN-based applications successfully without having to wade through volumes of technical details which are primarily of interest to system level programmers. We have chosen to focus our discussion on IBM's *Token Ring Network* because of its popularity and growing market share. By early 1989 the IBM Token Ring had captured 50 percent of the microcomputer local area network market and over 20 percent of all LAN applications worldwide (Glass, 1989). In addition, you will find that much of the discussion will also apply to competing networks. Examples shown were compiled and tested using Borland's Turbo C version 2.0, although every effort was made to ensure compatibility with Microsoft C version 5.1.

This chapter provides an introduction to networking in general and IBM's Token Ring Network in particular. The purpose of this introduction is to familiarize the reader with basic networking concepts which will be used later in the book.

1.1 What Is a Local Area Network?

Stallings defines a local area network as "a communications network that provides interconnection of a variety of data communicating devices within a small area" (Stallings, 1984). We can look at four aspects of this definition to understand LANs better:

1. *Communications network* A LAN consists of a communications network. The communications are provided by various protocols and drivers designed into the network adapters or resident above the adapters in the host device.

2. *Interconnection* The devices we are dealing with must be connected via some form of cable (or other data transmission path).

3. *Data Communicating Devices* LANs typically include both other computers and shared devices, including printers and disk drives.

4. *Small area* A local area network typically covers a small geographic area. For our purposes, we can interpret this to mean that data communications will be at a fairly high rate. You will find performance over the IBM

Token Ring Network to be roughly equivalent to performance using a slow hard disk.

1.2 Network Components

A network consists of user devices (computers and peripherals) connected together by some form of transmission media. The devices are connected to the transmission media using an adapter interface, which can be a card plugged into the computer's backplane or a chip set already built into the motherboard of the device.

Transmission Media

Network transmission media normally consists of physical wires or cables which must be installed throughout the building, although wireless networks using infrared or radio wave transmissions have been attempted with some success. These wires can be:

- Coaxial cables
- Fiber optic cable
- Twisted pair

Coaxial cable is familiar to most people because this is the transmission media used by cable TV companies to broadcast their programming. Coaxial cables can be either 75 ohm or 50 ohm. 50-ohm cable is used for digital networks (on-off signaling), including ethernet. 75-ohm cable is used by the cable TV companies and in networking is typically used for broadband network transmissions using analog waveforms. Broadband networks are normally used to connect different buildings from one company, often combining video, voice, and digital information. Digital information on broadband networks must be encoded for transmission using a modem.

Fiber optic cable is a thin, flexible cable with a center consisting of a thread of glass or plastic, used to guide light. Fiber optic cable is more expensive than coaxial cable, although reduced installation costs (due to its light weight and immunity to nearby electrical energy) may offset much of this difference. Light is generated using either a high-intensity light-emitting diode (LED) or an injection laser diode. Reception is accomplished using a photodiode. Modulation is typically accomplished using the presence or absence of light at a given frequency. Data rates over 3 gigabytes per second have been demonstrated in the laboratory, and ranges of hundreds of megabytes per second are common (Stallings, 1987). Token ring networks operating at approximately 100 Mbps have been demonstrated (e.g., Housley, 1987; Tanimoto, 1987) using fiber optic cable. IBM supports a fiber optic repeater which is normally used to extend the range of ordinary token ring networks (without increasing the data rate).

Twisted pair wiring is used throughout most homes and offices for telephone connections. IBM's Token Ring Network operates over twisted pair wires. When running at 4 Mbps, the token ring adapters can use either IBM type 3 unshielded twisted pair cable (24 gauge wire), or conventional twisted pair phone lines can be used with a media filter. the media filter is a low-pass filter designed to filter out high-frequency harmonics of the signal that might interfere with other equipment near the network lines. When running at 16 Mbps, IBM type 1 or 2 shielded twisted pair cable is required, and conventional phone lines cannot be used. These cables contain 22 gauge twisted pair wire with a metallic shield and plastic cover. Type 1 cable contains two twisted pairs, while type 3 contains four pairs.

For the IBM Token Ring Network, the adapter (computer) connector is a DB-9 connector. The other end of each workstation's wire consists of a special plastic connector, called a *data connector,* that plugs into the multistation access unit (MAU). The multistation access unit is described in the following section.

Connectivity

Fig. 1 illustrates four common network topologies. The *star topology* involves centralized switching between pairs of stations and is often used for digital PBX and digital data switch products. The *bus topology* involves attaching all stations to a single wire. This is the topology of most ethernet networks. The *tree topology* is a generalization of the bus topology. Tree topologies are common for factory automation networks. Finally, the *ring topology* consists of a set of repeaters configured in a ring. The IBM Token Ring Network uses a modified ring topology.

For the IBM Token Ring Network, the basic ring topology was modified to look similar to Fig. 1.2. Although this topology resembles the star topology at first, a closer look will reveal that it is simply a ring in which each segment has been looped to a common point. This modification of the basic ring was made to facilitate network maintenance. It is useful when isolating network problems and simplifies adding and deleting nodes from the ring.

At the center of the IBM Token Ring Network star you will find one or more multistation access units. Each MAU allows up to eight workstations to joining the network and also contains two plugs (called *ring-out* and *ring-in*) to connect multiple MAUs together. The MAU is a box approximately 4 inches high, 6 inches deep, and 18 inches long with all connectors on the front panel. The MAU is completely passive, using the power from the network adapter to open a relay, thus connecting the adapter into the network. When a computer is disconnected from the network (or turned off), the network adapter loses power, the relay opens, and the computer is automatically disconnected from the network. IBM also provides a battery-operated test plug which allows the proper functioning of each MAU port to be

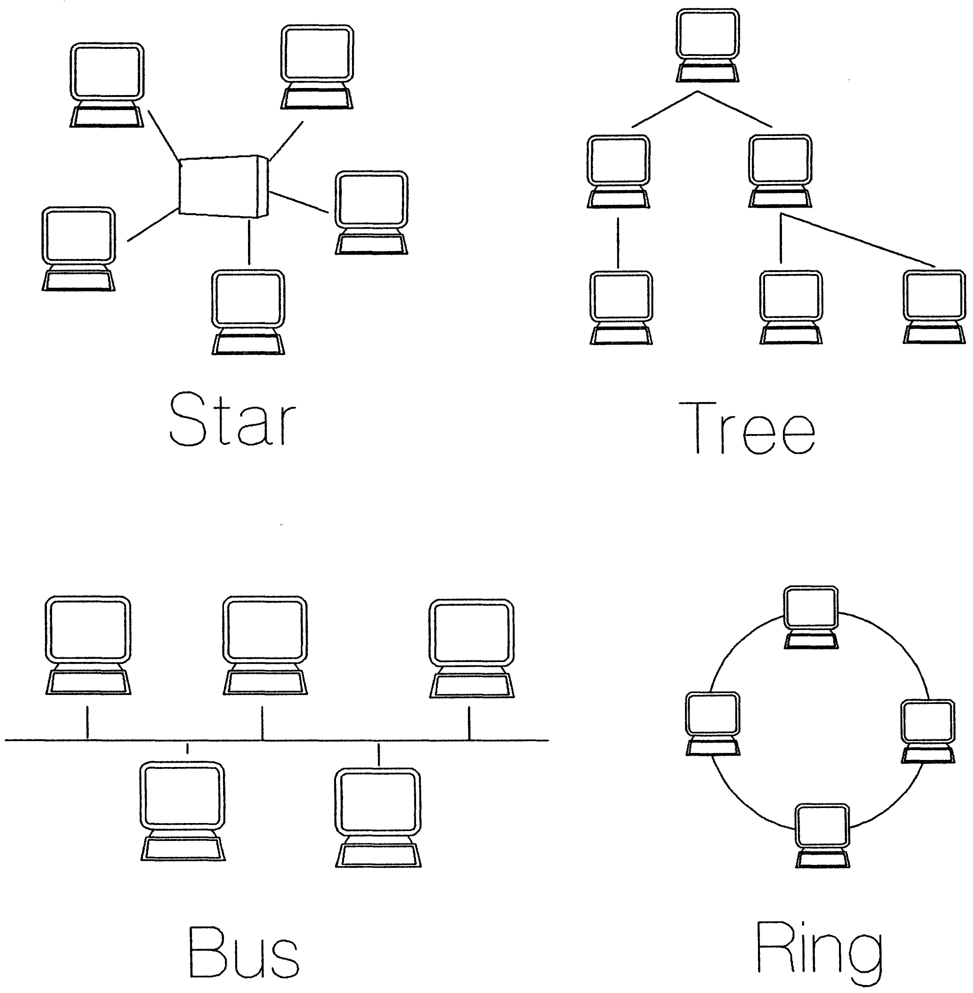


Fig. 1.1 Network topologies.

tested off-line.

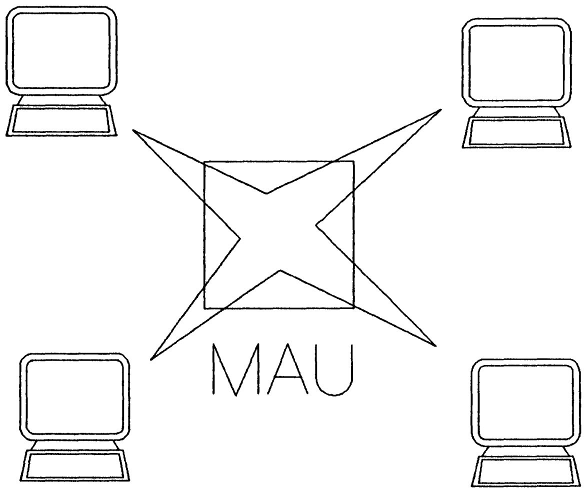


Fig. 1.2 IBM Token Ring Network configuration.

Media Access Method

The most significant difference between IBM's Token Ring Network and its leading competitor, ethernet, is the method used to control media access. Ethernet networks operate with a bus topology using a media access method called carrier sense multiple access/collision detection, or CSMA/CD for short. *Carrier sense multiple access* implies that each network adapter listens to the ethernet bus to determine when the bus is available. Data is then transmitted. Because two adapters may both listen, hear that the bus is available, and begin transmitting at the same time, it is possible for data to collide and be destroyed. This is where the *collision detection* capability comes into play. When an ethernet adapter senses that its data was destroyed, it waits for a period of time, and then retransmits it. The wait time is normally a randomized, exponentially increasing number.

Token ring networks use a different media access method. An electronic token (unique series of bits) is passed around the bus continuously. When an adapter has data to transmit, it waits for the token to arrive, removes the token, transmits its data, then puts the token back on the bus. With this approach, the problem of collisions is avoided.

Ethernet's media access protocol is simpler to implement (in the adapter), is less susceptible to errors due to adapter malfunctions, and performs well as long as the network is lightly loaded. The biggest single advantage of the token ring metwork's media access protocol is that the worst case delay time prior to being able to transmit is deterministic. It is possible to look at the largest allowed network packet size and the number of computers on the network and then determine what the absolute worst case delay is between your desiring to send data and the network becoming available. This information can then be used when sizing your network requirements.

Network Adapters

A network adapter is a hardware interface which allows transmission of data on a local area network. The network adapter often includes firmware to support functions such as media access (who transmits when), flow control, and error detection and correction or retransmission. Some "software-only" networks have been tried using a serial port on the networked computers, but the low cost is offset by the poor performance. Many VME-based single-board computers offer a built-in ethernet interface as part of the motherboard. This is possible because of the low cost and wide availability of VLSI chips supporting the ethernet interface. IBM's Token Ring Network currently requires that a network adapter be purchased as a board which is plugged into the computer's backplane, although the introduction of a Texas Instrument chip set supporting token ring network protocols (Carlo, 1986; Lang, 1989) may point to the future incorporation of the adapter on some vendor's motherboards.

Token ring adapters are available from IBM as well as third-party vendors (e.g., Proteon, 3Com, and Lantana). Up to two adapters can be put in one computer. Two adapters are used when you want the computer to act as a bridge between two token ring networks. Five adapter models are currently available from IBM:

1. The original adapter comes with 8 Kbytes of shared RAM. This adapter operates at 4 Mbps.

2. The Adapter II improves performance by including 16 Kbytes of shared RAM. This adapter operates at 4 Mbps.

3. The Adapter/A is used on IBM PS/2 computers using the MicroChannel architecture. This adapter operates at 4 Mbps.

4. The 16/4 Adapter has 64 Kbytes of shared RAM. This adapter operates at 4 or 16 Mbps. Frame sizes are increased from 2 Kbytes to 18 Kbytes in 16 Mbps mode or 4.5 Kbytes in 4 Mbps mode. In addition, the new adapter implements an early token release capability to decrease token propagation lag on large networks.

5. The 16/4 Adapter/A is identical to the 16/4 Adapter, but is designed for use with the MicroChannel architecture.

Token ring adapters share RAM with the host system. This shared RAM can be configured to start at 0xCC000 or 0xDC000. 0xCC000 is the default. The adapter can be set to operate using hardware interrupt levels 2, 3, 6, or 7. Level 2 is the default, and level 6 should normally not be used (IRQ 6 is used by the disk controller). Finally, the adapter must be told if it is the primary adapter (default) or a secondary adapter in a system unit with two adapter cards installed.

1.3 To What Level Will You Code?

As shown in Figure 3, programmers can work with the IBM Token Ring Network at five different levels.

1. At the highest level, you can require that your users install a local area network program (e.g., PC LAN) and then rely on the BIOS redirector within the LAN program to implement network data transfers. The PC LAN program offered by IBM requires between 50 and 350 Kbytes of memory and uses enough CPU cycles to slow down the routine operation of most applications. Relying on this level of compatibility provides the greatest amount of hardware and network protocol independence. You can be reasonably sure that a properly written application will operate on virtually all LANs available for MS-DOS machines, including both token ring network products and ethernet. This approach is the simplest for the application programmer to implement. For applications which are primarily *not* network oriented, but where network compatibility is important, relying on the BIOS redirector is often best. We discuss this level of network support in Chapter Two.

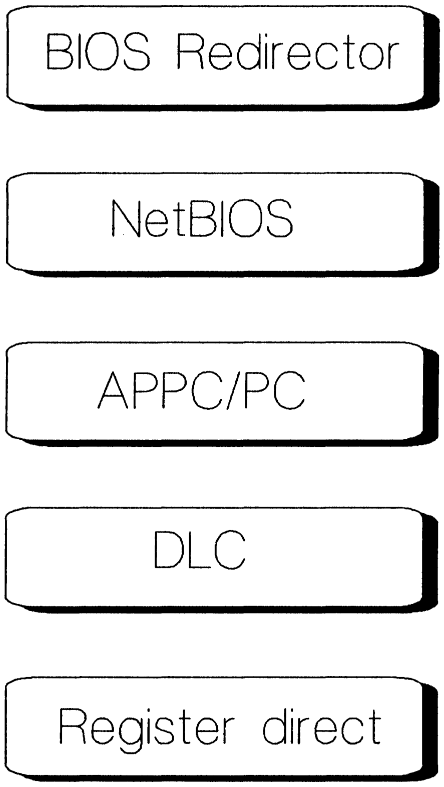


Fig. 1.3 To what level will you code?

2. You can achieve a good degree of network portability by programming using NetBIOS services. NetBIOS support is available from many vendors for a wide range of networks, including several minicomputer networks running UNIX and most PC ethernet networks. NetBIOS support is also included in OS/2. NetBIOS support for the IBM Token Ring Network will require approximately 24 Kbytes of RAM and normally does not have a noticeable affect on CPU performance. NetBIOS programming is

not difficult, although your application must be designed to use NetBIOS services. For general-purpose applications where network support is central to the application's success, NetBIOS programming is often appropriate. We discuss this level of network support in Chapter Three.

3. An alternative to NetBIOS is IBM's Advanced Program-to-Program Communication (APPC) protocol, which is available for all IBM networks. APPC is a remote transaction processing protocol that is common in the IBM mainframe world. Because APPC is much less popular than NetBIOS in the PC area (and APPC is quite complex), the discussion of APPC is delayed until Chapter Seven.

4. It is possible to improve the performance of network data transfers significantly by programming at the *Data Link Control,* or DLC, level. This involves programming using the IEEE 802.2 standards for link level control (LLC). DLC support is available with all IBM Token Ring Network adapters and is built into OS/2. DLC support requires approximately 16 Kbytes of RAM and normally does not have a noticeable effect on CPU performance. DLC programming is normally appropriate for specialized, short network programs (a file transfer utility, for example) or for callable network functions for which performance is critical. If you wish to implement a new network protocol (e.g., TCP/IP), it would probably be appropriate to program at the DLC level. We discuss this level of network support in Chapter Four.

5. At the lowest level, it is possible to program the adapter directly using the registers and shared RAM. The adapter supports the IEEE 802.5 specifications for token ring networks at this level of programming. This method is obviously very hardware dependent. The code is extremely timing sensitive. Shared RAM must be used for data transfers, requiring careful attention to problems involving concurrent updates. Multilevel interrupts must be handled, often with stringent timing constraints. This level of programming is normally only appropriate for diagnostic programs and perhaps for network programmers wishing to implement new low-level protocols that will not operate efficiently over DLC. This level of network support is briefly discussed in Chapter Five.

1.4 Road Map to This Book

As noted, Chapters Two through Five discuss programming for the IBM Token Ring Network at varying levels of support. Chapter Six presents a more technical description of the token ring hardware. Chapter Seven discusses the APPC program interface.

Appendix A is a glossary, Appendix B is a list of acronyms, and Appendix C is a list of references.

This book is designed to provide you with a broad understanding of issues surrounding programming for the IBM Token Ring Network. In addition, the book provides sufficient detail to allow you to exploit 80 percent of the capabilities of the adapter. Many details and exceptions are glossed over to simplify and clarify the key requirements, capabilities, and procedures. For most applications, the level of detail in this book will be completely adequate to do all necessary programming to exploit IBM's Token Ring Network fully. If you find yourself working on one of those rare applications where the information presented here is too broad or general, you can use the Suggested Reading section found at the end of each chapter to delve further into the details.

1.5 Suggested Reading

Glass, B. (1989), "The Token Ring," *Byte*, Vol. 14, No. 1 (January), pp. 363 – 376.

Keller, H, and H.R. Mueller (1985), "Engineering Aspects for Token-ring Design," *Proceedings of the IEEE COMPINT 85 Conference*, September, (Washington, D.C.: IEEE Computer Society Press).

Stallings, William (1987), *Handbook of Computer Communications Standards,* (Volume 2) *Local Network Standards,* New York: Macmillan.

Strole, N.C. (1987), "The IBM Token-ring Network: A Functional Overview," *IEEE Network*, Vol. 1, No. 1, (January), pp. 23 – 30.

Strole, N.C. (1989), "Inside Token Ring Version II, According to Big Blue," *Data Communications*, (January), pp. 117 – 125.

Tanenbaum, Andrew (1988), *Computer Networks*, Englewood Cliffs, NJ: Prentice-Hall.

Townsend, Carl (1987), *Networking with the IBM Token-Ring*, Blue Ridge Summit, PA: Tab Books

2. Relying On The Redirector

The easiest way to use IBM's Token Ring Network is to simply ignore it. When the token ring network is installed and a LAN program is operating on each user's computer, a small routine called the *BIOS Redirector* is initialized. The BIOS Redirector was developed by Microsoft under the name MS-NET and is offered with most PC-based LAN programs. This routine intercepts BIOS and DOS interrupts and redirects them, as necessary, to the appropriate network handling routines. These routines provide an equivalent capability (from your application's perspective), but access data using the network. Your program cannot tell the difference between opening, reading, and closing a file on the local disk and doing the same operations on a remote disk. The entire process is transparent. Although your users must install the LAN program prior to running your application program, this is also true for most other applications and the user's will expect it. The reduced available RAM must be allowed for, and some degradation in application performance anticipated. Because available RAM is so critical to many applications, we will begin by helping you estimate the amount of RAM that your user's LAN software will use. We will then discuss how to design your application so that your users can gain the most utility from their network, including discussions of file locking and record locking.

Finally, we will discuss some specific warnings and hints that apply when writing programs which may run on LAN-based computers.

In this chapter we will define the following BIOS redirector support functions which will simplify the development of application software at this level of network support:

- `net_open()` This function allows you to open files on the network file server.
- `lock_read()` This function allows you to lock and read a record from within a record structured file.
- `lock_write()` This function allows you to write then unlock a record from within a record structured file.
- `lock_open()` This function allows you to lock an entire file when opening it.
- `lock_close()` This function allows you to close and unlock a file.

2.1 PC LAN Memory Requirements

There are four possible configurations that computers running the PC LAN program can select (Fig. 2.1). The *file server* configuration is used for the computer that stores shared files, although the file server can also run application programs. By default, this configuration requires 350 Kbytes of the system memory. This total can be reduced by approximately 112 Kbytes if expanded memory is available or if the IBM disk cache program is used to replace the network cache. The total can be reduced by another 30 Kbytes if the file server will not also act as a printer server. The file server is normally the only computer which can share devices (disks, directories, printers).

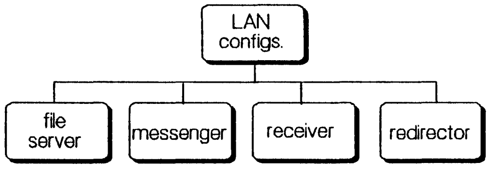


Fig. 2.1 PC LAN configurations.

The *messenger* configuration can use network devices which have been shared by the file server, but can not share its devices. The messenger configuration can also access the PC LAN from within an application through the use of network request keys, and can transfer network messages (notes from one user to another). This configuration requires approximately 160 Kbytes.

The *receiver* configuration requires 68 Kbytes of memory, which allows it to use network devices and send and receive messages. This configuration cannot access the PC LAN services from within an application, resulting in the bulk of the memory saving relative to the messenger configuration. This configuration is likely to be the most popular configuration for your end user workstations on the PC LAN.

For applications requiring that the most possible memory be available, the *redirector* configuration operates using a total of 50 Kbytes. This configuration can use network devices and send messages, but it cannot receive messages.

In general, if your application will function well on a computer with 70 Kbytes of memory allocated to an external application (the PC LAN), it should operate well over a networked computer.

2.2 Supporting Network Paths in Your Application

You will open files on the network file server by using a path which looks identical to paths used for current applications. For example, you might open a file located at "D:\GST\DATA\file01". You must allow your users to configure your application to look at the path (including drive designator) of their choice when opening files. Many users might also wish to store files in a local drive which is at their desk. The best solution is to define two global variables in your program:

- path: This variable points to the local drive and directory, if any.
- net_path: This variable points to the remote (network) drive and directory.

By default, these variables point to the same local drive and directory. If the user is operating over a network, a configuration screen is called up and the net_path variable is modified to point to the file server. In this manner you can create local files (including temporary files) using the path variable, and read and write global data using the net_path variable.

2.3 Using File and Record Locking

When operating over a network, extra precautions must be taken when doing file input-output because more than one person might be simultaneously manipulating the same file. For example, suppose we have a file called DATA01 that stores the account balance for each customer. Let's look at the following scenario:

1. User George opens DATA01, reads in record 1, and begins to record a payment of $150 (i.e., reduces the amount due by $150).

2. User Mary opens the same file, reads in record 1, and begins to record an invoice of $300 (i.e., increases the amount due by $300).

3. User George writes out his modified record 1 to DATA01.

4. User Mary writes out her modified record 1 to DATA01.

At this point, user George thinks that the file has been modified based on his entries, but the record of the customer's $150 payment was lost when user Mary wrote out the record. File and record locking were features added to MS-DOS version 3.0 and higher to prevent this type of problem. In general, software written to operate properly on a LAN requires MS-DOS version 3.0 or higher for this reason.

When implementing file and record locking, the type of protection which is appropriate depends on the nature of the data in the file.

1. *Read only files.* Files which are read only (they are *never* written to) do not create any problems when used over a network. These files can be opened, accessed, and closed exactly as you would for a single user system.

2. *Temporary files.* Temporary files must be created to ensure that the filename does not conflict with another user creating the same temporary file at the same time.

3. *Sequential files.* Sequential files must be accessed using the appropriate *file locking* calls.

4. *Record oriented files* - Record oriented files must be accessed using the appropriate *record locking* calls.

We will discuss the proper approach to handling temporary files, sequential files, and record oriented files in the paragraphs that follow.

Temporary Files

You must be careful when creating temporary working files within your application. For example, suppose you hard coded your application to use a temporary file called *MY-APP.TMP* using a call to **`fopen()`** designed to create the file; that is,

```
fp = fopen("MY-APP.TMP", "w+b");
```

Let's look at what might happen to this code in a network environment:

1. User Susan runs your application and the file "MY-APP.TMP" is created on the file server. She begins to enter data which is stored in the temporary file.

2. User Bob runs your application and the file "MY-APP.TMP" is created on the fileserver. Note that this deletes the existing version of the temporary file. Bob begins to enter data.

3. The next time Susan attempts to access the temporary file, the program will bomb because the file has been deleted (and Susan will lose her work to date).

Luckily, the solution is easy. Whenever you need to open a temporary file, use the code fragment shown in Code Box 2.1. The function **`tmpfile()`** creates a guaranteed unique file in the **`"w+b"`** (binary read/write) mode. The files are created in your working directory and named **`tmpXXXXX`** where **`XXXXX`** is a sequential number.

The file is automatically deleted when closed or when your program terminates normally. If your program terminates abnormally, these temporary files will remain in your working directory. This is especially common when using a debugger to debug the program. In this case, you may need to periodically delete all files in your working directory starting with **tmp**.

Record-Oriented Files

Record oriented files contain multiple blocks of data, with each block the exact same size. For example, a file which stores an array of structures would be a record oriented file. The blocks of data in the file (records) are typically accessed and updated individually. For example, suppose we defined a structure to contain name and address information for our customers:

```
#include        <stdio.h>

main()
{
        FILE    *fp;

        fp = tmpfile();  /* open temporary file with unique name */
        if (fp == NULL) perror("Could not open temporary file");
        ●
        ●
        ●
        Application code here
        ●
        ●
        ●
        fclose(fp);             /* close temporary file */
}
```

Code Box 2.1 Using temporary files.

```
struct  address
{
        char    name[41];
        char    addr1[41];
        char    addr2[41];
        char    city[21];
        char    state[3];
        char    zip[10];
};
```

We can then read, edit, and write an individual record within the file using the approach outlined in Code Box 2.2. Unfortunately, we are still faced with our familiar concurrency problem; i.e., two users simultaneously read the record, then perform some updates, then write the record out resulting in one update not being recorded. We can overcome this problem by locking the record while we are working on it.

Record locking capability is *not* built into the resident BIOS on the computers. These capabilities are built into a **share**, **.exe** or **.com** program distributed with MS-DOS version 3.X and higher. **Share** *must* be executed by the user prior to running your application for record locking to be available. File locking (discussed in the following section) is available without running **share**. Code Box 2.3 and Code Box 2.4 show a routine which will test for the presence of the **share** software. If **share** is *not* installed, **test_share()** attempts to install it. If **share** was found to be installed, or was successfully installed, **test_share()** then disables the control-break interrupt. This must be done to prevent the user from terminating your application while record locks remain on a file (the results of doing this are officially "undefined"). Prior to terminating your program, you should call **test_share()** again with the input flag set to **RESTORE**. This will restore the control-break interrupt vector.

```
int     fh;
struct  address record;
int     record_number;

fh = open("database", O_RDWR | O_CREAT | O_BINARY, S_IREAD | S_IWRITE);

record_number = 5;       /* Update record number 5 */
lseek(fh, record_number * sizeof(struct address), SEEK_SET);
read(fh, record, sizeof( struct address));
●
●
●
Edit record
●
●
●
lseek(fh, record_number * sizeof(struct address), SEEK_SET);
write(fh, record, sizeof(struct address));

close(fh);
```

Code Box 2.2 Record-oriented file access.

Records only need to be locked if the user will (or might) write the record back out to disk. There is no reason you need to prevent multiple users from simultaneously reading the same record so the approach described in Code Box 2.2 is satisfactory. When a write is possible, each record is locked just prior to reading the data and then is unlocked just after the data is written. The command used to lock a specific portion of a file varies from one compiler to another. For portability reasons, it is best localize this compiler dependency in one function. We can conveniently work with record oriented network files by defining and using three new functions. These functions will supplement **open()**, **read()**, and **write()** in the previous sample code fragment.

Code Box 2.5 shows the include file used with all BIOS redirector support functions. This include file contains function prototypes and compiler defines.

The **net_open()** function defined in Code Box 2.6 allows you to open a network file without worrying about the appropriate path each time. The global variable **net_path** should be initialized early

```
#include        <process.h>
#include        <stdio.h>
#include        <dos.h>
#include        "redirect.h"

void    interrupt       do_nothing();

/*****************************************************************
*       test_share() - test to see if share is installed
*
*       Parameters
*               flag (in) - set to INSTALL or RESTORE
*       Global
*               Turbo C global variables for register values
*
*       Returns:
*               0 for success, -1 for failure
*
*       Notes:
*               This function tests for share.exe.  If the share program is
*               not installed, it tries to install it.  If share is successfully
*               installed, it disables interrupt 0x23 to prevent
*               abnormal termination (leaving locks in place).  This interrupt
*               is re-enabled by calling this function with the flag set to
*               RESTORE.
*
*       History:
*               Original code by William H. Roetzheim
*******************************************************************/

int     test_share(int flag)
{
        void    (*interrupt_function);
        void    interrupt do_nothing();
        static  *old_interrupt;

        if (flag == RESTORE)
        {
                if (old_interrupt == NULL) return -1;
                else
                {
                        _AH = 0x25;
                        _AL = 0x23;
                        _DS = FP_SEG(old_interrupt);
                        _DX = FP_OFF(old_interrupt);
                        geninterrupt(0x21);
                        return 0;
                }
        }

        ... continued next code box
```

Code Box 2.3 `test_share()` function definition.

```
        . . . continued from previous code box

        _AH = 0x10;                 /* test for share */
        _AL = 0x00;                 /* get installed state */
        geninterrupt(0x2F);

        if (_AL == 0x00) /* not installed, OK to install */
        {
                spawnlp(P_WAIT, "share", "share", NULL);
                _AH = 0x10; _AL = 0x00; geninterrupt(0x2F);
        }

        if (_AL == 0xFF)            /* successfully installed */
        {
                /* get original value */
                _AH = 0x35;
                _AL = 0x23;
                old_interrupt = MK_FP(_ES, _BX);

                /* set new value */
                interrupt_function = do_nothing;
                _AH = 0x25;
                _AL = 0x23;
                _DS = FP_SEG(interrupt_function);
                _DX = FP_OFF(interrupt_function);
                geninterrupt(0x21);
                return 0;
        }
        else return -1;
}

void interrupt do_nothing()
{
        return;
}
```

Code Box 2.4 `test_share()` function definition (continued).

```
#define MAX_PATH        80      /* maximum path to network files */
#define RETRY           10      /* retrys on failure during read/lock */

#define INSTALL 0
#define RESTORE 1

/* function prototypes */
int     test_share();
int     net_open(char *file_name, int access, unsigned mode);
int     lock_read(int fh, char *buffer, unsigned int length);
int     lock_write(int fh, char *buffer, unsigned int length);
int     lock_open(char *filename, int access, unsigned mode);
int     lock_close(int fh);
```

Code Box 2.5 `redirector.h`.

```
#include        <string.h>
#include        <io.h>
#include        "redirect.h"

extern  char    net_path[];

/*******************************************************************
*       net_open() - open shared file
*
*       Parameters:
*               file_name (in) - file name to be opened
*               access (in) - defined identical to open()
*               mode (in) - defined identical to open()
*
*       Global:
*               net_path - location of all shared files.
*
*       Notes:
*               Assumes that net_path variable already contains
*               trailing backslash (if required).
*
*       Returns:
*               Same as return value from open()
*
*       History:
*               Original code by William H. Roetzheim, 1989
*********************************************************************/

int     net_open(char *file_name, int access, unsigned mode)
{
        char    file[MAX_PATH];

        strcpy(file, net_path);
        strcat(file, file_name);
        return open(file, access, mode);
}
```

Code Box 2.6 `net_open()` function definition.

in your program using some form of configuration file which the user can modify.

The `lock_read()` function shown in Code Box 2.7 works exactly like `read()`, except that `lock_read()` locks the record prior to reading. If the record cannot be locked (possibly because another user has already locked it), the function retries at 1 second intervals until either successful or timed out. `lock_read()` should be used instead of read whenever you expect to write the record out to disk.

The `lock_write()` function shown in Code Box 2.8 works exactly like `write()` except that `lock_write()` unlocks the record just after writing. `lock_write()` should be used anytime you write a record to a network file, and *must* be used whenever you have previously used `lock_read()` to read a record. If you use

```
#include        <io.h>
#include        <errno.h>
#include        <dos.h>
#include        "redirect.h"

/*******************************************************************
*       lock_read() - read shared data from a network file
*
*       Parameters:
*               handle (in) - file handle to read from
*               buffer (in) - buffer to place data in
*               count (in) - number of bytes to read
*
*       Returns:
*               Return code is identical to read()
*
*       Notes:
*               1.      If the desired record is locked, this routine will
*                       retry at 1 second intervals for RETRY attempts.
*
*       History:
*               Original code by William H. Roetzheim, 1989
********************************************************************/

int lock_read(int fh, char *buffer, unsigned int length)
{
        int             timeout = RETRY;
        unsigned        int     count = EACCES;

        while ((lock(fh, lseek(fh, 0, SEEK_CUR), length) != 0) && (timeout > 0))
        {
                timeout--;
                sleep(1);       /* wait one second */
        }

        if (timeout > 0)        /* record is successfully locked */
        {
                count = read(fh, buffer, length);
        }

        return count;
}
```

Code Box 2.7 `lock_read()` function definition.

```
#include        <io.h>
#include        <errno.h>
#include        <dos.h>
#include        "redirect.h"

/*********************************************************************
*       lock_write() - write shared data to a network file
*
*       Parameters:
*               handle (in) - file handle to read from
*               buffer (in) - buffer to place data in
*               count (in) - number of bytes to read
*
*       Returns:
*               Return code is identical to write()
*
*       Notes:
*               1.      This routine assumes that the record to be written
*                       was previously read using net_read().
*
*       History:
*               Original code by William H. Roetzheim, 1989
**********************************************************************/

int lock_write(int fh, char *buffer, unsigned int length)
{
        int     count;

        count = write(fh, buffer, length);       /* write data out */
        unlock(fh, lseek(fh, -length, SEEK_CUR), length);

        return count;
}
```

Code Box 2.8 **`lock_write()`** function definition.

`lock_read()` to lock and read a record, then decide to not write the record back to disk at all, you must call **`unlock()`** directly in your code.

Sequential Files

Sequential files are treated as a single entity which is either locked or unlocked (file locking). The file is locked when it is opened, then unlocked when it is closed. File locking is appropriate for any file which is not record oriented. If you are reading (but not writing) data from a sequential file, you should use the regular **`net_open()`** and **`close()`** functions because there is no need to lock the file. If you

may write new or modified data to the file, you will want to lock the file just prior to opening it, then unlock it
after closing it. The function **lock_open()** as defined in Code Box 2.9 locks the file while opening it. Although Turbo C supports this function with the addition of a flag to the **open()** function, you should use a separate function here to isolate system dependencies.

```
#include        <fcntl.h>
#include        <sys\stat.h>
#include        <share.h>
#include        <io.h>
#include        "redirect.h"

/*****************************************************************
*       lock_open() - lock, then open a file
*
*       Parameters:
*               filename (in) - filename to open
*               access (in) - defined as in open()
*               mode (in) - defined as in open()
*
*       Returns:
*               return values identical to open()
*
*       Copyright:
*               Original code by William H. Roetzheim, 1989
******************************************************************/

int     lock_open(char *filename, int access, int mode)
{
        return net_open(filename, access | SH_DENYRW, mode);
}
```

Code Box 2.9 **lock_open()** function definition.

```
#include        <sys\stat.h>
#include        <stdio.h>
#include        <string.h>
#include        <io.h>
#include        "redirect.h"

extern  char    net_path[];

/*****************************************************************
*       lock_close - unlock and close a file
*
*       Parameters:
*               fh (in) - file handle for open file
*
*       Global:
*               net_path (in) - path to network files
*
*       Notes:
*               Assumes that net_path variable already contains
*               trailing backslash (if required).
*
*       Returns:
*               return value is identical to close()
*
*       History:
*               Original code by William H. Roetzheim, 1989
******************************************************************/

int     lock_close(int fh)
{
        setmode(fh, S_IREAD | S_IWRITE);
        return close(fh);
}
```

Code Box 2.10 `lock_close()` function definition.

Sequential files must be unlocked after they are closed. The function **`lock_close()`** (Code Box 2.10) can be used to perform this operation. If your program fails to unlock a file (e.g. power fails prior to closing the file), you will find that the file is left with the read only bit set. The file can be unlocked using the DOS **`attrib`** command with the following syntax **`attrib -r filename`**. You should include clear instructions in your User's Manual describing how to unlock files which are accidentally left locked. You might also include a built in capability within your application to unlock a locked file (using a call to **`chmod()`**).

2.4 Hints and Warnings

The following miscellaneous hints and warnings will help smooth your programming when relying on the BIOS Redirector:

- Use the highest level of interrupt available for each function. For example, print data by opening a printing device and outputing to that device rather than directly calling the routines to print a single character.

- Do not directly modify the display mode, as this will confuse the LAN software if the user attempts to pop up a LAN
control screen from within your application. Use the BIOS service routine for this instead, or simply use the built in Turbo-C functions for modifying the display mode.

- It is possible for a non-file server configuration to allow another PC to temporarily access its local devices (disk, printer, etc.) Most application software does not use this feature, but if your application would benefit from this capability, look up the **`NET USE`** and **`NET SHARE`** commands in your LAN manual and the **`permit`** command in your DOS manual.

- Test your application with an appropriate amount of RAM reserved for network software use. If your computer does not have the network software installed, set aside the appropriate amount of RAM by setting up a RAM drive sized to the amount of memory (from Section 2.1) that a LAN program will require to function.

2.5 Suggested Readings

Borland International (1988), *Turbo C Reference Guide*, Scotts Valley, CA: Borland International.

IBM (1987), *IBM PC Local Area Network Program User's Guide*, Austin, TX: International Business Machines Corporation.

Microsoft (1987), *Microsoft C Run-Time Library Reference*, Redmond, WA: Microsoft.

Svobodova, Liba (1984), "File Servers for Network Based Distributed Systems," *ACM Computing Surveys*, Vol. 16, no. 4, (December).

Van Name, M.L. (1989), "Anatomy of a LAN Operating System," *Byte*, Vol. 14., no. 6, (June), pp. 157-160.

Wiederhold, Gio (1983), *Database Design*, New York: McGraw-Hill.

3. Portability Using NetBIOS

Programming to the NetBIOS interface provides the programmer with independence from the LAN hardware, the LAN protocol, and the underlying operating system. In addition, NetBIOS provides a high level of protocol support which handles many of the network communication issues (e.g., error detection) common to network communications. NetBIOS support is available on vitually all MS-DOS and OS/2 machines which are attached to a LAN and requires very little system memory to operate.

NetBIOS programming assumes that two processes (programs) are cooperating to exchange data. For example, you cannot use NetBIOS alone to open and read a file on a remote network node. Rather, you must write and run a program on the remote system that will listen for file input-output commands (over the network), read the data locally, and then send it back to you.

NetBIOS works by first establishing a unique name (unique within the network) for each participant. It is then possible for users to exchange data using either a connection-oriented protocol or a datagram protocol. A *connection-oriented protocol* (also called a *virtual circuit*) offers guaranteed delivery of the data, but with a performance sacrifice. Guaranteed delivery in this context means that the data was either safely and accurately delivered or else your application was notified of the problem. *Datagram* protocols make a best effort to deliver the data, but there is no guarantee that it was safely received, nor are you notified in the event of a failure. A connection-oriented protocol is normally appropriate for functions such as file transfers. A datagram protocol is appropriate for functions such as periodic status updates, where timeliness is more important than guaranteed delivery. Datagram protocols are also used when your application is performing its own flow control and error correction/detection processing.

In this chapter we will learn how to use NetBIOS to send and receive data over the network. We begin by discussing the NetBIOS Network Control Block and interrupt procedures. We then discuss name conventions over NetBIOS and show how to register a name. The next two sections are devoted to session-oriented data transmission and datagram oriented data transmission. We then wrap up our discussion by describing some miscellaneous NetBIOS commands. Finally, the end of the chapter contains a reference table of NetBIOS commands and NetBIOS return codes.

The following functions are defined in this chapter:

`init_ncb()` This function initializes a network control block.

`int_netbios()` This function issues an interrupt to NetBIOS, requesting the processing of a Network Control Block.

`init_netbios()` This function initializes NetBIOS and registers an application name.

`shutdown_netbios()` This function terminates NetBIOS processing and deletes an application name.

`dg_write()` Write a datagram using NetBIOS.

`dg_read()` Receive a datagram using NetBIOS.

`max_dg()` Return the largest possible datagram size.

`sn_open()` Open a session over NetBIOS.

`sn_read()` Read data from a session.

`sn_write()` Write data over a session.

`sn_close()` Terminate a session.

`sn_receive()` Initiate a session-oriented receive operation in background.

`sn_send()` Initiate a session-oriented send operation in background.

`get_session_status()` Get current session status.

3.1 NetBIOS Network Control Blocks

Communication between your application and the NetBIOS is accomplished using a structure called the Network Control Block, or NCB. The format for the NCB is shown in Code Box 3.1. The structure description assumes that you are using unsigned characters (a Turbo C compiler option) and that pointers are 32 bits long (large or huge memory model). If you have your compiler defaulting to signed characters, you should explicitely declare the structure variables as **`unsigned char`**. If you are using a memory model with 16 bit pointers for data, cast the pointer to **`buffer`** to be a far pointer. If you are using a memory model with 16 bit pointers for code, cast the pointer to `post` to be a far pointer.

The variables in the structure have the following meanings:

- **`command`** This is the command number to be executed.
- **`ret_code`** This is the return code after completion of the command.
- **`lsn`** This is the logical session number assigned by NetBIOS. This field is only used for connection oriented communication as discussed in Section 3.3.
- **`number`** This field contains the number assigned by NetBIOS to your application program. This field is discussed in Section 3.2.
- **`buffer`** This field points to your local buffer from which data will be sent or into which data will be received.
- **`length`** For transmitted data, this field contains the length of the data to send. For received data, this field contains the number of characters received.
- **`r_name`** Remote system name. This field is discussed further in Section 3.2.
- **`l_name`** Local system name. This field is discussed further in Section 3.2.
- **`rto`** Receive time out in .5 second increments.

```
struct  net_control_block
{
        char    command;
        char    ret_code;
        char    lsn;     /* logical session number */
        char    number;  /* application name table entry */

        char    *buffer;
        unsigned         int     length;  /* buffer length */

        char    r_name[16];
        char    l_name[16];
        char    rto;     /* receive time out */
        char    sto;     /* send time out */

        void    *post;   /* post routine location */
        char    adapter;         /* adapter number */
        char    complete;

        char    reserved[14];
};
```

Code Box 3.1 **net_control_block** structure definition.

- **sto** Send time out in .5 second increments.
- **post** Address of post routine. This field is discussed later in this section.
- **adapter** Adapter number (in this computer): 0 for the primary adapter, 1 for the alternate adapter (normally 0 except when performing gateway functions).
- **complete** This field is set to 0xFF during adapter processing, then set to the same value as **ret_code** upon completion. This field is discussed later in this section.
- **reserved** Used as a work area by NetBIOS during processing.

When you call NetBIOS to process the Network Control Block, it is possible to instruct the adapter to perform its processing (e.g., data transfer) independently (in background) while control is returned to your application immediately. For example, you might use this capability to prepare the next data packet while the current data packet is being transmited. One way to tell when the adapter has completed the current command is to use the **`post`** field of the Network Control Block. The **`post`** field is a pointer to a function that should get control after completion of the command. The function pointed to by **`post`** should be a short interrupt handling function. When called, the AX register will contain the completion code for the command while the ES and BX registers will point to the Network Control Block. The post routine should be declared to be of type **`void interrupt`**. When the post handling function is done, it must use an interrupt return instruction,[1] which is handled automatically by Turbo C when you declare the routine to be of type interrupt.

The more common method of determining when the adapter has finished processing the Network Control Block command is to monitor the **`complete`** field in the structure. This field is set (by the adapter) to 0xFF during processing, so your application can simply test this field until the value is something other than 0xFF. Until the **`complete`** field indicates that the adapter is done processing this Network Control Block, you *must not modify either the Network Control Block structure contents nor the buffer contents pointed to by the Network Control Block.* When using this approach, the value in the **`post`** field should be set to NULL, which tells the adapter that no post routine is installed. This is the normal way the NetBIOS commands are handled. Of course, if you do not use the **`NO_WAIT`** option to the commands, your application program will be suspended until completion of the adapter processing and you will not need to concern yourself with

[1] The post routine is slightly different if you are using OS/2. Refer to the OS/2 specific chapter of this book for details.

either the **post** routine or monitoring the **complete** field in the Network Control Block.

When the adapter is done processing the Network Control Block, it sets both the **complete** field and the **ret_code** field to the same return value. The adapter always uses 0x00 to indicate successful completion of the command. The meaning of other possible return codes (error conditions) are listed in Section 3.7.

The following listing shows the **netbios.h** header file used for all NetBIOS support functions described in this chapter. This file includes a number of defines to clarify NetBIOS calls, the **net_control_block** structure definition, and our NetBIOS function prototypes.

```
#define RECEIVE_TIMEOUT 0       /* no timeout, wait forever */
#define SEND_TIMEOUT    0

/* flags included for clarity */
#define CLIENT  0
#define SERVER  1
#define FIRST   0
#define NEXT    1

/* NetBIOS commands */
#define NCB_ADD_GROUP_NAME              0x36
#define NCB_ADD_NAME                    0x30
#define NCB_CALL                                0x10
#define NCB_CANCEL                      0x35
#define NCB_CHAIN_SEND                  0x17
#define NCB_CHAIN_SEND_NO_ACK           0x72
#define NCB_DELETE_NAME                 0x31
#define NCB_FIND_NAME                   0x78
#define NCB_HANG_UP                     0x12
#define NCB_LAN_STATUS_ALERT            0xF3
#define NCB_LISTEN                      0x11
#define NCB_RECEIVE                     0x15
#define NCB_RECEIVE_ANY                 0x16
#define NCB_RECEIVE_BROADCAST_DATAGRAM  0x23
#define NCB_RECEIVE_DATAGRAM            0x21
#define NCB_RESET                       0x32
#define NCB_SEND                                0x14
#define NCB_SEND_BROADCAST_DATAGRAM     0x22
#define NCB_SEND_DATAGRAM               0x20
#define NCB_SEND_NO_ACK                 0x71
#define NCB_SESSION_STATUS              0x34
#define NCB_STATUS                      0x33
#define NCB_TRACE                       0x79
#define NCB_UNLINK                      0x70

/* NetBIOS command flags */
#define WAIT            0x00
#define NO_WAIT         0x80

/* NetBIOS return values */
#define NO_NETBIOS      0x00
```

```
#define INVALID_NAME    0xFF

struct  net_control_block
{
        char    command;
        char    retcode;
        char    lsn;            /* logical session number */
        char    number;         /* application name table entry */

        char    *buffer;
        unsigned        int     length;  /* buffer length */

        char    r_name[16];
        char    l_name[16];
        char    rto;            /* receive time out */
        char    sto;            /* send time out */

        void    *post;          /* post routine location */
        char    adapter;        /* adapter number */
        char    complete;

        char    reserved[14];
};

struct  session_status
{
        char    number;         /* name table entry */
        char    number_of_sessions;
        char    outstanding_receive_datagram;
        struct
        {
                char    session_number;
                char    state;
                char    l_name[16];
                char    r_name[16];
                char    outstanding_receive;
                char    outstanding_send;
        } session;
};


/* function prototypes */
void            init_ncb(struct net_control_block *ncb);
void            int_netbios(struct net_control_block *ncb);
unsigned int    init_netbios(char *name);
unsigned int    shutdown_netbios(char *name);
int             dg_write(unsigned int number, char *destination, char *buffer, int
length);
int             dg_read(unsigned int number, char *from, char *buffer, int length);
int             max_dg();
int             sn_open(char *from, char *to, int flag);
int             sn_read(char lsn, void *buffer, unsigned int nbytes);
int             sn_write(char lsn, void *buffer, unsigned int nbytes);
int             sn_close(char lsn);
struct  net_control_block *sn_receive(char lsn, void *buffer, unsigned int nbytes);
struct  net_control_block *sn_send(char lsn, void *buffer, unsigned int nbytes);
struct          session_status *get_session_status(char *name, int flag);
```

Most NetBIOS commands can be executed with or without blocking.[2] When executed with blocking, your application program blocks (waits) until the adapter has finished executing the command. This is the default for all commands. When executed without blocking, your application program continues to execute while the adapter processes the command in background. To execute a command with blocking, you would use the following syntax:

```
ncb.command = NCB_CALL | WAIT;
```

The **WAIT** flag is optional because this is the default for all commands. **NCB_CALL** is one sample NetBIOS command defined in **netbios.h**. To execute the same command without blocking, you would use

```
ncb.command = NCB_CALL | NO_WAIT;
```

Our programming will be simplified with two support NetBIOS functions, **init_ncb()** and **int_netbios()**. **init_ncb()** (Code Box 3.2) clears a Network Control Block and sets the defaults for sent timeout and receive timeout. **int_netbios()** (Code Box 3.3) clears a Net Control Block and sets the defaults for sent timeout and receive timeout. **int_netbios()** (Code Box 3.4) executes a NetBIOS NCB.

3.2 Naming Conventions and Procedures

Each IBM Token Ring Network adapter can store up to 255 network user names. 0x00 and 0xFF are not used, and 0x01 is permanently assigned based on a unique number burned into each adapter, leaving 252 available name slots. Name 0x01 is used as a guaranteed unique name which is assigned to each adapter. Names can be up to 16

[2] The exceptions are **NCB_RESET**, **NCB_CANCEL**, and **NCB_UNLINK**.

```
#include        <string.h>
#include        "netbios.h"

/*******************************************************************
*       init_ncb - clear and initialize net control block
*
*       Parameters:
*               ncb (in/out) - net control block to be cleared
*
*       Notes:
*               This code sets the network adapter number to 0 (primary)
*
*       History:
*               Original code by William H. Roetzheim, 1990
*********************************************************************/

void    init_ncb(struct net_control_block *ncb)
{
        memset(ncb, 0, sizeof(struct net_control_block));
        ncb->rto = RECEIVE_TIMEOUT;
        ncb->sto = SEND_TIMEOUT;
}
```

Code Box 3.2 `init_ncb()` function definition.

characters long, although restrictions on the range of the last character make it simpler to restrict names to 15 characters. Names are normally assigned so that they are unique on any given network. For example, if your application registered a unique name of **`fileserver`**, no other adapter could use this name until you released it. The only exception is *group names*, which may be shared among adapters.

For two cooperating processes, you will normally know both the registered network name of your application and the network name of the other application. These two names can then be used to communicate. This approach would not work very well for a file server, however, because the file server has no way of knowing in advance who will call on it for assistance. In this case, the file server name is made available to other applications and the file server issues a *receive any* network request. This will allow it to receive any messages directed to it without requiring that it know the name of the sender a priori. Client applications would then address messages to the server by its previously known name.

```
#include        <dos.h>
#include        "netbios.h"

extern  int     net_error;

/*****************************************************************
*       int_netbios - interrupt NetBIOS with net control block
*
*       Parameters:
*               ncb (in/out) - initialized net control block
*
*       Global:
*               _ES - ES register
*               _BX - BX register
*               net_error - set to command return code
*
*       History:
*               Original code by William H. Roetzheim
******************************************************************/

void    int_netbios(struct net_control_block *ncb)
{
        _ES = FP_SEG(ncb);
        _BX = FP_OFF(ncb);
        geninterrupt(0x5C);
        net_error = ncb->retcode;

}
```

Code Box 3.3 `int_netbios()` `function definition.`

The NetBIOS commands which are related to naming are

- **NCB_ADD_GROUP_NAME** This command allows you to add a group name to your adapter's network name table. Group names are not necessarily unique across the network. The command will fail if another adapter has previously registered the same name as a unique name.

- **NCB_ADD_NAME** This command allows you to add a unique name to your adapter's network name table. This command will fail if another adapter has previously registered the same name as either a unique or group name.

- **NCB_DELETE_NAME** This command deletes a name from your adapter's network name table.

- **NCB_FIND_NAME** This command uses a broadcast message to find every adapter on the network with a specified name registered. It uses the adapter unique name (stored in slot 0x01 of the name table) to tell you specifically which adapters are using the name.

Code Box 3.5 shows a function which should be used at the beginning of your NetBIOS program. **init_netbios()** begins by testing to ensure that NetBIOS is installed and appears to be functioning properly. If this test fails, the function returns **NO_NETBIOS** (defined in **netbios.h**). It then attempts to registered your application name as a unique name. If this fails, it returns **INVALID_NAME**. You can then either display an error message or try again with a different name. If NetBIOS is installed and the name is successfully registered, **init_netbios()** returns the name table number assigned to your unique name. This number will be needed later if your are using NetBIOS datagram services.

When you are done using NetBIOS, you can use the **shutdown_netbios()** function defined in Code Box 3.6 to delete your application name from the adapter name table. It is important that you delete unused names to avoid filling the adapter name table with unused names as well as preventing other users on the network from using those names. The **NCB_RESET** command can also be used to delete your application name from the adapter name table because this command deletes *all* names from the name table. Many books on NetBIOS routinely use **NCB_RESET** to "clean up" when their application is done. You must be careful with this approach because under DOS **NCB_RESET** clears *all* names in the table, *not* just those that your application registered (**NCB_RESET** works differently under OS/2). This could cause obvious problems if other applications were using NetBIOS (and the adapter name table) along with your program.

```
#include        <dos.h>
#include        "netbios.h"

/********************************************************************
*       init_netbios - test for NetBIOS presence and register application
*
*       Parameters:
*               name (in) - application name for network use
*
*       Returns:
*               Name table number if successful, else:
*                       NO_NETBIOS      if NetBIOS not installed or adapter
*                                       failure
*                       INVALID_NAME    if name is already in use or invalid
*
*       Notes:
*               The name table number is required for datagram support but not
*               for connection oriented support.
*
*               Application names longer than 15 characters are truncated.
*
*               The first three characters of the name should not be "IBM".
*
*       History:
*               Original code by William H. Roetzheim
*********************************************************************/

unsigned int init_netbios(char *name)
{
        int     i;
        unsigned        long    int_vector;
        struct          net_control_block       ncb;

        /***** start by testing for NetBIOS installation *****/
        /* is interrupt vector initialized? */
        int_vector = (unsigned long) getvect(0x5C);
        if ((int_vector == 0x0000) || (int_vector == 0xF000))
        {
                /* no interrupt handler installed */
                return NO_NETBIOS;
        }

        /* is NetBIOS responding? */
        init_ncb(&ncb);
        ncb.command = 0xFF;     /* an invalid command */
        int_netbios(&ncb);
        if (ncb.retcode != 0x03) /* error, invalid command */
        {
                return NO_NETBIOS;
        }

        /* now attempt to register name on network */
        init_ncb(&ncb);
        for (i = 0; i < 15; i++)
        {
                if (name[i] == 0) break;
                ncb.l_name[i] = name[i++];
        }
        ncb.command = NCB_ADD_NAME;
        if (ncb.retcode != 00) return INVALID_NAME;
        else return ncb.number;
}
```

Code Box 3.4 `init_netbios()` function definition.

```
#include        <dos.h>
#include        "netbios.h"

/*****************************************************************
*       shutdown_netbios - Clear name table entry
*
*       Parameters:
*               name (in) - application name used during init_netbios()
*
*       Returns:
*               0 for success, else
*               return codes defined for NCB_DELETE_NAME command
*
*       History:
*               Original code by William H. Roetzheim
******************************************************************/

unsigned int    shutdown_netbios(char *name)
{
        int             i;
        struct          net_control_block       ncb;

        init_ncb(&ncb);
        for (i = 0; i < 15; i++)
        {
                if (name[i] == 0) break;
                ncb.l_name[i] = name[i++];
        }
        ncb.command = NCB_DELETE_NAME;
        return ncb.retcode;
}
```

Code Box 3.5 `shutdown_netbios()` function definition.

3.3 Datagram-Oriented Communication

Datagrams are messages which are transmitted over the network without any attempt to verify error free reception. They are fast, easy on network resources, simple to use, and require a minimum of coordination between the communicating programs. For these reasons, datagrams are often used for functions such as status updates, initial coordination to establish session-oriented communications (discussed in the following section), and applications where your software will be performing error correction and detection anyway.

The application which will be receiving datagram packets should be initialized first. It then waits to receive any datagrams addressed to it. The **dg_read()** function defined in Code Box 3.7 can be used to receive datagrams. Its syntax is somewhat similar to the **read()**

function. You need to supply the function with the name table number assigned to your application name. This is the number returned from your call to **`init_netbios()`**. You also supply a pointer to a buffer area and the length of the buffer. Note that you *do not* provide the name of the application you expect to receive the datagram *from*. Upon successful receipt of a datagram, the function returns the number of bytes successful placed in the buffer. The adapter provides us this information by modifying the **`length`** field of the Network Control Block. The adapter also provides us with the name of the application sending the datagram (using the **`r_name`** field), which is returned as the **`from`** parameter. The **`from`** parameter should point to a block of memory at least 16 bytes long.

In the event of an error, **`dg_read()`** returns minus 1 and sets the global variable **`net_error`** equal to the NetBIOS error return code. **`net_error`** is an integer which should be declared above your main function. The possible return codes are defined in Section 3.7 of this chapter.

The **`dg_write()`** function, shown in Code Box 3.8, is used to send datagrams to another application. Unlike **`dg_read()`**, this function requires that you include both your own name table entry number *and the name of the destination*. The destination is addressed by a 16-character name, not a name table entry number. You also supply a pointer to a buffer containing the data to send and the number of bytes of data to send. Note that the syntax is similar to the standard **`write()`** function.

The maximum size of a datagram packet will vary from one adapter to the next. There is no 100 percent consistent method of determining the maximum packet size except to try various sizes until you find the one which is just barely too big. Code Box 3.9 illustrates a brute-force approach to performing this function. The function **`max_dg()`** returns the largest acceptable datagram size for the current adapter. This brute force approach is normally acceptable because this function only needs to be performed one time, so optimization is not

```
#include        <string.h>
#include        "netbios.h"

/*******************************************************************
*   dg_read - read a datagram over the network
*
*       Parameters:
*               number (in) - your name table address number
*               from (out)  - name of user sending datagram
*               buffer (in) - location to put received data
*               length (in) - maximum number of bytes to receive
*
*       Global:
*               net_error - used to store NetBIOS return code for error
*               processing.
*
*       Returns:
*               Number of bytes received for success, -1 for failure
*
*       Notes:
*               Number is the value returned from a successful init_netbios().
*
*               This code assumes that you are using a memory model which will
*               result in buffer being a far pointer.
*
*               From must point to a block of memory at least 16 bytes long.
*
*       History:
*               Original code by William H. Roetzheim, 1990
*******************************************************************/

int     dg_read(unsigned int number, char *from, char *buffer, int length)
{
        struct  net_control_block       ncb;

        init_ncb(&ncb);
        ncb.command = NCB_RECEIVE_DATAGRAM;
        ncb.length = length;
        ncb.buffer = buffer;
        ncb.number = number;
        int_netbios(&ncb);
        memcpy(from, ncb.l_name, 16);
        if (ncb.retcode == 0) return ncb.length;
        else return -1;
}
```

Code Box 3.6 **dg_read()** function definition.

```
#include        <string.h>
#include        "netbios.h"

/*****************************************************************
*   dg_write - write a datagram over the network
*
*       Parameters:
*               number (in) - your name table address number
*               destination (in) - destination name (1-15 characters)
*               buffer (in) - data to be transmitted
*               length (in) - number of bytes to transmit
*
*       Global
*               net_error - global integer used to return net error codes.
*               net_error is set to zero for normal return
*
*       Returns:
*                       Number of bytes transmitted for success.  -1 for failure.

*               In the event of failure, the global variable net_error is
*               set to the NetBIOS return code for error processing.
*
*       Notes:
*               Destination must have already executed an
*               NCB_RECEIVE_DATAGRAM command
*
*               Number is the value returned from a successful init_netbios().
*
*               This code assumes that you are using a memory model which will
*               result in buffer being a far pointer.
*
*       History:
*               Original code by William H. Roetzheim, 1990
*****************************************************************/

int     dg_write(unsigned int number, char *destination, char *buffer, int length)

{
        struct  net_control_block       ncb;
        char    dest_name[16];
        int     i;

        memset(dest_name, 0, 16);
        for (i = 0; i < 15; i++)
        {
                if (destination[i] == 0) break;
                else dest_name[i] = destination[i];
        }

        init_ncb(&ncb);
        ncb.command = NCB_SEND_DATAGRAM;
        ncb.length = length;
        ncb.buffer = buffer;
        ncb.number = number;
        strcpy(ncb.r_name, dest_name);
        int_netbios(&ncb);
        if (ncb.retcode == 0) return length;
        else return -1;
}
```

Code Box 3.7 dg_write() function definition.

```
#include        <string.h>
#include        "netbios.h"

/*****************************************************************
*   max_dg - Determine largest acceptable datagram size
*
*       Parameters:
*
*       Returns:
*                       Maximum valid datagram size in bytes
*
*       Notes:
*               This code assumes that you are using a memory model which will
*               result in buffer being a far pointer.
*
*       History:
*               Original code by William H. Roetzheim, 1990
******************************************************************/

int     max_dg()
{
        struct  net_control_block       ncb;
        int     length = 0;

        init_ncb(&ncb);
        while (ncb.retcode == 0)
        {
                length++;
                init_ncb(&ncb);
                ncb.command = NCB_SEND_DATAGRAM;
                ncb.length = length;
                ncb.number = 0x01;      /* use our adapter standard name */
                int_netbios(&ncb);
        }
        return length -1;
}
```

Code Box 3.8 **max_dg()** function listing.

very rewarding in terms of overall application performance. If you find the running time to be burdensome, you can easily
modify the function to use a binary search algorithm to find the largest acceptable size.

3.4 Session-Oriented Communication

Session-oriented communication is appropriate for the majority of NetBIOS oriented data communication. This protocol provides acknowledgments to give you some assurance that the data has been received intact. Session-oriented communication is logically similar to placing a telephone call. The steps involved are

1. You call a remote adapter (dial the number).
2. You send and receive data (talk and listen).
3. You terminate the session (hangup the phone).

We will describe four session-oriented communciation support functions. These functions are designed to operate similar to standard file I/O functions included with your C compiler. The functions we will define are

- **sn_open()** This function opens a connection
- **sn_write()** This function write data over a connection.
- **sn_read()** This function reads data over a connection.
- **sn_close()** This function closes a connection.

Code Box 3.10 shows the code for the **sn_open()** function. This function opens a logical session (connection) between two adapters. Two communicating application programs must both call **sn_open()**. One must call **sn_open()** while setting the value for the **flag** parameter to **CLIENT** while the other must set the **flag** parameter to **SERVER**. It does not matter which application is the client and which is the server, as long as they are not both one or the other. The names used to establish a session are the names used by each application program during their call to **init_netbios()**. **sn_open()** returns the logical session number for success, or minus 1 for error. If minus 1 is returned, the global variable **net_error** can be checked to determine the exact error number. You use **sn_open()** just like you would use **open()** and treat the return value just like you would treat a handle returned by **open()**.

After you have opened a connection using **`sn_open()`**, you can read and write data over the connection using **`sn_read()`** and **`sn_write()`**. These functions are defined in Code Box 3.11 and Code Box 3.12, respectively. The syntax is very similar to the syntax for the **`read()`** and **`write()`** functions you are already familiar with, except that
the file handle has been replaced with a logical session number (returned by **`sn_open()`**).

Although the syntax is similar to **`read()`** and **`write()`**, you must remember that there is a big difference between session oriented communications and reading or writing to/from a disk drive. When you are interacting with a disk drive, you are in command. The disk controller waits for your command, the either performs the read or write action as directed. When you are communicating over a network, you are no longer automatically in control. If you execute an **`sn_read()`**, *the other application program* must execute an **`sn_write()`** for anything to happen. Similarly, if you execute an **`sn_write()`**, the other application program must execute an **`sn_read()`** for communication to be effective. This coordination problem is often handled in one of two ways.

One approach is to establish a protocol for who should be sending and who should be receiving at any point in time. For some applications, there might be only one sender (the server) and one receiver (the client). For other applications, it might be appropriate for the application programs to take turns, alternating sending and receiving. Another approach is to establish an *in-band* or *out-of-band* coordination protocol. For an in band protocol, you might include some header information at the beginning of each transmission to tell if the receiver should continue to listen or should transmit after receipt of the data buffer. Out-of-band coordination might involve using communicating using datagrams (in addition to the session-oriented communication) for coordination information.

```
#include        <string.h>
#include        "netbios.h"

/*******************************************************************
*       sn_open - open a connection oriented session using NetBIOS
*
*       Parameters:
*               from (in) - your application name
*               to (in) - name of destination application
*               flag (in) - CLIENT or SERVER
*
*       Global
*               net_error - integer giving latest net error condition
*       Returns:
*               logical session number (LSN) on success, -1 on error
*
*       Notes:
*               On error, check net_error for error number
*
*       History:
*               Original code by William H. Roetzheim, 1990
********************************************************************/

int     sn_open(char *from, char *to, int flag)
{
        int     i;
        struct  net_control_block       ncb;
        char    name[16];

        init_ncb(&ncb);
        if (flag == SERVER) ncb.command = NCB_CALL;
        else ncb.command = NCB_LISTEN;
        memset(name,0,16);
        for (i = 0; i < 16; i++)
        {
                if (from[i] == 0) break;
                else name[i] = from[i];
        }
        strcpy(ncb.l_name, name);

        memset(name, 0, 16);
        for (i = 0; i < 16; i++)
        {
                if (to[i] == 0) break;
                else name[i] = to[i];
        }
        strcpy(ncb.r_name, name);
        int_netbios(&ncb);
        if (ncb.retcode == 0) return ncb.lsn;
        else return -1;
}
```

Code Box 3.9 sn_open( ) function definition.

```
#include        <stdio.h>
#include        "netbios.h"

/*******************************************************************
*       sn_read - read data from an already open logical session number
*
*       Parameters:
*               lsn (in) - logical session number from sn_open()
*               buffer (in) - far pointer to data buffer
*               nbytes (in) - available size of buffer area
*
*       Returns:
*               number of bytes actually received, or -1 for error
*
*       Notes:
*               On error, check net_error for error number
*
*       History:
*               Original code by William H. Roetzheim, 1990
********************************************************************/

int     sn_read(char lsn, void *buffer, unsigned int nbytes)
{
        struct  net_control_block       ncb;

        init_ncb(&ncb);
        ncb.command = NCB_RECEIVE;
        ncb.lsn = lsn;
        ncb.length = nbytes;
        ncb.buffer = buffer;
        int_netbios(&ncb);
        if (ncb.retcode == 0) return ncb.length;
        else return -1;
}
```

Code Box 3.10 sn_read() function definition.

An alternate approach, which is often more appropriate if extensive two-way communication is required, is to establish two sessions simultaneously. One session is used for transmitting data from application A to application B, while the other is used for transmitting data from application B to application A. Basically, each of the sessions is used as a simplex (one-way) communication link. One approach to accomplishing this is as follows:

1. Call **sn_open()** twice to establish two sessions. You do not need to use different names, nor do you need to change the value for the flag. It is completely appropriate, and normally best, to simply make two calls with the identical parameters.

```
#include        <stdio.h>
#include        "netbios.h"

/*******************************************************************
*       sn_write - write data to an already open logical session number
*
*       Parameters:
*               lsn (in) - logical session number from sn_open()
*               buffer (in) - far pointer to data to transmit
*               nbytes (in) - number of bytes to transmit
*
*       Returns:
*               number of bytes actually transmitted, or -1 for error
*
*       Notes:
*               On error, check net_error for error number
*
*       History:
*               Original code by William H. Roetzheim, 1990
**********************************************************************/

int     sn_write(char lsn, void *buffer, unsigned int nbytes)
{
        struct  net_control_block       ncb;

        init_ncb(&ncb);
        ncb.command = NCB_SEND;
        ncb.lsn = lsn;
        ncb.length = nbytes;
        ncb.buffer = buffer;
        int_netbios(&ncb);
        if (ncb.retcode == 0) return ncb.length;
        else return -1;
}
```

Code Box 3.11 `sn_write()` function definition.

2. Use **`sn_receive()`** (Code Box 3.13) rather than **`sn_read()`**. **`sn_receive()`** is modified as follows:

   - The NetBIOS command has been modified to use the **`NO_WAIT`** option.

   - The net_control_block variable has been modified to be a static variable (so that it will remain available after return from the function).

```
#include        <stdio.h>
#include        "netbios.h"

/*****************************************************************
*       sn_receive - initialize a receive operation in background
*
*       Parameters:
*               lsn (in) - logical session number from sn_open()
*               buffer (in) - far pointer to data buffer
*               nbytes (in) - available size of buffer area
*
*       Returns:
*               address of net control block used during receive
*
*       Notes:
*               Only one sn_receive() operation must be outstanding at a time
*
*       History:
*               Original code by William H. Roetzheim, 1990
******************************************************************/

struct net_control_block *sn_receive(char lsn, void *buffer, unsigned int nbytes)
{
        static  struct  net_control_block       ncb;

        init_ncb(&ncb);
        ncb.command = NCB_RECEIVE | NO_WAIT;
        ncb.lsn = lsn;
        ncb.length = nbytes;
        ncb.buffer = buffer;
        int_netbios(&ncb);
        return &ncb;
}
```

Code Box 3.12 **sn_receive()** function definition.

- The function returns the address of the net_control_block variable rather than an indication of the number of bytes received.

3. Use **sn_send()** (Code Box 3.14) rather than **sn_write()**. **sn_send()** received the same modifications as **sn_receive()**.

4. Call **sn_receive()** early in your application, using a buffer area set aside for this purpose. Save the return value as a pointer to the receive Network Control Block, perhaps calling it **receive_ncb**. Periodically check the **complete** field to determine if a message has arrived

```
#include        <stdio.h>
#include        "netbios.h"

/*********************************************************************
*       sn_send - write data out in background
*
*       Parameters:
*               lsn (in) - logical session number from sn_open()
*               buffer (in) - far pointer to data to transmit
*               nbytes (in) - number of bytes to transmit
*
*       Returns:
*               Address of net control block used for write
*
*       Notes:
*               Only one sn_send operation can be outstanding at a time
*
*       History:
*               Original code by William H. Roetzheim, 1990
**********************************************************************/

struct net_control_block *sn_send(char lsn, void *buffer, unsigned int nbytes)
{
        static  struct  net_control_block       ncb;

        init_ncb(&ncb);
        ncb.command = NCB_SEND | NO_WAIT;
        ncb.lsn = lsn;
        ncb.length = nbytes;
        ncb.buffer = buffer;
        int_netbios(&ncb);
        return &ncb;
}
```

Code Box 3.13 **sn_send()** function definition.

(**receive_ncb->complete** will be some value other than 0xFF). If a message has arrived, immediately copy the receive buffer contents to a working area, then call **sn_receive()** again for the next message.

5. Call **sn_send()** whenever you need to send data, using a buffer area set aside for this purpose (*not* the same buffer area used for **sn_receive()**!). Save the return value as a pointer to the send Network Control Block, perhaps calling it **send_ncb**. Periodically check the **complete** field to determine if the message has been successfully transmitted (**send_ncb->complete** will be

some value other than 0xFF). After the message has been sent, you may send another if desired.

Using this approach, it is possible to simultaneously conduct two-way data communication over the two data connections (sessions).

3.5 Miscellaneous NetBIOS Commands

NCB_RESET is used to reset the adapter. When the adapter is reset under DOS, all current NetBIOS names are deleted, all current sessions are aborted, and all outstanding Network Control Blocks are purged. In addition to the **command** field, three Network Control Block fields are used by **NCB_RESET**:

1. **adapter** This field indicates the adapter number to reset (0 for primary, 1 for secondary). **init_ncb()** sets this field to 0.

2. **lsn** This field is used to indicate the maximum number of sessions the adapter should support. This number cannot be greater than the maximum value specified as a load parameter when NetBIOS was loaded. Setting this field to 0 allows the adapter to select a logical number. The adapter will use 6 if the load parameter **RESET.VALUES** = no (default), or else it will use the load parameter maximum sessions. **init_ncb()** sets this field to 0.

3. **number** This field is used to indicate the maximum number of outstanding Network Control Block commands. This number can not be greater than the maximum value specified as a load parameter when NetBIOS was loaded. Setting this field to 0 allows the

adapter to select a logical number. The adapter will use 12 if the load parameter **RESET.VALUES** = no (default), or else it will us the load parameter maximum commands. **init_ncb()** sets this field to 0.

NCB_SESSION_STATUS is used to monitor the status of all currently active NetBIOS sessions on your adapter. The structure **session_status** (as defined in **netbios.h**) is used to examine the current status. The structure contains four fields which apply to all outstanding sessions for a given name along with a structure called **session,** which contains information for a single outstanding session. If we declared a structure of type **session status** (e.g., **struct session_status ss**), we would access the state field for a single session as **ss.session.state**. The **state** field within the structure has the following possible meanings:

0x01	- Listen outstanding
0x02	- Call pending
0x03	- Session established
0x04	- Hang up pending
0x05	- Hang up complete
0x06	- Session aborted

Because one application can have multiple sessions simultaneously outstanding, there may be more than one **session** portion of the **session_status** structure. Code Box 3.15 shows one possible approach to handling this possibility. The function **get_session_status()** is initially called with the **FIRST** parameter. A pointer to a **session_status** structure is returned and the **session** portion of the structure is initialized to the first session. **get_session_status()** can then be called repeatedly using the **NEXT** parameter. Each call replaces the **session** portion of the structure with the next session. When all sessions have been viewed,

```
#include        <stdio.h>
#include        <string.h>
#include        "netbios.h"

extern  int             net_error;
#define MAX_SESSIONS    12
#define BUFFER_SIZE     (4 + (36 * MAX_SESSIONS))
/*****************************************************************
*       get_session_status - get session status information
*
*       Parameters:
*               name (in) - name to inquire about
*               flag (in) - FIRST or NEXT
*
*       Global:
*               net_error - set if problem encountered
*
*       Returns:
*               Pointer to session_status structure, or NULL when no
*               more information
*       History:
*               Original code by William H. Roetzheim, 1990
*****************************************************************/
struct  session_status *get_session_status(char *name, int flag)
{
        int             i;
        static  int     location;
        char            application_name[16];
        static  char    buffer[BUFFER_SIZE];
        static  struct  session_status ss;
        struct  net_control_block       ncb;

        if (flag == FIRST)
        {
                memset(buffer, 0, BUFFER_SIZE);
                /* read status information into buffer */
                init_ncb(&ncb);
                ncb.command = NCB_SESSION_STATUS;
                memset(application_name, 0, 16);
                for (i = 0; i < 15; i++)
                {
                        if (name[i] == 0) break;
                        else application_name[i] = name[i];
                }
                strcpy(ncb.l_name, application_name);
                ncb.length = BUFFER_SIZE;
                ncb.buffer = buffer;
                int_netbios(&ncb);
                /* copy initial portion to session status structure */
                memcpy(&ss, buffer, sizeof(struct session_status));
                location = sizeof(struct session_status);
        }
        else
        {
                if (location < 4 + (ss.number_of_sessions * 36))
                {
                        memcpy(&ss.session, &buffer[location],
                                sizeof(struct session_status));
                        location += sizeof(struct session_status);
                }
                else location = -1;     /* past end */
        }
        if (location != -1) return &ss;
        else return NULL;
}
```

Code Box 3.14 `get_session_status()` function definition.

a NULL is returned.

Sections 3.6 and 3.7 summarize all NetBIOS commands and return values. Some less common NetBIOS commands were not covered earlier in this chapter, so you should read these two sections to be aware of all available NetBIOS commands. If you need to work with the NetBIOS at a more detailed level than covered in this chapter, you should refer to Section 3.8 (Suggested Readings) for additional details.

3.6 NetBIOS Command Summary

The following table presents a summary of all NetBIOS commands. The columns have the following meanings:

1. **Command** The command name. These names are defined in **netbios.h**. These are the values to use for the Network Control Block's **command** field prior to calling the NetBIOS for processing. All of these commands can be used with the **NO_WAIT** flag (i.e., **COMMAND | NO_WAIT**) except for **NCB_CANCEL**, **NCB_LAN_STATUS_ALERT**, **NCB_RESET**, and **NCB_UNLINK**.

2. **Input** The fields within the Network Control Block (in addition to **command**) that should be initialized prior to using the command.

3. **Outputs** The fields within the Network Control Block that are modified by the command during processing.

4. **Summary** A brief description of the command function.

Command	Inputs	Outputs	Summary
NCB_ADD_GROUP_NAME (0x36)	adapter l_name post	retcode number reserved	Add shared name to adapter name table.
NCB_ADD_GROUP_NAME (0x30)	adapter l_name post	retcode number reserved	Add unique name to adapter name table.
NCB_CALL (0x10)	adapter l_name r_name post rto sto	retcode lsn reserved	Call to establish session oriented connection.
NCB_CANCEL (0x35)	adapter buffer	retcode reserved	Cancel command located at buffer.
NCB_CHAIN_SEND (0x17)	adapter length buffer post lsn r_name (bytes 0-1 = length2, bytes 2-5 = *buffer2)	retcode reserved	Send one buffer, then immediately send a second buffer.
NCB_CHAIN_SEND_NO_ACK (0x72)	adapter length buffer post lsn r_name (bytes 0-1 = length2, bytes 2-5 = *buffer2)	retcode reserved	Send on buffer, then immediately send a second buffer. Do not request acknowledgments.
NCB_DELETE_NAME (0x31)	adapter l_name post	retcode reserved	Delete a name from the adapter's name table.
NCB_FIND_NAME (0x78)	adapter length buffer post r_name	retcode reserved length	Find the address of any adapters which have registered a specific name.
NCB_HANG_UP (0x12)	adapter lsn post	retcode reserve	Close a connection oriented session.

NCB_LAN_STATUS_ALERT (0xF3)	post	retcode	Used to notify an application in the event of low- level ring errors.
NCB_RECEIVE (0x15)	adapter lsn buffer post length	retcode length reserved	Receive connection oriented data via a session.
NCB_RECEIVE_ANY (0x16)	adapter length buffer post number	retcode lsn length reserved number	Receive connection oriented data from any session.
NCB_RECEIVE_BROADCAST_DATAGRAM (0x23)	adapter length buffer post number	retcode reserved length r_name	Receive a datagram from any name on the network sending a broadcast datagram.
NCB_RECEIVE_DATAGRAM (0x21)	adapter length buffer post number	retcode reserved length r_name	Receive a datagram addressed to number.
NCB_RESET (0x32)	adapter lsn number	retcode reserved	Clear the adapter name table, abort all sessions, purge all outstanding NCB's, and open the adapter.
NCB_SEND (0x14)	adapter length buffer post lsn	retcode reserved	Send data to the session partner (identified by lsn).
NCB_SEND_BROADCAST_DATAGRAM (0x22)	adapter length buffer post number	retcode reserved	Send a datagram to every station with an outstanding NCB_RECEIVE_BROADCAST_DATAGRAM.
NCB_SEND_DATAGRAM (0x20)	adapter length buffer post number r_name	retcode reserved	Send a datagram to a specific name identified by r_name.

NCB_SEND_NO_ACK (0x71)	adapter length buffer post lsn	retcode reserved	Send data to a session partner without requiring an acknowledgment upon receipt.
NCB_SESSION_STATUS (0x34)	adapter length buffer post l_name	retcode reserved length	Obtain the status of all sessions for a local name or all sessions for all local names.
NCB_STATUS (0x33)	adapter buffer length r_name post	retcode reserved length	Query the status of a local or remote NetBIOS.
NCB_TRACE (0x79)	adapter buffer length number (0xFF = trace on, 0x00 = local trace off, 0x01 = local and remote trace off)	retcode reserved buffer length	Activate or deactivate a trace of all Network Control Blocks.
NCB_UNLINK (0x70)	adapter	retcode reserved	Provided for compatibility only. Performs no function.

NetBIOS Command Specifics

NCB_ADD_GROUP_NAME

Add a group name to the local adapter name table. Group names can be used by more than one adapter (or application on an adapter). Group names can not also be registered as unique names. The name must be 16 characters, although the last character must not be in the range of 0x00 through 0x1F and the first three characters cannot be "IBM." The add name request is processed by transmitting name query requests over the network and monitoring any responses. When successful, the command returns the name table entry number. This number is assigned between 0x02 and 0xFE (0x00 and 0xFF are not used, 0x01 is permanently assigned based on the adapter's unique serial number). If more than 252 names are registered, the later names overwrite the earlier names (i.e., the numbers roll over). See also NCB_ADD_NAME, NCB_DELETE_NAME, NCB_FIND_NAME.

NCB_ADD_NAME

This command works identically to **NCB_ADD_GROUP_NAME** except that the name must be unique across the network. See also NCB_ADD_GROUP_NAME, NCB_DELETE_NAME, NCB_FIND_NAME.

NCB_CALL

Establish a session by calling a remote application. The remote application must have an **`NCB_LISTEN`** outstanding for this command to succeed. The session is opened with the application that has a registered name of **`r_name`**. Multiple sessions may be established between the same pair of names. Timeout intervals (**`rto`** and **`sto`**) are 500 millisecond units, with 0 implying no timeout. Upon success, a local session number (**`lsn`**) is returned. Lsn's are assigned in a round-robin technique in the range of 0x01 – 0xFE. See also NCB_LISTEN, NCB_HANG_UP, NCB_SEND, NCB_RECEIVE.

NCB_CANCEL

Cancel a Network Control Block (command). The command which is cancelled is located at the address pointed to by **`buffer`**. Canceling any session oriented command will automatically close the session. The following commands can be canceled:

- NCB_CALL
- NCB_CHAIN_SEND
- NCB_CHAIN_SEND_NO_ACK
- NCB_HANG_UP
- NCB_LAN_STATUS_ALERT
- NCB_LISTEN
- NCB_RECEIVE
- NCB_RECEIVE_ANY
- NCB_RECEIVE_BROADCAST_ANY
- NCB_RECEIVE_DATAGRAM
- NCB_SEND
- NCB_SEND_NO_ACK
- NCB_STATUS

See also NCB_SESSION_STATUS, NCB_STATUS.

NCB_CHAIN_SEND

This command allows two buffers to be automatically concatenated together and sent at once. The meaning of the **`r_name`** field is modified to contain the following information:

- Bytes 0 – 1 (length of second buffer)
- Bytes 2 – 5 (far pointer to second buffer)

Lengths between 1 and 65,535 are valid for *each* of the two buffer, allowing this command to send up to 131,070 bytes. See also NCB_CHAIN_SEND_NO_ACK, NCB_SEND, NCB_RECEIVE.

NCB_CHAIN_SEND_NO_ACK

This command works identically to **`NCB_CHAIN_SEND`**, except that the receiving adapter is not required to send acknowledgments back (this becomes an application responsibility). See also NCB_CHAIN_SEND.

NCB_DELETE_NAME

Delete a name from the local name table. When data is queued for transmission or reception over a session using this name, the actual name deletion is delayed until the data transmission/reception is complete. See also NCB_ADD_NAME.

NCB_FIND_NAME

Find the network location of the adapter owning a 16 character name, including how the name is registered (unique or group). If the name is not found, the **`retcode`** field is set to 0x05 (command timed out). If one or more adapters did respond, the **`length`** field is set to the length of the returned data. Data is placed at the location pointed to by **buffer**. The number of responses will be the first two bytes located at **buffer**. This number will always be 0x01 unless the name is registered as a group name. The format of the remainder of the data in the buffer varies and is described fully in (IBM, 1988).

NCB_HANG_UP

This command closes a session (connection) as specified by the local session number (**lsn**). The command will complete any Network Control Block which is in the process of being sent, but will cancel all other outstanding Network control blocks destined for this session. See also NCB_CALL, NCB_LISTEN, NCB_CANCEL.

NCB_LAN_STATUS_ALERT

This command *always* runs in the **NO_WAIT** mode. As long as the token ring operates properly, this command is queued by NetBIOS (does not return). The command complete when a ring error condition occurs which lasts longer than one minute. This command can be used for network administration or network management software. See also NCB_SESSION_STATUS, NCB_STATUS.

NCB_LISTEN

This command enables a session to begin with the application identified as **r_name**. If the first character of the **r_name** is an asterisk ('*'), a session will begin with any network node that calls this application. An **NCB_LISTEN** for a specific name will preempt data over an **NCB_LISTEN** for a wildcard name. **rto** and **sto** are the timeout intervals in 500 millisecond increments, with 0 implying no timeout. This command returns a local session number (**lsn**). If a wildcard name was used, the **r_name** is also modified to be the actual name of the network node performing the **NCB_CALL**. See also NCB_CALL, NCB_RECEIVE.

NCB_RECEIVE

After a session is established, this command is used to receive data from a session partner. This command receives data sent using

- NCB_CHAIN_SEND
- NCB_CHAIN_SEND_NO_ACK
- NCB_SEND
- NCB_SEND_NO_ACK

NCB_RECEIVE has priority over **NCB_RECEIVE_ANY**. See also NCB_SEND, NCB_RECEIVE_ANY, NCB_LISTEN.

NCB_RECEIVE_ANY

This command receives data for any session registered to the network node identified by **number** in the name table. If **number** is set to 0xFF, this command will receive data for any session addressed to any name in the local adapter. The **NCB_RECEIVE** command has priority over **NCB_RECEIVE_ANY**. It is possible to use this command and receive data destined for a different application running on the local computer. See also NCB_RECEIVE.

NCB_RECEIVE_BROADCAST_DATAGRAM

This command receives a broadcast datagram from any application on the network which issued a

SEND_BROADCAST_DATAGRAM. The buffer length (specified by **length**) must be large enough to receive the entire datagram or the remaining data will be lost. See also NCB_RECEIVE_DATAGRAM, NCB_RECEIVE_ANY.

NCB_RECEIVE_DATAGRAM

This command receives a datagram from any name on the network that issues an **NCB_SEND_DATAGRAM** to the local name table entry given by **number**. If **number** is set to 0xFF, then datagrams addressed to *any* name in the local name table will be received. This command will *not* receive a broadcast datagram. See also NCB_RECEIVE_BROADCAST_DATAGRAM, NCB_RECEIVE.

NCB_RESET

The exact functioning of **NCB_RESET** is dependent on the version of NetBIOS you are running and whether or not you are running under OS/2. For all variations, **NCB_RESET** deletes NetBIOS names from the name table, closes current sessions, purges all outstanding NCBs, and turns the adapter on. Under DOS, *all* names and sessions are closed, while under OS/2 only those names and sessions specific to your local process are closed. See also NCB_CANCEL.

NCB_SEND

This command sends the data located at **buffer** to the session partner defined using the local session number (**lsn**). If more than one **NCB_SEND** is pending, buffers are transmitted in a FIFO order. Message buffers can be between 0 and 65,535 bytes long, with the length of the buffer passed in **length**. If the **NCB_SEND** cannot be completed the session is closed. See also NCB_RECEIVE, NCB_SEND_NO_ACK.

NCB_SEND_BROADCAST_DATAGRAM

This command sends a broadcast datagram to every station with an **NCB_RECEIVE_BROADCAST_DATAGRAM** outstanding. If the station transmitting the broadcast datagram also has an **NCB_RECEIVE_BROADCAST_DATAGRAM** outstanding, it will receive its own transmission. Receipt of one broadcast datagram satisfied *all* outstanding **NCB_RECEIVE_BROADCAST_DATAGRAM** commands (multiple commands are not queued). See also NCB_RECEIVE_BROADCAST_DATAGRAM.

NCB_SEND_DATAGRAM

This command sends a datagram to any unique name or group name on the network. The destination is shown in the **r_name** field. The source (your name table entry) is passed in the **number** field. See also NCB_RECEIVE_DATAGRAM.

NCB_SEND_NO_ACK

This command works exactly like **NCB_SEND**, except that the recipient of the data is not required to transmit an acknowledgment back (this becomes an application responsibility). See also NCB_SEND.

NCB_SESSION_STATUS

This command is used to determine the status of all sessions for a local name (it can also be used to determine the status for all sessions for *all* local names). To return the status for all local names, the first character of the **l_name** field must be an asterick ('*'). Normally, the **l_name** field will tell the local name you are interested. The space pointed to by buffer is used to store data, and its maximum size (shown in **`length`**) should be at least 4 bytes plus 36 times the maximum number of sessions you expect to be returned. The format of the returned data was described in Section 3.6. See also NCB_STATUS.

NCB_STATUS

This command returns the current status of a local or remote NetBIOS. To return the local status, the first byte of the **`r_name`** field must be an asterick ('*'). To return the status of a remote adapter, the **`r_name`** field should contain the name of the adapter. The status information is returned to the memory location identified by **`buffer`**. The length of the buffer (designated in **`length`**) must be at least 60 for this command to succeed. If you want to receive all available information, the length of the buffer must be 60 plus 18 times the maximum number of names registered for the adapter. The structure of the returned information is shown in Code Box 3.16. This command may not be available on non-IBM versions of NetBIOS, or if available, the returned information may be different. See also NCB_SESSION_STATUS.

NCB_TRACE

This command activates and deactivates a trace of all Network Control Blocks processed by NetBIOS, including both transmits and receives. This command only available under DOS. The **`number`** field is used to determine the action of the command (0xFF = trace on, 0x00 = local trace off, 0x01 = local and all remote traces off). The field **`length`** is set to the length of your trace table (1024 bytes or larger) and **`buffer`** points to the start of the trace table. The trace table contains a 32 byte trace table header followed by by each trace entry. The exact format of the trace table is somewhat complex, and

```
struct  ncb_status
{
    char         adapter[6];       /* encoded adapter address */
    char         release;          /* NetBIOS version 1, 2, or 3 */
    char         reserved_1;
    char         netbios_1;        /* 0xFF = Token Ring Adapter */
                                   /*  0xFE = PC Network Adapter */
    char         netbios_2;        /* NetBIOS version 1, software level */
                                   /* NetBIOS version 2/3 */
                                   /* bits 0-3 = software version;  */
                                   /* bits 4-7 = 0x1 */
                                   /* for old parameters */
                                   /* 0x2 for new parameters */
    unsigned     duration;         /* Duration of rep period in minutes */
    unsigned     f_received;       /* Number of frames received */
    unsigned     f_sent;           /* Number of frames transmitted */
    unsigned     f_rec_error;      /* Number of receive frames in error */
    unsigned     f_aborted;        /* Number of transmissions aborted */
    unsigned     long    packets_sent;    /* Number of successfully */
                                   /*  transmitted packets */
    unsigned     long    packets_rec;     /* Number of successfully */
                                   /*  received packets */
    unsigned     f_sent_error;     /* Number of transmit frames in error */
    unsigned     long    reserved_2;
    unsigned     free_ncb;         /* Number of free net control blocks */
    unsigned     max_ncb;          /* Maximum number of net control blocks */
    unsigned     max_ncb_poss;     /* Maximum number of net control blocks */
    unsigned     buf_not_avail;    /* Number of times a xmit buf was not avail */
    unsigned     max_datagram;     /* Maximum datagram size */
    unsigned     pend_ses;         /* Number of pending sessions */
    unsigned     max_pend;         /* Configured Max number of pending sessions */
    unsigned     max_pend_poss;    /* Maxinum number of pending sessions */
    unsigned     max_packet;       /* Maximum size of session data packet */
    unsigned     tot_names;        /* Number of names in the local name table */
    struct
    {
        char     name[16];
        char     number;
        struct
        {
            unsigned int    type : 1;         /* 0x00 = unique, 0x01 = group */
            unsigned int    reserved_3 : 4;   /* bit field */
            unsigned int    status : 3;       /* 0x0 being registered */
                                              /* 0x4 registered */
                                              /* 0x5 deregistered */
                                              /* 0x6 detected duplicate */
                                              /* 0x7 detected dup, pending */

        } bit;
    } name_table_entry[];
}
```

Code Box 3.15 NCB_STATUS return structure.

interested readers are referred to (IBM, 1988) for the details. See also NCB_STATUS, NCB_SESSION_STATUS.

NCB_UNLINK

This command acts like a NOP for IBM NetBIOS implementations. It is provided for compatibility reasons only.

3.7 NetBIOS Return Code Summary

This section lists all return codes which are valid for the NetBIOS. Return codes are returned in the **`retcode`** field of the Network Control Block structure. If you are using the **`init_netbios()`** function defined earlier in this chapter, the return code is also placed in the global variable **`net_error`**. A return of 0x00 is always a valid return without error. A return of anything other than 0x00 indicates some type of error. The specific meaning of each possible return code is as follows:

Code	Name	Description	Action
0x00	SUCCESS	Operation completed normally.	None.
0x01	BUF_LENGTH	The buffer length passed in the Network Control Block was invalid.	Modify the length.
0x03	INVALID_COMMAND	Invalid NetBIOS command.	Modify command.
0x05	TIME_OUT	Command timed out.	Check that a receive is outstanding for any send command. Retry.
0x06	BUFFER_SIZE	Received data could not fit in buffer.	NCB_RECEIVE and NCB_RECEIVE_ANY - reissue command to get remaining data. Other commands, the data is lost.

0x07	BAD_PACKET	One or more data packets send using NCB_SEND_NO_ACK or NCB_CHAIN_SEND_NO_ACK was not properly received.	Application-level error recovery is required.
0x08	INVALID_LSN	The local session number (lsn) specified is not valid.	Use correct lsn.
0x09	RSESSION_FULL	The remote application program does not have any available sessions remaining to establish a new session.	Try again later.
0x0A	SESSION_CLOSED	The transmitting side closed the session (this is the normal return code in this case).	None.
0x0B	CANCELLED	Command was cancelled.	None.
0x0D	DUP_NAME	Specified name is already in use.	Use a different name.
0x0E	NAME_FULL	The name table is full or has exceeded the number defined at initialization (default = 17).	Delete a name.
0x0F	SESSION_ACTIVE	The name was deregistered, but active sessions are outstanding.	Close all sessions using this name.
0x11	LSESSION_FULL	The local session table is full. The number of sessions can be modified at initialization or using NCB_RESET.	Close a session.
0x12	NO_LISTEN	The remote node does not have an outstanding NCB_LISTEN command.	Wait until an NCB_LISTEN is outstanding.
0x13	INVALID_NUMB	The name table number is not valid.	Use a valid number.
0x14	NO_RESPONSE	No response to NCB_CALL command.	Check receiving application for proper operation.
0x15	INVALID_NAME	The name was not found, contains an asterick ('*') as the first charcter, or contains a 0x0 as the first character.	Use a valid name.

0x16	NAME_IN_USE	The name is already registered by a remote NetBIOS.	Use a different name.
0x17	NAME_DEL	The name was already deleted.	Add the name to the name table.
0x18	SESSION_FAILURE	The session was terminated abnormally. Normally this occurs when a sending command times out while waiting for a receive command to be posted.	Use a longer timeout interval.
0x19	NAME_CONFLICT	Two or more identical names have been detected on the network.	Delete the identical names.
0x21	BUSY	NetBIOS is busy or out of local resources.	Try again later.
0x22	TOO_MANY_NCBS	Too many Network Control Blocks are outstanding.	Try later or increase the maximum.
0x23	INVALID_ADAPTER	An invalid adapter number was specified.	Use 0x00 for primary, 0x01 for secondary (if installed).
0x24	COMPLETED	Tried to cancel a command which has already been completed.	None.
0x26	CANT_CANCEL	Attempt to cancel a command which can not be cancelled.	None.
0x30	NAME_DEFINED	Another environment has already defined the name.	Use a different name.
0x34	NO_ENVIRONMENT	The environment has not been defined.	Issue an NCB_RESET command.
0x35	OS_OVERFLOW	The operating system resources are exhausted.	Retry later.
0x36	APP_OVERFLOW	The maximum number of applications defined at load time (NetBIOS 3.0 only) are already running (OS/2).	Wait until another application terminates.
0x37	NO_SAPS	The adapter has no SAPs available for NetBIOS (OS/2).	Try later.
0x38	RESOURSE_AVAIL	The requested resource is not available (OS/2).	Use a smaller number of resources.

0x39	INVALID_NCB	The Network Control Block address is invalid or its length will not fit in a segment. For this error, the return value is placed in register AL but not in the Network Control Block itself.	Correct NCB address.
0x3A	RESET_INVALID	An NCB_RESET command was issued while the adapter was processing a hardware interrupt (device driver OS/2 interface using NetBIOS 3 only).	Correct application driver.
0x3B	INVALID_DD_ID	The device driver identification was invalid (OS/2 using device driver interface only).	Correct DD_ID value.
0x3C	LOCK_FAILED	NetBIOS attempted to lock user storage (file locking) and lock failed.	Try later.
0x3F	DD_OPEN_FAIL	A device driver open failure. Either the device driver did not function properly or the NetBIOS device driver was not loaded.	Ensure that NetBIOS is initialized properly with all required device drivers.
0x40	OS2_ERROR	OS_2 indicates an operating system error.	Issue NCB_RESET and try again.
0x4E	NET_STATUS_1	Network hardware failure, bits 12, 14, or 15 indicate failure.	Issue NCB_RESET.
0x4F	NET_STATUS_2	Network hardware failure, bits 8 — 11 indicate failure.	Issue NCB_RESET.
0xF6	CCB_ERROR	Unexpected error on CCB completion (low level protocol error).	Issue NCB_RESET.
0xF7	DIR_INIT_ERROR	Error attempting to perform DIR_INITIALIZE (DLC command).	Issue NCB_RESET.
0xF8	DIR_OPEN_ERROR	Error attempting to perform DIR_OPEN command.	Issue NCB_RESET.
0xF9	INTERNAL_ERROR	NetBIOS support software internal error.	Issue NCB_RESET.

0xFA	ADAPTER_ERROR	Adapter hardware error.	Issue NCB_RESET.
0xFB	BAD_NETBIOS	The NetBIOS code is either not loaded or invalid.	Load NetBIOS.
0xFC	DLC_ERROR	Error attempting a DIR_OPEN_ADAPTER or DLC_OPEN_SAP.	Issue NCB_RESET.
0xFD	ADAPTER_CLOSED	The adapter was closed while NetBIOS was executing.	Issue NCB_RESET.
0xFE	NO_NETBIOS	The application program explicitly opened the adapter while NetBIOS was not operational.	Close the adapter and reissue the NetBIOS command.

3.8 Suggested Reading

Glass, B. (1989), "Understanding NetBIOS," *Byte*, Vol. 14, no. 1, (Jananuary), pp. 301–306.

IBM (1987), *NetBIOS Application Development Guide*, Research Triangle Park, NC, International Business Machine Corporation.

IBM (1988), *Local Area Network Technical Reference*, Research Triangle Park, NC: International Business Machine Corporation.

Schwaderer, W. David (1988), *C Programmer's Guide to NetBIOS*, Indianapolis, IN: Howard W. Sams.

4. Speed with DLC Programming

When a token ring network adapter card is installed, the user modifies his/her *config.sys* file to install two new device drivers. *Dxma0mod.sys* is a token ring network interrupt arbitrator and *dxmc0mod.sys* is the adapter support device driver. These two drivers plus firmware on the adapter card itself provide full support for the IEEE 802.2 Logical Link Control (LLC) services, which are called Data Link Control (DLC) services by IBM. The DLC support is normally available on any computer with a token ring network adapter installed, and requires a minimal amount of system RAM (less than 16 Kbytes). DLC support provides the programmer with both connection oriented service (guaranteed delivery) and connectionless service (datagrams). The NetBIOS (and all higher services) translate user requests into appropriate DLC commands and use DLC for all actual network operations.

Although DLC programs run fast, they are somewhat more difficult to program than NetBIOS programs. You may also find that DLC programs are less portable in the PC environment than NetBIOS (although they may actually be *more* portable to some wide area network environments). Using DLC services is very similar to using NetBIOS. Instead of a Net Control Block (NCB), the DLC interface uses a *Command Control Block* (CCB). As with the NCB, the CCB contains both the command and the pass parameters. The CCB is then executed using interrupt 0x5C. You might wonder how the same interrupt (0x5C) can be used both for NetBIOS interrupts and for DLC programming? The answer is that the adapter support software looks at the first byte of the memory block that is passed to it. If the first byte is a 0x00 or 0x01, it assumes that the block is a CCB and processes it accordingly. If the first byte is greater than 0x03, it assumes that the block is an NCB and passes the block on to NetBIOS for processing (0x02 and 0x03 are reserved and return an error).

In this chapter, we begin by studying the CCB in more detail. We then discuss addressing using the DLC interface, which is considerably different from the addressing we used with NetBIOS. We are then prepared to discuss adapter initialization, connectionless communication, connection oriented communication, and adapter shutdown. Finally, we conclude the chapter with a summary of DLC commands and a description of return values.

The following functions are defined in this chapter:

`init_ccb()` Initialize a Command Control Block (CCB).

`int_adapter()` Interrupt the adapter and instruct it to execute a CCB.

`init_adapter()` Initialize the adapter to prepare it for communication.

`open_sap()` Open a SAP (defined later) for communication.

`build_lan_header()` Build a token ring network LAN header for use with connectionless DLC services.

`transmit_ui_frame()` Transmit a datagram oriented frame (data packet).

`receive_dlc()` Receive a datagram or connection oriented DLC frame.

`receive_process()` Interrupt handler for incoming DLC data frames.

`buffer_free()` Return an adapter buffer list to the buffer pool.

`close_sap()` Close a SAP.

`open_station()` Open a link access station for connection oriented communication using DLC.

`connect_station()` Establish a connection.

`xmit_i_frame()` Transmit a connection oriented frame (data packet) using DLC.

4.1 DLC Command Control Block Structure

To execute DLC commands, a Command Control Block (CCB) is used. Code Box 4.1 shows the format for all CCBs. The fields within the structure have the following meanings:

adapter: This field is set to 0x00 if the primary adapter is to be used, or 0x01 for the secondary adapter. Secondary adapters are only used on PCs which are acting as a gateway (or bridge) between networks.

command: This field defines the command to be performed. Valid commands are shown in the **`dlc.h`** header file later in this section, and are discussed in the remainder of this chapter. 0xFF is permanently defined as an invalid command code.

retcode: This command is set (by the adapter) to 0xFF while the command is pending. Upon completion, it is set to 0x00 for uccess or an error number for failure.

```
struct  command_control_block
{
        char    adapter;
        char    command;
        char    retcode;
        char    work;
        void    *queue;
        void    *post;
        void    *parameters;
};
```

Code Box 4.1 Command control block structure definition.

work: This field is a buffer for internal use by the adapter.

queue: While processing, this field is used internally by the adapter. When the command is complete, this field may contain a pointer to a CCB queue (a queue of CCBs). This capability is used when sending DLC "I" frames and using the `post` field (as discussed next).

post: This field contains a far pointer to an interrupt processing function to be called upon completion of the command. When your function is called, the address of the CCB block will be found in registers ES and BX and the `retcode` field will be copied to the AL register. If the CCB indicates that a DLC "I" frame has been acknowledged, the `queue` field may point to the next CCB in a list of CCBs which have all been acknowledged. This approach is used because one acknowledgment may acknowledge an entire series of "I" frames.

parameters: Most DLC commands require additional parameters. If these parameters require four or fewer bytes, they are passed in this field. If they require more than four bytes, this field contains a far pointer to a buffer containing the additional parameters.

The following listing shows the contents of the **`dlc.h`** header file. This header file contains the DLC commands which are available with the IBM token ring network, our function prototypes, and some miscellaneous defines included to improve the clarity of function code.

```
#include        "paramblk.h"

/* DLC Related Commands */
#define DIR_INTERRUPT                   0x00
#define DIR_OPEN_ADAPTER                0x03
#define DIR_CLOSE_ADAPTER               0x04
#define DIR_INITIALIZE                  0x20

#define DLC_RESET                       0x14
#define DLC_OPEN_SAP                    0x15
#define DLC_CLOSE_SAP                   0x16
#define DLC_OPEN_STATION                0x19
#define DLC_CLOSE_STATION               0x1A
#define DLC_CONNECT_STATION             0x1B
#define DLC_MODIFY                      0x1C
#define DLC_FLOW_CONTROL                0x1D
#define DLC_STATISTICS                  0x1E

#define RECEIVE                         0x28
#define RECEIVE_CANCEL                  0x29
#define RECEIVE_MODIFY                  0x2A

#define TRANSMIT_DIR_FRAME              0x0A
#define TRANSMIT_I_FRAME                0x0B
#define TRANSMIT_UI_FRAME               0x0D
#define TRANSMIT_XID_CMD                0x0E
#define TRANSMIT_XID_RESP_FINAL         0x0F
#define TRANSMIT_XID_RESP_NOT_FINAL     0x10
#define TRANSMIT_TEST_CMD               0x11

#define BUFFER_FREE                     0x27
#define BUFFER_GET                      0x26

#define PDT_TRACE_ON                    0x24
#define PDT_TRACE_OFF                   0x25

/* define received data message types */
```

```
#define MT_MAC                          0x02
#define MT_I                            0x04
#define MT_UI                           0x06
#define MT_XID_CP                       0x08
#define MT_XID_CNP                      0x0A
#define MT_XID_RF                       0x0C
#define MT_TEST_RF                      0x0C
#define MT_TEST_RNF                     0x12
#define MT_OTHER                        0x14

/* general defines for readability */
#define NO_ADAPTER                      0
#define INIT_FAILURE                    1
#define OPEN_FAILURE                    2

#define WAIT                            0
#define NO_WAIT                         1

#define NOT_RECEIVED                    1
#define OVERLOAD                        2
#define OTHER_ERROR                     3

/* limitations */
#define DLC_MAX_SAP                     2
#define DLC_MAX_STATIONS                6
#define WORK_AREA_SIZE   (48+(36*DLC_MAX_SAP) + (6*DLC_MAX_STATIONS))


struct  command_control_block
{
        char    adapter;
        char    command;
        char    retcode;
        char    work;
        void    *queue;
        void    *post;
        void    *parameters;
};


/* function prototypes */
void            init_ccb(struct command_control_block *ccb);
void            int_adapter(struct command_control_block *ccb, int wait);
unsigned int    init_adapter(void);
unsigned int    open_sap(int sap, int resv_link);
void            *buff[illegible](unsigned int station_id, int number);
void            build_lan_header(char destination[6], void *buffer);
int             transmit_ui_frame(unsigned int station_id, unsigned int sap,
                char destination[6], unsigned int data_len, char *data);
int             receive_dlc(unsigned int station_id);
void interrupt  receive_process();
unsigned int    buffer_free(unsigned int station_id, void *buffer);
unsigned int    close_sap(unsigned int station_id);
unsigned int    open_station(unsigned int station_id, unsigned int sap,
                                char destination[6]);
unsigned int    connect_station(unsigned int station_id);
int             xmit_i_frame(unsigned int station_id, unsigned int sap,
                unsigned int data_len, char *data);
```

In addition, a number of parameter structures will be used throughout these chapters. These structures are defined as follows (and described fully as they are used during the remainder of the chapter):

```
struct  dir_initialize_parameters
{
        unsigned int    bring_ups;          /* Not normally used */
        unsigned int    sram_address;       /* 0 will default to 0xD800 for */
                                            /* adapter 0, 0xD400 for adapter 1 */

        char     work[4];                   /* Work space */
                                            /* --- Interrupt function pointers --- */
        void     (*adapter_error);          /* Adapter error handler */
        void     (*netw_status_error);      /* Network error */
        void     (*pc_error);               /* Operating system or PC hardware */
};

struct dir_open_adapter_parameters
{
        struct  adapter_parms     *ap;      /* pointer to adapter param table */
        struct  direct_parms      *dp;      /* pointer to direct param table */
        struct  dlc_parms         *dlcp;    /* pointer to dlc param table */
        struct  ncb_parms         *ncbp;    /* pointer to NetBIOS param table */
};

struct  adapter_parms     /* used by dir_open_adapter command */
{
        unsigned int      open_error_code;          /* return - set by adapter */
        unsigned int      open_options;             /* 16 bit flags */
        char              node_address[6];          /* this node's address */
        char              group_address[4];         /* set group address */
        char              functional_addr[4];       /* set functional address */
        unsigned int      number_rcv_buffers;       /* number of receive buffers */
        unsigned int      rcv_buffer_len;           /* length of each receive buffer */
        unsigned int      dhb_buffer_length;        /* length of transmit buffers */
        unsigned char     data_hold_buffers;        /* number of transmit buffers */
        char              reserved;
        unsigned int      open_lock;                /* protection code */
        void              *product_id_address;      /* Address of 18 byte product ID */
};

struct  direct_parms
{
        unsigned int      dir_buf_size;             /* size of direct buffers */
        unsigned int      dir_pool_blocks;          /* length of buffers in segments */
        void              *dir_pool_address;        /* location of buffer pool */
        void              (*adpt_chk_exit);         /* adapter error interrupt hndlr */
        void              (*netw_status_exit);      /* network status err int. hndlr */
        void              (*pc_error_exit);         /* OS or PC hdwr err int hndlr */
        void              *work_addr;               /* adapter work area */
        unsigned int      work_len_req;             /* requested work area size */
        unsigned int      work_len_act;             /* required work area size */
};

struct  dlc_parms
{
        unsigned char     dlc_max_sap;              /* maximum number of SAPs */
        unsigned char     dlc_max_stations;         /* maximum number of stations */
        unsigned char     dlc_max_gsap;             /* maximum number of group SAPS */
        unsigned char     dlc_max_gmem;             /* maximum group members per SAP */
        unsigned char     dlc_t1_tick_one;          /* dlc timer t1 interval */
                                                    /* group 1 */
        unsigned char     dlc_t2_tick_one;          /* dlc timer t2 int, group 1 */
        unsigned char   dlc_ti_tick_one;        /* dlc timer ti int, group 1 */
        unsigned char   dlc_t1_tick_two;        /* dlc timer t1 int, group 2 */
        unsigned char   dlc_t2_tick_two;        /* dlc timer t2 int, group 2 */
        unsigned char   dlc_ti_tick_two;        /* dlc timer ti int, group 2 */
};

struct ncb_parms
{
        char            work_area1[4];
        unsigned char   ncb_timer_t1;           /* response timer value */
        unsigned char   ncb_timer_t2;           /* acknowledgment timer value */
        unsigned char   ncb_timer_ti;           /* inactivity timer */
        unsigned char   ncb_maxout;             /* transmit window size */
        unsigned char   ncb_maxin;              /* receive window size */
        unsigned char   ncb_maxout_incr;        /* dynamic window increment value
*/
        unsigned char   ncb_max_retry;          /* N2 value */
        char            work_area2[4];
        unsigned char   ncb_access_pri;         /* ring access priority */
        unsigned char   ncb_stations;           /* maximum netbios link stations */
        char            work_area3[19];
        unsigned char   ncb_max_names;          /* maximum entries in name table */
        unsigned char   ncb_max;                /* maximum outstanding NCBs */
        unsigned char   ncb_max_sessions;       /* maximum number of sessions */
        char            work_area4[2];
        unsigned char   ncb_options;            /* various bit options */
        unsigned int    ncb_pool_length;        /* length of ncb buffer pool */
        void            *ncb_pool_address;      /* start of ncb buffer pool */
        unsigned char   ncb_transmit_timeout;   /* time to wait for one query */
        unsigned char   ncb_transmit_count;     /* max times to transmit queries */
};


struct  dlc_open_sap_parms
{
        unsigned int    station_id;             /* SAP station ID */
        unsigned int    user_stat_value;        /* User value passed back on */
                                                /* DLC status */
        unsigned char   timer_t1;               /* T1 response timer value */
        unsigned char   timer_t2;               /* T2 ack timer value */
        unsigned char   timer_ti;               /* ti inactivity timer value */
        unsigned char   maxout;                 /* Maximum xmits w/o receive ack */
        unsigned char   maxin;                  /* Maximum rcvs w/o transmit ack */
        unsigned char   maxout_incr;            /* dyn window increment value */
        unsigned char   max_retry_cnt;          /* N2 value */
        unsigned char   max_members;            /* Maximum SAPs for a group SAP */
        unsigned int    max_i_field;            /* Max rcved information field */
        unsigned char   sap_value;              /* SAP value to be assigned */
        unsigned char   options_priority;       /* Sap options and ring */
                                                /* access priority */
        unsigned char   station_count;          /* Number of link access */
                                                /* stations to reserve */
        char            reserved1[2];
        unsigned char   group_count;            /* Length of data in group_list */
        void            *group_list;            /* Far pointer to address of */
                                                /* group SAP values */
        void            *dlc_status_exit;       /* Function pointer to status */
                                                /* change interrupt routine */
        unsigned int    dlc_buf_size;           /* Size of each dlc buffer */
        unsigned int    dlc_pool_len;           /* Size of entire dlc buf pool */
        void            *dlc_pool_addr;         /* Far pointer to dlc buf pool */
};

struct  buffer_parms
{
        unsigned int     station_id;              /* SAP station id */
        unsigned int     buffer_left;             /* number of buffs left in pool */
        unsigned char    buffer_get;              /* number of buffers to get */
        char             reserved1[3];
        void             *first_buffer;           /* addr of first buff obtained */
};

struct transmit_parms
{
        unsigned int     station_id;              /* station sending data */
        unsigned char    transmit_fs;             /* returned FS field */
        unsigned char    rsap;                    /* remote SAP value */
        void             *xmit_queue_one;         /* address of the 1st xmit queue */
        void             *xmit_queue_two;         /* address of the 2nd xmit queue */
        unsigned int     buffer_len_one;          /* length of 1st xmit buffer */
        unsigned int     buffer_len_two;          /* length of 2nd xmit buffer */
        void             *buffer_one;             /* address of 1st buffer */
        void             *buffer_two;             /* address of 2nd buffer */
};

struct  receive_buffer_type
{
        unsigned int     data_length;
        char             *data;
        struct  receive_buffer_type      *next_buffer;
};

struct receive_parms
{
        unsigned int     station_id;              /* station receiving data */
        unsigned int     user_length;             /* length of user data in buffer */
        void             *received_data;          /* user exit for received data */
        void             *first_buffer;           /* pointer to first buffer */
        unsigned char    options;                 /* receive options */
};

struct dlc_open_station_parms
{
        unsigned int     sap_station_id;          /* SAP station id */
        unsigned int     link_station_id;         /* link station id (0xnnss) */
        unsigned char    timer_t1;                /* response timer value */
        unsigned char    timer_t2;                /* acknowledgment timer value */
        unsigned char    timer_ti;                /* inactivity timer value */
        unsigned char    maxout;                  /* max xmits w/o an ack */
        unsigned char    maxin;                   /* max receives w/o an xmit ack */
        unsigned char    maxout_incr;             /* dynamic wind increment value */
        unsigned char    max_retry_cnt;           /* N2 value */
        unsigned char    rsap_value;              /* remote SAP value */
        unsigned int     max_i_field;             /* max received info field */
        unsigned char    access_priority;         /* ring access priority */
        char             reserved1;
        void             *destination;            /* pointer to remote address */
};

struct  dlc_connect_station_parms
{
        unsigned int     station_id;              /* link stat ID to be connected */
        char             reserved[2];
        void             *routing_addr;           /* pnter to 18 bytes of rte info */
};
```

```
#include        <string.h>
#include        "dlc.h"

/*******************************************************************
*       init_ccb - clear and initialize command control block
*
*       Parameters:
*               ccb (in/out) - command control block to be cleared
*
*       Notes:
*               This code sets the network adapter number to 0 (primary)
*
*       History:
*               Original code by William H. Roetzheim, 1990
********************************************************************/

void    init_ccb(struct command_control_block *ccb)
{
        memset(ccb, 0, sizeof(struct command_control_block));
}
```

Code Box 4.2 `init_ccb()` function definition.

In addition, it will simplify the remainder of our examples if we define two functions at this point. **`init_ccb()`** (Code Box 4.2) initializes a CCB, which simply involves clearing the CCB. **`int_adapter()`** (Code Box 4.3) performs an interrupt 0x5C using the passed address of a CCB. **`int_adapter()`** is passed both the address of your **`ccb`** and a flag which is set to either **`WAIT`** or **`NO_WAIT`** based on whether or not you want the function to wait for command completion prior to returning control to your program. As with **`int_netbios()`**, **`int_adapter()`** sets a global variable **`net_error`** equal to the return value in the **`ccb`** on return. You should note that this value will only be valid if the **`WAIT`** option was selected.

4.2 Addressing While Using DLC

Recall from our discussion of NetBIOS addressing that every IBM token ring network adapter is identified by a unique 6 byte number (12 hexadecimal digits), which NetBIOS calls the permanent node name. These six bytes are assigned by the manufacturer and are permanently

```
#include        <dos.h>
#include        "dlc.h"

extern  int     net_error;

/*******************************************************************
*       int_adapter - interrupt adapter with command control block
*
*       Parameters:
*               ccb (in/out) - initialized command control block
*               wait (in) - flag set to WAIT or NO_WAIT
*
*       Global:
*               _ES - ES register
*               _BX - BX register
*               net_error - set to command return code
*
*       Note:
*               net_error will only be set to a valid value if the WAIT option
*               is used.  If the NO_WAIT option is used, the ccb.retcode must
*               be checked and net_error set by the calling program.
*
*       History:
*               Original code by William H. Roetzheim
**********************************************************************/

void    int_adapter(struct command_control_block *ccb, int wait)
{
        _ES = FP_SEG(ccb);
        _BX = FP_OFF(ccb);
        geninterrupt(0x5C);
        if (wait == WAIT)
        {
                while (ccb.retcode == 0xFF);
        }
        net_error = ccb.retcode;
}
```

Code Box 4.3 `int_adapter()` function definition.

burned into the adapter card's ROM, although it is possible to override this number at boot time with a locally assigned 6-byte number. NetBIOS simplified addressing by allowing us to register symbolic names over the network which could then be used for addressing instead of the 6-byte number. *DLC programming requires that you know your own 6 byte permanent node name **and** the 6 byte permanent node name of every adapter you wish to communicate with.* It is this address which is used over the LAN to identify which adapter should read a packet of information. You use the permanent node names either when you establish a connection (connection oriented service) or when you build your own LAN header (datagram-oriented service). In most environments, this information is simply read (by an operator) from the literature included with each adapter, then entered into a table for use by your software. When an adapter is changed or new users are added, the table is updated. It is also possible to implement a protocol to transmit this information over the network (as is done by NetBIOS), but this is beyond the scope of this book.

Within the adapter, DLC programming requires that you use one or more protocol engines called *service access points*, or SAPs. SAPs contain data link level protocol capabilities, including windows, timeouts, media access, etc. SAPs can be initialized to receive all frames (packets) addressed to this adapter, only Media Access Control (MAC) frames, or only non-MAC frames. MAC frames are used for direct communication (by-passing DLC) as covered in the next chapter. Non-MAC frames are DLC and NetBIOS frames.

When you open a SAP, a SAP number is returned to you for use during communications. Each adapter supports up to 255 different SAPs, although the actual maximum is set when the adapter is initialized and is normally closer to 2 user defined SAPs (plus the 3 just mentioned). Within DLC programming, SAPs are used for connectionless communication.

As with NetBIOS names, SAPs can be either unique (individual) or group. A unique SAP number is always even while a group SAP number is always odd.

If you are using DLC's connection-oriented protocols, you need to establish a connection between you and another adapter. This connection is called a *link access point*, and is assigned as a connection *over a SAP*. When you open the connection, the link access point number is provided to you by the adapter for use during subsequent communications. Each adapter can support up to 255 simultaneous link access points (connections), although the actual number supported is established when the adapter is initialized. These link access points can be distributed among your available SAPs however you chose. For example, you could have 255 link access points assigned to a single SAP, or you could have 1 link access point assigned to each of 255 SAPs.

The SAP number and, optionally, a link access point number within the SAP are combined and called the *Station ID*. The first byte of this two byte number is the SAP number, the second byte is the link access point number. SAPs alone would be represented as a number of the form 0xSS00 where SS was replaced with the SAP number and second byte was set to 0. Link access points are represented as 0xSSLL, where SS is the SAP number and LL is the link access point number. Station IDs are used extensively in DLC programming.

When the token ring adapter is initialized, two SAPs are automatically opened:

- 0x00 is automatically opened and provides the capability to respond to remote nodes when no other SAPs have been opened. This SAP responds to only XID and Test Command frames (discussed later).

- 0xFF is a group SAP with all individual open SAPs as members. Sending frames to an adapters group SAP

(0xFF) will ensure that the frames are passed to *each* of the individual SAPs that are opened.

In addition, three station IDs are automatically established when the adapter is open:

- 0x0000 receives all frames not directed to other defined stations within this adapter.

- 0x0001 receives just MAC frames not directed to other defined stations within this adapter.

- 0x0002 receives just non-MAC frames not directed to other defined stations within this adapter.

Any of these three stations (0x0000 – 0x0002) can be used to transmit MAC *and* non-MAC frames. You should also keep in mind that the *ability* to receive frames via a SAP does not imply that data is received. Some type of receive command must be initiated prior to receipt of any data via a SAP.

```
#include        <string.h>
#include        <stdio.h>
#include        <dos.h>
#include        "dlc.h"

/*****************************************************************
*       init_adapter - test for adapter presence and initialize adapter
*       Returns:
*         0 for success, or
*               NO_ADAPTER              if adapter or dlc driver not installed
*               INIT_FAILURE    if error during initialization
*               OPEN_FAILURE    if error during adapter open
*       Notes:
*               If INIT_FAILURE or OPEN_FAILURE is returned, net_error can
*               be checked for the specific error return code.
*       History:
*               Original code by William H. Roetzheim
******************************************************************/
unsigned int init_adapter()
{
        unsigned        long                    int_vector;
        struct  command_control_block           ccb;
        struct  dir_initialize_parameters       init_parm;
        struct    dir_open_adapter_parameters   parm;
        struct  adapter_parms                   adapter;
        struct  direct_parms                    direct;
        struct  dlc_parms                       dlc;

        /***** start by testing for adapter installation *****/
        /* is interrupt vector initialized? */
        int_vector = (unsigned long) getvect(0x5C);
        if ((int_vector == 0x0000) || (int_vector == 0xF000))
        {
                /* no interrupt handler installed */
                return NO_ADAPTER;
        }
        init_ccb(&ccb);
        ccb.command = DIR_INTERRUPT;
        ccb.retcode = 0xF0;     /* invalid code */
        int_adapter(&ccb, WAIT);
        if (ccb.retcode == 0xF0) return NO_ADAPTER;
        /* adapter and driver are installed */

        /* now initialize adapter */
        init_ccb(&ccb);
        ccb.command = DIR_INITIALIZE;
        memset(&init_parm, 0, sizeof(struct dir_initialize_parameters));
        ccb.parameters = &init_parm;
        int_adapter(&ccb, WAIT);
        if (ccb.retcode != 0x00) return INIT_FAILURE;

        ***** continued next code box
```

Code Box 4.4 `init_adapter()` function definition.

```
        ***** Continued from previous code box

        /* now open the adapter */
        init_ccb(&ccb);
        ccb.command = DIR_OPEN_ADAPTER;
        ccb.parameters = &parm;
        parm.ap = &adapter;
        parm.dp = &direct;
        parm.dlcp = &dlc;
        parm.ncbp = NULL; /* use default for NetBIOS */
        memset(&adapter, 0, sizeof(struct adapter_parms));
        memset(&direct, 0, sizeof(struct direct_parms));
        memset(&dlc, 0, sizeof(struct dlc_parms));
        dlc.dlc_max_sap = DLC_MAX_SAP;
        dlc.dlc_max_stations = DLC_MAX_STATIONS;
        int_adapter(&ccb, WAIT);
        if (ccb.retcode != 0x00) return OPEN_FAILURE;
        return 0;
}
```

Code Box 4.5 `init_adapter()` function definition continued.

4.3 Adapter Initialization

Code Boxes 4.4 and 4.5 show the code for our function, **`init_adapter()`**, which tests for the adapters presence, ensures that the drivers are loaded, initializes the adapter, and opens the adapter. The function operates as follows:

- We begin by testing that the interrupt vectors at 0x5C are properly initialized. This will ensure that the drivers are installed.

- We then interrupt the adapter with a **nop** (no operation) type instruction after first placing an invalid return code into the **`retcode`** field of our **ccb** structure. If the returned **ccb** *still* has the same invalid return code, we know that the adapter is not installed (or responding) or that there is some other problem with the driver at 0x5C. If the return code *is* valid, we know that the adapter and driver appear to be installed and working properly.

- We then initialize the adapter. This command resets all adapter tables and buffers and forces the adapter to run the bring-up tests. You should note that this command will cause any outstanding **`ccbs`** to be lost. The **`DIR_INITIALIZE`** command is covered in more depth later in this section.

- Finally, we open the adapter with the **`DIR_OPEN_ADAPTER`** command. This command makes the adapter ready for normal network communication and sets adapter parameters and limitations. This command is also covered in more depth later in this section.

The structure fields discussed in the remainder of this section are defined in the **`paramblk.h`** include file which was listed earlier in this chapter.

4.3.1 DIR_INITIALIZE command

When using **`DIR_INITIALIZE`**, the **`parameters`** field of the **`ccb`** must be set equal to the address of a **`dir_initialize_parameters`** structure. The contents of **`dir_initialize_parameters`** can be set to zero and the adapter will use default values for all fields. The sample code sets all parameters to zero (default). The fields within the **`dir_initialize_parameters`** structure have the following meanings:

- **`bring_ups:`** This field should be initialized to zero. If the adapter detects an error during initialization, it will return a return code (in the CCBs **`retcode`** field) of 0x07 and will put an amplifying error description in this field for use by your application. These "bring-up errors" are listed at the end of this chapter.

- **sram_address:** This field contains the *segment* where the adapter shared RAM should be located. If the field is zero, the default values of 0xD800 for the primary adapter or 0xD400 for the secondary adapter. If you select a different value, you must ensure that the address is on an even boundary of the adapter shared RAM size (e.g., for an adapter with 16 Kbytes of shared RAM, the segment must be on a 16K memory boundary).

- **adptr_chk_exit:** This field contains a far pointer to a function which the adapter should call *whenever* it encounters an adapter hardware failure. This address is stored in the adapter for use until the adapter is reinitialized or the value is changed by your application. A value of zero means that no error handler is installed for this type of error.

- **netw_status_exit:** This field is identical to **adptr_chk_exit**, except that it is called whenever a network error is encountered.

- **pc_error_exit:** This field is identical to **adptr_chk_exit** (above), except that it is called whenever an operating system or PC hardware failure is encountered.

4.3.2 DIR_OPEN_ADAPTER Command

When using this command, the **parameters** field of the CCB structure must be initialized to point to a

dir_open_adapter_parameters structure. This structure is used for double indirect address, containing four far pointers to other structures:

1. **adapter_parms**: Contains adapter initialization parameters.

2. **direct_parms**: Contains direct interface initialization parameters. We discuss the adapter's direct interface in the next chapter.

3. **dlc_parms:** Contains DLC interface initialization parameters.

4. **ncb_parms:** Contains NetBIOS interface initialization parameters.

The first three pointers *must* be initialized to point to valid structures. Null pointers (zero) are not accepted. The last pointer, **ncb_parms** can be set to NULL to instruct the adapter to use default values for NetBIOS. The *contents* of the three structures *can* be initialized to zeros, which will instruct the adapter to use the default values for each class of initialization. The sample code sets most of these parameters to zero (default).

The **adapter_parms** structure contains the following fields which you can set, if desired:

- **open_error_code:** This field is used to return (to your application) the adapter error code upon opening. If the CCB **retcode** field is 0x07, you should check this field to determine the specific problem. The error codes are documented at the end of this chapter.

- **open_options:** This field contains bit fields which are used to turn on (or off) varous options. 1 indicates on, 0 indicates off. 0 is the default (and usual) value for each field. The only bit field you will probably be interested in is bit 9. If this bit is set to 1 and the **DIR_OPEN_ADAPTER** command is called, the fields in the **adapter_parms** structure will be set to their current values (i.e. to the values the adapter is currently using). Other bit field meanings are documented in IBM (1988).

- **node_address:** If the **NODE_ADDRESS** parameter was provided by the user when the adapter support software was loaded, this field is strictly used to return the current adapter node address to you. If the **NODE_ADDRESS** parameter was not specified, this field may be used to override the default (hardware) adapter **NODE_ADDRESS**. If the field is all zeros, the hardware address will be used and returned in this field.

- **group_address:** Sets the adapter's group address for receipt of group messages. Zero means the adapter belongs to no groups.

- **functional_addr:** Sets the adapter's functional address. Zero means the adapter belongs to no functional group. Functional addresses are discussed later.

- **number_rcv_buffers:** If this field is less than 2, the adapter will only successfully open if 8 receive buffers are available. If you wish to run with less than 8 buffers, you must set the required number of buffers in this field.

- **`rcv_buffer_len:`** Each receive buffer uses 8 bytes for overhead and stores data in the remaining space. The default receive buffer size (used if this field is zero) is 112 bytes, or 104 bytes of data. This field can change the buffer size to a value between 96 and 2048, although the new value must be a multiple of 8. Although received data which overflows a buffer will be chained, your application performance can be improved if you match the receive buffer size to your expected packet (message) size.

- **`dhb_buffer_length:`** The length of each transmit buffer. The data space within the buffer is equal to the buffer length minus 6 bytes overhead. The default (obtained using zero for this field) is 600 bytes (594 bytes of data). The maximum size available for the original token ring network adapters was 2048. If *all* adapter cards on the network are the newer models, the maximum size is 4464 at 4 Mbps or 17960 at 16 Mbps.

- **`data_hold_buffers:`** The number of transmit buffers on the card. This number should be two or less to protect the integretity of your data. If this field is zero, the default value of one buffer is used. Transmission efficiency will be improved somewhat by setting this value to two, but you will have less space available for receive buffers.

- **`open_lock:`** This field allows the adapter to be protected in a multi-tasking DOS environment. If this field is set to anything other than zero, the adapter is opened in a keyed mode where the key value is the number passed in this field. The adapter can then only be closed (or initialized) by a program using the proper number in this field.

- **product_id_address:** This field should always be initialized to all zeros.

The **direct_parms** structure contains the following fields which you can set, if desired:

- **dir_buf_size:** The size of buffers in the direct buffer pool. The number must be at least 80 bytes and must be a multiple of 16. The default (zero) value is 160.

- **dir_pool_blocks:** This parameter is only used if the **dir_pool_address** field is nonzero. This field indicates the number of 16 byte blocks assigned as the direct station buffer pool, with a default of 256 (4096 bytes).

- **dir_pool_address:** A far pointer to the address where the adapter should build a buffer pool. If this field is zero, the application program must build its own buffer pool using **BUFFER_FREE** and **BUFFER_GET**.

- **adpt_chk_exit:** This field is identical to the same field described under **DIR_INITIALIZE**.

- **netw_status_exit:** This field is identical to the same field described under **DIR_INITIALIZE**.

- **pc_error_exit:** This field is identical to the same field described under **DIR_INITIALIZE**.

- **work_addr:** The adapter work area can be internal to the adapter or external to the adapter (in your application memory space). The amount of space required is 48 plus (36 * **DLC_MAX_SAPS**) plus (6 * **DLC_MAX_STATION**). If you chose to use your application memory as the adapter work space, this field should point to the memory location to be used.

- **work_len_req:** If this field is zero, the adapter's internal memory will be used. If you chose to identify the adapter work area, this field points to the buffer size of the space located at **work_addr**. The space must be at least as large as the size identified.

- **work_len_act:** This field contains the actual work area space which will be used by the adapter. If this number is greater than **work_len_req**, the open fails and a return code of 0x12 is returned.

The **dlc_parms** structure contains the following fields which you can set, if desired:

- **dlc_max_sap:** The maximum number of simultaneously opened SAPs. If NetBIOS is installed, it uses a SAP which is not counted in this total. The default is two, the maximum is 126.

- **dlc_max_stations:** The maximum number of simultaneously opened link stations. The default is 6, the maximum is 255.

- **`dlc_max_gsap:`** The maximum number of simultaneously open group SAPs. The default is zero, the maximum is 126.

- **`dlc_max_gmem:`** The maximum number of SAPs that can be assigned to any given group. The default is zero, the maximum is 127.

- **`dlc_t1_tick_one:`** The number of 40-millisecond intervals between timer ticks for the T1 timer. The T1 timer is the response timer. The default is 5 (200 milliseconds).

- **`dlc_t2_tick_one:`** The number of 40-millisecond intervals between timer T2 ticks. The T2 timer is the receiver acknowledgment timer. The default is one (40 milliseconds).

- **`dlc_ti_tick_one:`** The number of 40 millisecond intervals between timer Ti ticks. The Ti timer is the inactivity timer. The default is 25 (1 second).

- **`dlc_t1_tick_two:`** Each of the three timers has a long value associated with it. The default for this field is 25 40-millisecond ticks (1 second).

- **`dlc_t2_tick_two:`** The default for this field is 10 40-millisecond ticks (400 milliseconds).

- **`dlc_ti_tick_two:`** The default for this field is 125 40-millisecond ticks (5 seconds).

After the adapter is successfully opened, you can communicate using DLC in either a connectionless or connection-oriented mode. Recall that connectionless communication uses the SAPs while connection oriented communication uses link stations. We will begin by discussing DLC communication using connectionless protocols.

4.4 Connectionless Communication Using DLC

In this section we will describe how to use the token ring adapter to perform datagram communication at the DLC level. The steps involved are as follows:

1. Open a service access point. SAPs were discussed fully in section 4.2. This SAP is a protocol engine which can be used for DLC communication. Both the sending adapter *and* the receiving adapter must open a SAP on their token ring adapter cards. The SAPs can be (and often are) the same number for both the sending adapter and the receiving adapter. If your application will be the only application running on both the sending and receiving computers, you can use one of the standard SAPs if desired (see Section 4.2). In general, it is best to designate a specific SAP number to support your application over the network. The process of opening a SAP is covered in Section 4.4.1.

2. Build a LAN header. As shown in Figure 3.1, the LAN header is one of the most outermost layers of a frame on the network. This header is used to identify the destination adapter. When communicating at the datagram level, your application must build the LAN header. The process of building the LAN header is covered in Section 4.4.2.

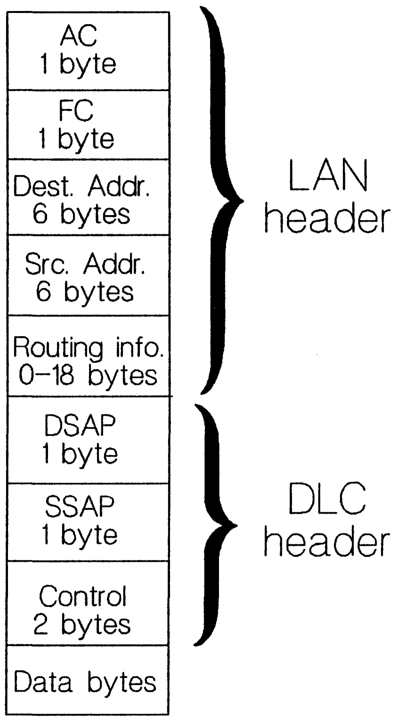


Fig. 3.1 Frame header layering.

3. Transmit datagrams using DLC. Data packets are then transmitted over the network. Your applications are responsible for ordering of data between packets, acknowledging packets, and so on. The token ring network makes a "best effort" to deliver the packet in an error free condition. In addition, the circular nature of the token ring allow your adapter to determine if the packet was successfully removed from the network by an adapter (hopefully, the destination adapter). This is possible because the receiving adapter sets a specified bit in

the data frame prior to sending it on its way (eventually back to you). This process is covered in Section 4.4.3.

4. Receive datagrams from the network. We will see that the receive processing is identical whether the received data is datagram or connection oriented. To avoid losing new packets while processing an adapter, we will implement the receive processing as an interrupt-driven background process. This approach then also serves as an example for programmers wishing to implement other functions as background processes. This process is covered in Section 4.4.4.

5. Received datagrams are temporarily stored in buffers under the control of the token ring adapter. We will want to remove the data from these buffers as quickly as possible, the tell the adapter that the buffer is free. This process is also covered in Section 4.4.4.

4.4.1 Opening a SAP

We normally open one SAP for each application using the token ring network. The SAP is can be opened with unique values for timers, window sizes (windows in the sense of protocol based acknowledgments), and so on. When you want to communicate with another application, you address the adapter card the process resides on *and* the SAP within the card.

In general, opening a SAP is a relatively simple matter (see Code Box 4.6). This is because zero for any of the parameters automatically causes the adapter to open the SAP with a predefined default value. If you look at the fields defined in the **dlc_open_sap_parms** structure, you will find that most of them are identical to fields we saw when initializing or opening the adapter itself

```
#include        <string.h>
#include        "dlc.h"

extern  int             net_error;

/*****************************************************************
*       open_sap() - open a SAP on this adapter
*
*       Parameters:
*               sap (in) - SAP number to use
*               resv_link (in) - number of link access stations to reserve
*
*       Global:
*               net_error is set by int_adapter().
*
*       Returns:
*               The opened SAP station id on success, 0 for failure
*
*       Notes:
*               This routine assumes that your application only opens one SAP
*               at a time.  If you open multiple SAPs, you must modify the code
*               to use malloc() to allocate the buffer space used by the SAP
*               (and free the memory when the SAP is closed).
*
*       History:
*               Original code by William H. Roetzheim
*******************************************************************/

unsigned int    open_sap(int sap, int resv_link)
{
        struct  dlc_open_sap_parms      parms;
        struct  command_control_block   ccb;
        static  char                    buffer[4096];   /* SAP buffer space */

        init_ccb(&ccb);
        ccb.command = DLC_OPEN_SAP;
        ccb.parameters = &parms;

        memset(&parms, 0, sizeof(struct dlc_open_sap_parms));
        parms.sap_value = sap;
        parms.station_count = resv_link;
        parms.dlc_pool_addr = buffer;
        int_adapter(&ccb, WAIT);
        if (net_error == 0) return parms.station_id;
        else return 0;
}
```

Code Box 4.6 `open_sap()` function definition.

(Section 4.3). When you open a SAP, the default is to use the values specified for the adapter when it was initialized or opened. However, you are allowed to override these values if they should be different for one specific SAP (perhaps to test two different protocols, for example).

The code shown assumes that this application will open only one SAP at a time, so the SAP local buffer pool is initialized to a static storage area within this function. If you may open multiple SAPs simultaneously, this would obviously not work (multiple SAPS would be trying to use the same storage space). In this case, you would need to assign each SAP its own storage space, probably using **`malloc()`** to obtain the memory. You must then keep track of the various memory pointers so that they can be freed up when you are done.

After the SAP is successfully opened, a **`station_id`** is returned which can be used to identify this SAPs location for outgoing packets. Incoming packets will address this SAP by its SAP number, *not* its **`station_id`**.

4.4.2 Building the LAN Header

When communicating over the token ring network, DLC programming uses a non-MAC frame. Media access control frames are used for direct programming as discussed in the next chapter. The non-MAC frame, or packet, consists of three parts:

1. A LAN header used by the token ring adapter to route the frame to its intended adapter.

2. A DLC header used by the destination adapter to route the frame to the appropriate SAP (and link station when applicable).

3. The data itself.

The DLC header is always provided by the adapter itself. The data portion of the frame is always provided by your application. The LAN header is provided by your application if you are using datagram-oriented DLC communication or is provided by the adapter if you are using connection oriented communication. Luckily, the LAN header can normally be built once, and then used for *all* outgoing frames without change.

The LAN header consists of five fields:

1. A 1-byte access control (AC) bit field, which specifies things like priority.

2. A 1-byte frame control (FC) bit field, which specifies things like the type of frame.

3. A 6-byte destination address (the destination adapter number in hex).

4. A 6-byte source address (your address).

5. A 0 – 18 bytes routing information field specifying up to three intermediate gateway adapter addresses.

Creating the header is not as complicated as this might lead you to believe. The first two bytes are automatically filled in by the adapter, so you don't need to worry about them. The 6-byte destination address must be filled in by your application. The 6-byte source address is automatically filled in by your adapter, so you don't need to worry about this. Finally, the routing information field is required only if you will be sending data from one token ring network to another token ring network.

```
/******************************************************************
*       build_lan_header() - build LAN header in a buffer
*
*       Parameters:
*               destination (in) - six byte destination address
*               buffer (in/out) - address of buffer for LAN header
*
*       History:
*               Original code by William H. Roetzheim
**********************************************************************/

void    build_lan_header(char destination[6], char *buffer)
{
        int             i;

        memset(buffer,0,14);
        /* byte 0 = Access Control: supplied by adapter */
        /* byte 1 = Frame Control:  supplied by adapter */
        /* bytes 2-7 = destination address */
        memcpy(&buffer[2], destination, 6);
        /* bytes 8-13 = source address: supplied by adapter */
}
```

Code Box 4.7 `build_lan_header()` function definition.

Code Box 4.7 shows sample code to build an LAN header. If you wish to go through a gateway to another token ring network, you must add the routing information.

4.4.3 Transmitting Datagrams Using DLC

For connectionless (datagram) DLC service, the transmit routine is called **TRANSMIT_UI_FRAME**. This routine needs to know your **station_id** (assigned when you opened the SAP), the destination SAP number (which is normally the same as your local SAP number), and the destination address. In addition, a pointer to your data and an unsigned integer indicating the length of data in the buffer are required. The data length should be small enough to fit within one frame.

The **TRANSMIT_UI_FRAME** command takes *two* buffers (and buffer lengths) in its parameter table. The first buffer and length describes the LAN header. This buffer will normally be used for *all* outgoing transmissions to a given destination adapter. The *second*

```
#include        <string.h>
#include        "dlc.h"

/*******************************************************************
*       transmit_ui_frame() - transmit datagram at DLC level
*
*       Parameters:
*               station_id (in) - value returned from DLC_OPEN_SAP
*               sap (in) - SAP number used for communication
*               destination (in) - destination address
*               data_len (in) - length of data in bytes
*               data (in) data to transmit
*
*       Global:
*               net_error is set by int_adapter()
*
*       Returns:
*               0 for success, NOT_RECEIVED if the frame was not copied from
*               the ring, OVERLOAD if the destination was overloaded and
*               did not receive the frame, or else OTHER_ERROR
*
*       History:
*               Original code by William H. Roetzheim
*******************************************************************/
int             transmit_ui_frame(unsigned int station_id, unsigned int sap,
                        char destination[6], unsigned int data_len,
                        char *data)
{
        struct  command_control_block   ccb;
        struct  transmit_parms          parms;
        char                            lan_header[14];

        init_ccb(&ccb);
        ccb.command = TRANSMIT_UI_FRAME;
        ccb.parameters = &parms;
        memset(parms, 0, sizeof(struct transmit_parms);
        parms.station_id = station_id;
        parms.rsap = sap;
        build_lan_header(destination, lan_header);
        parms.buffer_len_one = 14;      /* buffer 1 MUST be the lan header */
        parms.buffer_one = lan_header;  /* The adapter will take buffer 1, */
        parms.buffer_len_two = data_len; /* add the DLC header, then the data */
        parms.buffer_two = data;        /* and transmit the entire frame */
        int_adapter(&ccb, WAIT);
        switch (parms.transmit_fs)
        {
                case 0xCC: return 0;
                case 0x00: return NOT_RECEIVED;
                case 0x88: return OVERLOAD;
        }
        return OTHER_ERROR;
}
```

Code Box 4.8 `transmit_ui_frame()` function definition.

buffer and length contains the actual data. Code Box 4.8 shows sample code allowing you to transmit a datagram frame.

After the data is successfully transmitted, the adapter can look at the frame control field to determine if the frame was successfully removed from the ring. The three possibilities are

1. The frame could have been received successful (at least as far as the hardware is concerned). The defined function then returns 0x00.

2. The frame could have been removed successfully from the token ring network (by the destination adapter) but never successfully picked up by the destination software. This often indicates that the destination adapter is temporarily too busy or that all buffers are temporarily filled. This routine then returns **OVERLOAD**. It would also be logical to modify the function so that when this condition occurs, the function is called again (recursively) to retransmit the data. You might include a static variable indicating the level of recursion to prevent infinite recursion (and an eventual crash).

3. The frame could have gone all the way around the network without ever being removed by the destination adapter. This normally indicates a hardware problem with the destination adapter (perhaps it is turned off) and is indicated by a return value of **NOT_RECEIVED**. You should *not* immediately try to recursively send the packet again if this error is returned (it probably won't do any good).

Note that when you are sending multiple packets, you normally only need to change the value for **data** and **data_len**. The routine, as written, will build a new LAN header for each outgoing data packet. As mentioned earlier, this is normally not necessary and you can modify the code to reflect this fact if performance is really critical.

4.4.4 Receiving DLC Data Packets

Under DLC, the identical receive function is used to receive *all* types of frames, including datagrams and connection oriented frames. The function **`receive_dlc()`** shown in Code Box 4.9 can be used to start the receive process running. This function takes as an argument a **`station_id`** as returned by **`open_sap()`** for datagram service or **`open_station()`** for connection-oriented service (as we will see in Section 4.5). This function only needs to be called *once* for each SAP or link access station. It then sets up the interrupt vectors so that incoming frames will be received continuously until the SAP/station is closed or an error occurs. The interrupt processing routine that it sets up is called **`receive_process()`**. This is the routine which actually processes incoming frames upon demand. Because this interrupts the adapter with the **`NO_WAIT`** option, the *proper* return value is 0xFF, showing that the command is running after the return.

Code Box 4.10 and Code Box 4.11 show our interrupt routine designed to handle incoming frames. This function is setup by **`receive_dlc()`** and should never be directly called by your application. It is called (via interrupt) by the adapter itself when a frame is available. When the function is called, registers _ES and _BX point to the first receive buffer, while registers _DS and _SI point to the Command Control Block used during the call.

Our goal is to move the received data from the adapter work area to user buffer space as quickly as possible so that the adapter is free to receive additional buffers. We have created a structure, called **`receive_buffer_type`** to identify received data available for user processing. This structure contains a pointer to the received data (stripped of header information) and the length of data at the address.

```
#include        <string.h>
#include        <stdlib.h>
#include        "dlc.h"

extern  int     net_error;

/*******************************************************************
*       receive_dlc() - receive dlc frame data
*
*       Parameters:
*               station_id (in) - station id to receive frames from
*
*       Global:
*               net_error is set by int_adapter().
*
*       Returns:
*               0 for success, net_error for error
*
*       Notes:
*               This function runs continuously (in background) to receive
*               data from the specified station_id.  Data is placed in
*               receive_buffer (a linked list of received buffers).  The
*               command terminates when the SAP or Link Station is closed.
*
*       History:
*               Original code by William H. Roetzheim
*******************************************************************/

int             receive_dlc(unsigned int station_id)
{
        struct  receive_parms           parms;
        struct  command_control_block   ccb;
        void                            receive_process();

        init_ccb(&ccb);
        ccb.command = RECEIVE;
        ccb.parameters = &parms;

        memset(&parms, 0, sizeof(struct receive_parms));
        parms.station_id = station_id;
        parms.received_data = (*receive_process);
        int_adapter(&ccb, NO_WAIT);
        if (net_error == 0xFF) return 0;
        else return net_error;
}
```

Code Box 4.9 **receive_dlc()** function definition.

```
#include        <dos.h>
#include        <stdlib.h>
#include        <string.h>
#include        <stdio.h>
#include        "dlc.h"

extern  struct  receive_buffer_type     rb;

/*******************************************************************
*       receive_process() - interrupt function to receive buffers
*
*       Global:
*               rb - used to store incoming buffer
*               _ES, _BX - pseudo-variables used by Turbo C for these registers
*
*       Returns:
*               None - interrupt function
*
*       Notes:
*               Data buffers pointed to by rb should be freed when you
*               are done with them (use free() ).
*
*               This function is automatically "called" by the adapter
*               every time a frame is received.  It is NOT called by
*               your application.  It is initialized by receive_dlc(),
*               which IS called by your application.
*
*       History:
*               Original code by William H. Roetzheim
*********************************************************************/

void    interrupt       receive_process()
{
        char            *first_buffer;
        int             start;
        int             length;
        unsigned int    station_id;
        int             receive_index;
        char            *working_buffer;
        struct  receive_buffer_type     *rb_ptr;
        struct  command_control_block   *ccb;
        struct  receive_parms           *parms;

        /* find first available receive buffer */
        /* NOTE:  if multiple applications might use the same receive */
        /*      buffer linked list, the next three lines of code */
        /*      should be protected by a semaphore or executed with */
        /*      interrupts turned off */
        rb_ptr = &rb;
        while (rb_ptr->next_buffer != NULL) rb_ptr = rb_ptr->next_buffer;
        rb_ptr->next_buffer = malloc(sizeof(struct receive_buffer_type));
        . . . Continued in next code box
```

Code Box 4.10 `receive_process()` function definition.

```
        /* ... continued */
        first_buffer = MK_FP(_ES, _BX);   /* init first buffer to _ES and _BX */
        ccb = MK_FP(_DS, _SI);            /* points to CCB */
        parms = ccb->parameters;
        station_id = parms->station_id;

        /* find length of DATA portion of buffers */
        rb_ptr->data_length = 0;
        for (working_buffer = first_buffer; working_buffer != NULL;
                working_buffer = *(char **) (&working_buffer[0]))
        {
                rb_ptr->data_length += *(unsigned int *)(&working_buffer[6]);
        }
        rb_ptr->data = malloc(rb_ptr->data_length);

        /* read data from buffers into our application buffer */
        receive_index = 0;
        for (working_buffer = first_buffer; working_buffer != NULL;
                working_buffer = *(char **) (&working_buffer[0]))
        {
                start = *(unsigned int *) (&working_buffer[8]) +
                        *(unsigned int *) (&working_buffer[10]);
                length = *(unsigned int *) (&working_buffer[6]);
                memcpy(&rb_ptr->data[receive_index],    &working_buffer[start],
                        length);
                receive_index += length;
        }

        buffer_free(station_id, first_buffer);
}
```

Code Box 4.11 **`receive_process()`** function definition.

Because we may receive more than one frame before the user processes the data, we will use a linked list of these structures. The last structure in the list is available for our use. The other members of the chain contain data received previously which has not been processed by the user.

We begin by moving down the linked list to determine the last structure in the list, which is the one we will use. The final element is identified by the **`next_buffer`** field being NULL. We then "claim" this structure for our use by using **`malloc()`** to add a new structure to the list. If you are running in a multitasking environment, this process should be protected by semaphores (or in some other way) to avoid contention for the buffer structure.

We then process all buffers containing the frame information. Received frames are stored in the adapters frame pool as a linked list of buffers. The exact format of each buffer will vary based on whether this is the first buffer in the list or one of the subsequent buffers, but the important field are common to *all* buffers as follows:

- The first four bytes are a pointer to the next buffer, or NULL for the final buffer.
- Bytes 6 and 7 contain the length of the *data* portion of the buffer.
- Bytes 8 and 9 contain the offset to an area of the buffer called the user space.
- Bytes 10 and 11 contain the length of the user space.

The data portion of the buffer begins at the end of the user space, or bytes 8 and 9 (the start of the user space) plus bytes 10 and 11 (the length of the user space).

We traverse the buffer list initially just to determine the total length of user data in the buffers. This number is then used to allocate memory for storage of the user data. We then traverse the buffer list again, copying the data from the buffers to the allocated memory block. Finally, we tell the adapter that its buffers are now available using the **`buffer_free()`** command, shown in Code Box 4.12.

4.5 Connection Oriented Communication Using DLC

Datagram communication involves communication between two applications using the SAPs on their token ring adapters. By their definition, datagram architectures provide no guarantees

- That the data was successfully received.

```
#include        <string.h>
#include        "dlc.h"

extern  unsigned        int     net_error;

/*****************************************************************
*       buffer_free() - free a buffer from the dlc buffer pool
*
*       Parameters:
*               station_id (in) - station id of opened SAP to use
*               buffer (in) - pointer to buffer to be freed
*
*       Global
*               net_error is set by int_adapter()

*       Returns:
*               0 for success, net_error for failure
*
*       Notes:
*               The buffer size is determined when the SAP is opened.  The
*               default is 160 bytes.
*
*       History:
*               Original code by William H. Roetzheim
*******************************************************************/

unsigned int    buffer_free(unsigned int station_id, void *buffer)
{
        struct  command_control_block ccb;
        struct  buffer_parms parms;

        init_ccb(&ccb);
        ccb.command = BUFFER_FREE;
        ccb.parameters = &parms;

        memset(&parms, 0, sizeof(struct buffer_parms));
        parms.station_id = station_id;
        parms.first_buffer = buffer;
        int_adapter(&ccb, WAIT);
        return net_error;
}
```

Code Box 4.12 **buffer_free()** function definition.

- That groups of datagrams will be delivered in the order they were transmitted.
- That the received data is not corrupted.
- That flow control will prevent buffer overflow.

The architecture of the token ring network improves the situation somewhat for all transmissions, including datagrams. Specifically,

- The circular nature of the network allows you to confirm that a frame was successfully removed from the network.
- The token passing protocol inherent in the token ring network assures that datagrams will be delivered in order (although there may be some gaps if frames were not successfully delivered).
- Receipt of corrupted data can be ignored for all but the most critical applications because of the inherent reliability of the network, a media access control approach which eliminates collisions, and built-in hardware-level error detection. The exceptions typically involve transaction-oriented systems where an error would be very damaging (e.g., bank money transfers), but these applications will almost certainly build in application-level error detection features no matter how reliable the underlying network was.

In spite of these advantages, it is often easier to use a connection-oriented approach to communications. Connections automatically provide you with complete assurance that the data was successfully received by the destination application. This is accomplished through the use of acknowledgements and automatic retransmissions as necessary. In addition, the protocols used for connection oriented service automatically provide you with flow control between your application and the destination application.

Under DLC, connection oriented communication is accomplished via link access stations. As discussed in Section 4.2, link access stations are an additional level of protocol which runs on top of the existing datagram service available through a SAP. In this section, we will

- Describe how to open a link access station.
- Describe how to open a connection over a link access station.
- Describe how to transmit data over a connection.

Note that we do not address receipt of data over a connection. This is because the function previously defined to receive datagram packets will operate, without change, for both receipt of datagram frames and connection-oriented frames. In fact, the receive buffers are identical for both forms of transmission.

4.5.1 Opening a Link Access Station

To perform connection oriented communication using DLC, you must first open a link access station. To open a DLC link access station, you use the **open_station()** function defined in Code Box 4.13. This function takes your local station_id (as returned from your call to **open_sap()**, the SAP number you wish to communicate with at your destination, and your destination address. Remember, the destination address is 12 hexadecimal digits assigned by the manufacturer or set

```
#include        <string.h>
#include        "dlc.h"

extern   int     net_error;

/*******************************************************************
*       open_station() - open a link access station on this sap
*
*       Parameters:
*               station_id (in) - value returned from DLC_OPEN_SAP
*               sap (in) - SAP number used for communication
*               destination (in) - destination address
*
*       Global:
*               net_error is set by int_adapter().
*
*       Returns:
*               The opened link station_id for success, 0 for failure
*
*       History:
*               Original code by William H. Roetzheim
*******************************************************************/

unsigned int    open_station(unsigned int station_id, unsigned int sap,
                                        char destination[6])
{
        struct  dlc_open_station_parms   parms;
        struct  command_control_block    ccb;

        init_ccb(&ccb);
        ccb.command = DLC_OPEN_STATION;
        ccb.parameters = &parms;

        memset(&parms, 0, sizeof(struct dlc_open_station_parms));
        parms.sap_station_id = station_id;
        parms.rsap_value = sap;
        parms.destination = destination;
        int_adapter(&ccb, WAIT);
        if (net_error == 0) return parms.link_station_id;
        else return 0;
}
```

Code Box 4.13 `open_station()` function definition.

when the adapter is initialized, all stored in 6 bytes. Other parameters available allow you to tailor the protocol parameters to your application by setting values for timers and windows. One field you may decide to use is **access_priority**. This field lets you assign different priorities to various applications using the token ring network. Higher-priority applications will be able to transmit data prior to low-priority applications.

```
#include        <string.h>
#include        "dlc.h"

extern  int     net_error;

/*******************************************************************
*       connect_station() - connect to a link access station
*
*       Parameters:
*               station_id (in) - value returned from DLC_OPEN_SAP
*
*       Global:
*               net_error is set by int_adapter().
*
*       Returns:
*               0 for success, net_error for failure
*
*       History:
*               Original code by William H. Roetzheim
**********************************************************************/

unsigned int    connect_station(unsigned int station_id)
{
        struct  dlc_connect_station_parms       parms;
        struct  command_control_block           ccb;

        init_ccb(&ccb);
        ccb.command = DLC_CONNECT_STATION;
        ccb.parameters = &parms;

        memset(&parms, 0, sizeof(struct dlc_connect_station_parms));
        parms.sap_station_id = station_id;
        int_adapter(&ccb, WAIT);
        return net_error;
}
```

Code Box 4.14 **connect_station()** function definition.

4.5.2 Establishing a Connection

When you open a link access station, all processing is local (i.e., within your adapter). You must still establish the connection between your application and the destination application prior to communicating. This process involves physically establishing connectivity over the network and allocating buffer space at both ends of the connection. To establish a connection, you use the **`connect_station()`** function shown in Code Box 4.14.

4.5.3 Transmitting a Connection-Oriented Data Packet

After a connection is established, the **`xmit_i_frame()`** function can be used to transmit data over the connection. Unlike the **`transmit_ui_frame()`** function, both the DLC header *and* the LAN header are automatically created by the adapter. You simply pass the local station_id, the remote SAP number, a pointer to the data to send (data only), and the length of the data. This function is defined in Code Box 4.15.

4.6 Adapter Shutdown

When you are done communicating, you should close your SAP. Closing the SAP automatically closes all link access stations associated with the SAP. To close a SAP, you can use the **`close_sap()`** function shown in Code Box 4.16.

4.7 Summary of DLC Commands

The following table presents a summary of all DLC commands. The columns have the following meanings:

```
#include        <string.h>
#include        "dlc.h"

extern      int    net_error;
/*******************************************************************
*       xmit_i_frame() - transmit datagram with connection service
*
*       Parameters:
*               station_id (in) - value returned from DLC_OPEN_SAP
*               sap (in) - SAP number
*               data_len (in) - length of data in bytes
*               data (in) data to transmit
*
*       Global:
*               net_error is set by int_adapter()
*
*       Returns:
*               0 for success, net_error for failure
*
*       History:
*               Original code by William H. Roetzheim
********************************************************************/

int             xmit_i_frame(unsigned int station_id, unsigned int sap,
                                unsigned int data_len, char *data)
{
        struct  command_control_block    ccb;
        struct  transmit_parms           parms;

        init_ccb(&ccb);
        ccb.command = TRANSMIT_I_FRAME;
        ccb.parameters = &parms;

        parms.station_id = station_id;
        parms.rsap = sap;
        parms.buffer_len_one = data_len;
        parms.buffer_one = data;
        int_adapter(&ccb, WAIT);
        return net_error;
}
```

Code Box 4.15 **`xmit_i_frame()`** function definition.

```
#include        <string.h>
#include        "dlc.h"

extern  int     net_error;

/*******************************************************************
*       close_sap() - close a SAP on this adapter
*
*       Parameters:
*               sap (in) - station id to close (returned from open_sap())
*
*       Returns:
*               0 for success, net_error for failure
*
*       Notes:
*               This routine assumes that your application only opened one SAP
*               at a time.  If you opened multiple SAPs, you must modify the code

*               to use free the buffer memory space allocated by open_sap().
*
*       History:
*               Original code by William H. Roetzheim
*********************************************************************/

unsigned int    close_sap(unsigned int station_id)
{
        struct  command_control_block   ccb;

        init_ccb(&ccb);
        ccb.command = DLC_CLOSE_SAP;
        ccb.parameters = (void *) (unsigned long) station_id;
        int_adapter(&ccb, WAIT);
        return net_error;
}
```

Code Box 4.16 **close_sap()** function definition.

1. **Command** The command name. These names are defined in **dlc.h**. These are the values to use for the Command Control Block's **command** field prior to calling the adapter for processing.

2. **Inputs** The fields within the Command Control Block (and in associated parameter structures) which are used as input.

3. **Outputs** The fields within the Command Control Block and associated parameter structures which are modified by the command during processing.

4. **Summary** A brief description of the command function.

Command	Inputs	Outputs	Summary
BUFFER_FREE (0x27)	station_id (*) first_buffer (*)	buffer_left	Returns one or more buffers to the SAP's buffer pool.
BUFFER_GET (0x26)	station_id (*) buffer_get	buffer_left first_buffer	Gets one or more buffers from the SAP's buffer pool.
DLC_CLOSE_SAP (0x16)	station_id (*)		Closes a SAP.
DLC_CLOSE_STATION (0x1A)	station_id (*)		Closes a link access station.
DLC_CONNECT_STATION (0x1B)	station_id (*) routing_addr		Places both local and remote stations in data transfer state.
DLC_FLOW_CONTROL (0x1D)	station_id (*) flow_control (*)		Sets a SAP or link access station busy status.
DLC_MODIFY (0x1C)	station_id (*) timer_t1 timer_t2 timer_ti maxout maxin maxout_incr max_retry_cnt access_priority group_count group_list		Modify working parameters for an open SAP or link access station.

DLC_OPEN_SAP (0x15)	timer_t1 timer_t2 timer_ti maxout maxin maxout_incr max_retry_cnt max_members max_i_field sap_value (*) options_priority group_count group_list dlc_status_exit dlc_buf_size dlc_pool_len dlc_pool_addr (*)	station_id	Activate a SAP and reserve SAP link access stations. A buffer pool is also assigned for the SAP.
DLC_OPEN_STATION (0x19)	sap_station_id (*) timer_t1 timer_t2 timer_ti maxout maxin maxout_incr max_retry_cnt rsap_value (*) max_i_field access_priority destination (*)	link_station_id	Allocates local resources for a link access station in preparation for establishing a connection.
DLC_RESET (0x14)	station_id (*)		Reset a SAP and all associated link access stations. If a station_id of 0x0000 is used, all SAPs and link access stations are reset.
DLC_SET_THRESHOL D (0x33)	station_id (*) buffer_threshold (*) alert_semaphore (*)		This command applies to OS/2 only and is not available for DOS.
DLC_STATISTICS (0x1E)	sap_station_id (*) log_buf_length (*) log_buf_addr (*) options	log_act_length	Read (and optionally reset) a DLC log.
DLC_CLOSE_ADAPTE R (0x04)	lock_code (if opened with a lock code)		Close an adapter and terminate network communication.

DIR_CLOSE_DIRECT (0x34)			This command applies to OS/2 only and is not available for DOS.
DIR_DEFINE_MIF_ENVIRONMENT (0x2B)	ncb_input (*) ncb_open (*) ncb_close (*)	ncb_enable	This command allows a NetBIOS emulator to operate with the adapter support software.
DIR_INITIALIZE (0x20)	sram_address adptr_chk_exit netw_status_exit pc_error_exit	bring_ups sram_address	This command initializes the adapter, resets all adapter tables and buffers, and performs bring-up tests.
DIR_INTERRUPT (0x00)			This command forces an adapter interrupt but performs no action (a NOP).
DIR_MODIFY_OPEN_PARMS (0x01)	dir_buf_size dir_pool_blocks dir_pool_address adpt_chk_exit netw_status_exit pc_error_exit open_options		This command allows you to modify many default values set when an adapter was opened.
DIR_OPEN_ADAPTER (0x03)	adapter_parms (*) direct_parms (*) dlc_parms (*) ncb_parms (NOTE: The contents of the four structures are described fully in Section 4.3.2)	Various - see section 4.3.2	This command opens the adapter and reinitializes all buffers and tables. Parameter structures set defaults for use within the adapter.
DIR_RESTORE_OPEN_PARAMETERS (0x02)			This command is used to restore adapter parameters set when the adapter was opened after they have been modified with DIR_MODIFY_OPEN_PARMS.
DIR_SET_EXCEPTION_FLAGS (0x2D)			This command applies to OS/2 only.

DIR_SET_FUNCTIONAL_ADDRESS (0x07)	bits_to_change (*)		This command allows you to temporarily modify the adapter's internal address used for receiving frames from the token ring network.
DIR_SET_GROUP_ADDRESS (0x06)	bits_to_change (*)		This command allows you to temporarily modify the adapter's internal group address.
DIR_SET_USER_APPENDAGE (0x2D)	adpt_chk_exit netw_status_exit pc_error_exit		This command allows you to set (or modify) the adapters interrupt service functions you wish called on certain conditions.
DIR_STATUS (0x21)		encoded_addr node_address group_address functional_addr max_sap open_sap max_station open_station avail_station adapter_config microcode_level adapter_parms_addr adapter_mac_addr tick_cntr_addr last_ntwk_status	This command returns status information about the adapter.
DIR_TIMER_CANCEL (0x23)	ccb_pointer (*)		This command cancels a timer set with DIR_TIMER_SET.
DIR_TIMER_CANCEL_GROUP (0x2C)	post (*)		This command cancels all timers whose post address (from the CCB) is equal to post.
DIR_TIMER_SET (0x22)	time (*)		This command starts a programmable timer set to interrupt your application at time * .5 seconds. Upon expiration, the post routine (from the CCB block) will be executed.

PDT_TRACE_ON (0x24)	table_length (*)	current_off start_tick_0 stop_tick_0 start_tick_1 stop_tick_1	This command logs all interrupts for adapter traffic.
PDT_TRACE_OFF (0x25)			This command stops loging of all interrupts for adapter traffic.
PURGE_RESOURCES (0x36)			This command applies to OS/2 only.
READ (0x31)			This command applies to OS/2 only.
READ_CANCEL (0x32)			This command applies to OS/2 only.
RECEIVE (0x28)	station_id (*) user_length (*) received_data options	first_buffer	This command is used to receive all DLC data, whether connection oriented or datagram.
RECEIVE_CANCEL (0x29)	station_id (*)		This command cancels an outstanding receive command.
RECEIVE_MODIFY (0x2A)	station_id (*) user_length received_data subroutine (*)	first_buffer	This command receives specially formated data and places it into both the SAP buffer pool and a user buffer.
TRANSMIT_I_FRAME (0x0B)	station_id (*) rsap (*) xmit_queue_one xmit_queue_two buffer_len_one buffer_len_two buffer_one buffer_two	transmit_fs	This command transmits a frame of data over a connection.
TRANSMIT_TEST_CMD (0x11)	station_id (*) rsap (*) xmit_queue_one xmit_queue_two buffer_len_one buffer_len_two buffer_one buffer_two	transmit_fs	This command transmits a test command frame with the poll bit set.

TRANSMIT_UI_FRAME (0x0D)	station_id (*) rsap (*) xmit_queue_one xmit_queue_two buffer_len_one buffer_len_two buffer_one buffer_two	transmit_fs	This command transmits a datagram.
TRANSMIT_XID_CMD (0x0E)	station_id (*) rsap (*) xmit_queue_one xmit_queue_two buffer_len_one buffer_len_two buffer_one buffer_two	transmit_fs	This command transmits an XID (transmit ID) command with the poll bit set to on.
TRANSMIT_XID_RESP _FINAL (0x0F)	station_id (*) rsap (*) xmit_queue_one xmit_queue_two buffer_len_one buffer_len_two buffer_one buffer_two	transmit_fs	This command transmits an XID response with the final bit on.
TRANSMIT_XID_RESP ONSE_NOT_FINAL (0x10)	station_id (*) rsap (*) xmit_queue_one xmit_queue_two buffer_len_one buffer_len_two buffer_one buffer_two	transmit_fs	This command transmits an XID response with the final bit off.

DLC Command Specifics

BUFFER_FREE

This command returns a linked list of buffers to the SAP's buffer pool. The address of the first buffer in the linked list is passed as a buffer address. The parameters are passed indirectly (using the **parameter** pointer in the CCB) in the following format:

Offset	Parameter	Type
0	station_id	(in) unsigned int
2	buffer_left	(out) unsigned int
4	reserved	4 bytes
8	first_buffer	(in) far pointer

`station_id:`	SAP number using buffer. Only SAP number portion of station id is used (the link access station portion is ignored).
`buffer_left:`	Upon completion, this variable will be set to the total number of buffers available in the SAP buffer pool.
`first_buffer:`	Pointer to first buffer to be added to the pool. A NULL value will cause an immediate return with no buffers freed.

BUFFER_GET

This command gets one or more buffers from the SAP buffer pool. The parameters are passed indirectly (using the **`parameter`** pointer in the CCB) in the following format:

Offset	Parameter	Type
0	station_id	(in) unsigned int
2	buffer_left	(out) unsigned int
4	buffer_get	(in) unsigned int
5	reserved	3 bytes
8	first_buffer	(out) far pointer

`station_id:`	SAP number using buffer. Only SAP number portion of station id is used (the link access station portion is ignored).
`buffer_left:`	Upon completion, this variable will be set to the total number of buffers available in the SAP buffer pool.
`buffer_get:`	The number of buffers to return, or 1 if the default value is requested (i.e., the field is set to zero).
`first_buffer:`	Pointer to first buffer available.

The first four bytes of each buffer (bytes 0 – 3) are a far pointer to the next buffer in the linked list of buffers. The final buffer in the list will contain a NULL pointer for the first four bytes. User data is placed in the buffer starting at byte number 4.

DLC_CLOSE_SAP

This command closes a service access point. Prior to calling this command, all associated link access stations must be closed using the DLC_CLOSE_STATION command. The station_id of the SAP to close is placed in the first two bytes of the **`parameter`** field of the CCB. There is no parameter structure associated with this command.

DLC_CLOSE_STATION

This command closes a link access station. The station_id of the link access station to close is placed in the first two bytes of the **`parameter`** field of the CCB. There is no parameter structure associated with this command.

DLC_CONNECT_STATION

This command is used to complete a connection between two applications using link access stations to perform connection-oriented communications. Both applications must issue a DLC_CONNECT_STATION for this command to work. The parameters are passed indirectly (using the **`parameter`** pointer in the CCB) in the following format:

Offset	Parameter	Type
0	station_id	(in) unsigned int
2	reserved	2 bytes
4	routing_addr	(in) far pointer

`station_id:`	This is the link access station ID to be connected, as returned by the DLC_OPEN_STATION command.
`routing_addr:`	This is a far pointer to a routing address. If the pointer is NULL, the station is assumed to be on the local token ring network. The routing address consists of up to three 6-byte addresses of intermediate gateway nodes.

DLC_FLOW_CONTROL

This command is used to control the flow of frames through a SAP (which will affect both datagram communications *and* all link access stations for that SAP). The first two bytes of the **`parameter`** field of the CCB contain the station_id of the SAP you are interested in controling. The third byte contains the flow control option byte. This bytes functions as follows:

- If bit 7 is off (0), the SAP enters a "local busy" state. If bit 7 is on, the adapter exits its "local busy" state in accordance with bit 6.

- If bit 6 is off, the adapter exits from a user set "local busy" state. If this bit is on, the adapter exits from an adapter set "local busy" state.

DLC_MODIFY

This command is used to modify default values for an open link access station or SAP. The parameters are passed indirectly (using the **`parameter`** pointer in the CCB) in the following format:

Offset	Parameter	Type
0	reserved	2 bytes
2	station_id	(in) unsigned int
4	timer_t1	(in) unsigned char
5	timer_t2	(in) unsigned char
6	timer_ti	(in) unsigned char
7	maxout	(in) unsigned char
8	maxin	(in) unsigned char
9	maxout_incr	(in) unsigned char
10	max_retry_cnt	(in) unsigned char
11	reserved	3 bytes
14	access_priority	(in) unsigned char
15	reserved	1 byte
19	group_count	(in) unsigned char
20	group_list	(in) far pointer

station_id:	SAP or link access station to be modified.
timer_t1:	Number of timer ticks to wait for an acknowledgement prior to generating an interrupt. The value of each tick is set when the adapter is opened (or as a configuration parameter when the adapter software is loaded). Valid range is 1 through 10.
timer_t2:	Number of timer ticks to delay prior to acknowledging a connection-oriented frame. This delay allows multiple frames to be received and acknowledged simultaneously. Valid range is 1 through 10. If the value is over 10, the delay is set to zero. If the value is zero, the current value is unchanged.
timer_ti:	Number of timer ticks to wait for link activity prior to generating an interrupt. Valid range is 1 through 10
maxout:	Number, between 1 and 127, of outstanding frames that can be transmitted over a connection prior to receipt of an acknowledgment. This value is called the transmit window in many protocol books.
maxin:	Number, between 1 and 127, of received frames that the station can receive over a connection prior to sending an acknowl-

edgment. This value is called the receive window in many protocol books.

`maxout_incr:`	This parameter is designed to reduce bridge congestion over multiple network connections. If the t1 timer expires and the adapter is forced to retransmit a frame, the transmit window is reset to a size of one. Each successful acknowledgment then causes the adapter to increase the transmit window by **`maxout_incr`** until it is eventually restored to the original value set in **`maxout`**.
`max_retry_cnt:`	Number of retry attempts, between 1 and 255, for transmissions where no acknowledgement is received.
`access_priority:`	The transmit access priority for the token ring network. Valid numbers are 0 through 3, with 3 being the highest priority. The actual access priority is left shifted five places prior to storing in this byte (i.e., the format is B'nnn00000'.
`group_count:`	Number of entries in group list (below).
`group_list:`	Far pointer to group list. Each entry in the group list is a 1-byte SAP number with the low order bit set to zero to join that SAP's group or 1 to leave that SAP's group.

For all parameters, a value of zero will leave the current settings unchanged.

DLC_OPEN_SAP

This command is used to open a SAP and override default parameters. The parameters are passed indirectly (using the **parameter** pointer in the CCB) in the following format:

Offset	Parameter	Type
0	station_id	(out) unsigned int
2	user_stat_value	(in) unsigned int
4	timer_t1	(in) unsigned char
5	timer_t2	(in) unsigned char
6	timer_ti	(in) unsigned char
7	maxout	(in) unsigned char
8	maxin	(in) unsigned char
9	maxout_incr	(in) unsigned char
10	max_retry_cnt	(in) unsigned char
11	max_members	(in) unsigned char
12	max_i_field	(in) unsigned int
14	sap_value	(in) unsigned char
15	options_priority	(in) unsigned char
16	station_count	(in) unsighed char
17	reserved	2 bytes
19	group_count	(in) unsigned char
20	group_list	(in) far pointer
24	dlc_status_exit	(in) far pointer
28	dlc_buf_size	(in) unsigned int
30	dlc_pool_len	(in) unsigned int

32	dlc_pool_addr	(in) far pointer

station_id: Station_id for the opened SAP. This value should be stored, as it will be used to identify this SAP for other functions.

user_stat_value: On entry to the DLC status interrupt function (as set in **dlc_status_exit**), this value is passed back to the user function in register SI.

timer_t1: Number of timer ticks to wait for an acknowledgement prior to generating an interrupt. The value of each tick is set when the adapter is opened (or as a configuration parameter when the adapter software is loaded). Valid range is 1 through 10. The default is 5.

timer_t2: Number of timer ticks to delay prior to acknowledging a connection oriented frame. This delay allows multiple frames to be received and acknowledged simultaneously. Valid range is 1 through 10. If the value is over 10, the delay is set to zero. If the value is zero, the default value of 2 is used.

timer_ti: Number of timer ticks to wait for link activity prior to generating an interrupt. Valid range is 1 through 10. The default is 3.

maxout:	Number, between 1 and 127, of outstanding frames that can be transmitted over a connection prior to receipt of an acknowledgement. This value is called the transmit window in many protocol books. The default is 2.
maxin:	Number, between 1 and 127, of received frames that the station can receive over a connection prior to sending an acknowledgment. This value is called the receive window in many protocol books. The default is 1.
maxout_incr:	This parameter is designed to reduce bridge congestion over multiple network connections. If the t1 timer expires and the adapter is forced to retransmit a frame, the transmit window is reset to a size of one. Each successful acknowledgement then causes the adapter to increase the transmit window by **maxout_incr** until it is eventually restored to the original value set in **maxout**. The default is 1.
max_retry_cnt:	Number of retry attempts, between 1 and 255, for transmissions where no acknowledgment is received. The default is 8.
max_members:	If this SAP is a group SAP, this field designates the maximum number of individual SAPs that may join this group. This field should normally be left at zero,

because this will default to the value specified in **DIR_OPEN_ADAPTER**, which is the largest permissible value anyway.

max_i_field:	This parameters specifies the largest I frame that can be received by this SAP. I frames are the data packets used for connection-oriented service. The default is 600 bytes.
sap_value:	The SAP value you wish assigned to this SAP. This value should be an even number because the low order bit is used by the adapter to designate if this SAP is an individual or group SAP.
options_priority:	Bit field with the following meaning: Bits 7–5, ring access priority; normally 0 Bit 4, reserved; set to zero Bit 3, XID option; 0 means adapter handles XID frames, 1 means XID frames are passed to your applica tion Bit 2, individual SAP bit (1 implies indi vidual SAP) Bit 1, group SAP bit (1 implies group SAP) Bit 0, group member bit (1 implies member of group SAP)
station_count:	Link stations to reserve within this SAP. A value of zero will reserve no link sta-

	tions and prevent you from performing connection-oriented communication.
group_count:	Number of entries in group list (below).
group_list:	Far pointer to group list. Each entry in the group list is a 1-byte SAP number with the low-order bit set to zero to join that SAP's group or one to leave that SAP's group.
dlc_status_exit:	Interrupt function to call if the DLC status changes.
dlc_buf_size:	The size of the buffers in the SAP buffer pool. The minimum size is 80, and the number must be a multiple of 16. The default is 160 bytes.
dlc_pool_len:	The number of 16 byte blocks (not buffers) in the buffer pool. The default is 256 (4096 bytes).
dlc_pool_addr:	The location within PC memory where the adapter should build the SAP buffer pool.

DLC_OPEN_STATION

This command is used to open a link access station and override default parameters. The parameters are passed indirectly (using the **parameter** pointer in the CCB) in the following format:

Offset	Parameter	Type
0	sap_station_id	(in) unsigned int
2	link_station_id	(out) unsigned int
4	timer_t1	(in) unsigned char
5	timer_t2	(in) unsigned char
6	timer_ti	(in) unsigned char
7	maxout	(in) unsigned char
8	maxin	(in) unsigned char
9	maxout_incr	(in) unsigned char
10	max_retry_cnt	(in) unsigned char
11	rsap_value	(in) unsigned char
12	max_i_field	(in) unsigned int
14	access_priority	(in) unsigned char
15	reserved	1 byte
16	destination	(in) far pointer

sap_station_id:	Station_id for the opened SAP, as returned by **DLC_OPEN_SAP**.
link_station_id:	This is the link access station station_id which is returned by the adapter for later use by you.
timer_t1:	Number of timer ticks to wait for an acknowledgment prior to generating an interrupt. The value of each tick is set when the adapter is opened (or as a configuration parameter when the adapter software is loaded). Valid range is 1 through 10. The default is 5.

`timer_t2:`	Number of timer ticks to delay prior to acknowledging a connection-oriented frame. This delay allows multiple frames to be received and acknowledged simultaneously. Valid range is 1 through 10. If the value is over 10, the delay is set to zero. If the value is zero, the default value of 2 is used.
`timer_ti:`	Number of timer ticks to wait for link activity prior to generating an interrupt. Valid range is 1 through 10. The default is 3.
`maxout:`	Number, between 1 and 127, of outstanding frames that can be transmitted over a connection prior to receipt of an acknowledgment. This value is called the transmit window in many protocol books. The default is 2.
`maxin:`	Number, between 1 and 127, of received frames that the station can receive over a connection prior to sending an acknowledgment. This value is called the receive window in many protocol books. The default is 1.
`maxout_incr:`	This parameter is designed to reduce bridge congestion over multi-

	ple network connections. If the t1 timer expires and the adapter is forced to retransmit a frame, the transmit window is reset to a size of one. Each successful acknowledgment then causes the adapter to increase the transmit window by **maxout_incr** until it is eventually restored to the original value set in **maxout**. The default is 1.
max_retry_cnt:	Number of retry attempts, between 1 and 255, for transmissions where no acknowledgement is received. The default is 8.
rsap_value:	This is the SAP number you wish to communicate with on the destination adapter. Note that this is a SAP number, *not* a station_id for the remote adapter (which you would have no way of knowing).
max_i_field:	This parameters specifies the largest I frame that can be received by this station. I frames are the data packets used for connection oriented service. The default is the number set with **DIR_OPEN_ADAPTER**.
access_priority:	The access priority, between 0 and 3, to be used for transmitted frames. The format is B'nnn00000'.

	The default is zero. Using numbers larger than zero will cause the open to fail if the adapter is not authorized to use the higher priority.
destination:	Far pointer to a 6-byte location which contains the destination node address.

DLC_REALLOCATE

This command is used to increase or decrease the number of link stations available for a given SAP. The parameters are passed indirectly (using the **parameter** pointer in the CCB) in the following format:

Offset	Parameter	Type
0	station_id	(in) unsigned int
2	option_byte	(out) unsigned char
3	station_count	(in) unsigned char
4	adapter_available_stns	(out) unsigned char
5	sap_available_stns	(out) unsigned char

station_id:	The SAP station_id to be modified.
option_byte:	Bit 7 indicates if you want the number of link stations increased (0) or decreased (1). Bits 0 through 6 are reserved and should be set to zero.

station_count:	The number of link stations to be added or deleted (in accordance with the option byte).
adapter_available_stns:	Number of available link stations remaining for this adapter after this command completes.
sap_available_stns:	Number of available link stations remaining for this SAP after this command completes.

DLC_RESET

This command resets one or more SAPs and their associated link access stations. The SAPs are closed after queued transmissions are completed. The station_id to reset is placed in the two high order bytes of the **parameter** field of the CCB. If the station_id is zero, *all* SAPs (and their link access stations) will be reset.

LLC_SET_THRESHOLD

This command applies to OS/2 only.

DLC_STATISTICS

This command reads the DLC logs and can also be used to reset the logs. The parameters are passed indirectly (using the **parameter** pointer in the CCB) in the following format:

Offset	Parameter	Type
0	sap_station_id	(in) unsigned int

2	log_buf_length	(in) unsigned int
4	log_buffer_addr	(in) far pointer
8	log_act_length	(out) unsigned int
10	options	(in) unsigned char

sap_station_id:	The station_id of the SAP or link access station you are interested in.
log_buf_length:	The length of your buffer space you have allocated for the log buffer.
log_buffer_addr:	A pointer to the log buffer space that you have allocated for this purpose.
log_act_length:	The actual length of the data transferred to your log buffer. If this length is greater than **log_buf_length** the remaining data is simply discarded by the adapter.
options:	Bit 7 on causes logs to be reset, off leaves logs intact. Bits 0 through 6 are reserved and should be zero.

For a SAP, the log format is as follows:

Bytes	Type	Meaning
0–3	unsigned long	Number of frames transmitted.
4–7	unsigned long	Number of frames received.

8–11	unsigned long	Number of frames discarded (no receive outstanding)
12–15	unsigned long	Number of times data was lost.
16–17	unsigned int	Numbers of buffers available in buffer pool.

For a link access station, the log format is as follows:

Bytes	Type	Meaning
0–1	unsigned int	Number of I frames transmitted
2–3	unsigned int	Number of I frames received
4	Unsigned char	Number of I frame receive errors
5	Unsigned char	Number of I frame xmit errors
6–7	unsigned int	Number of times t1 expired
8	unsigned char	Last command/response received
9	unsigned char	Last command/response sent
10	unsigned char	Link primary state
11	unsigned char	Link secondary state
12	unsigned char	Send state variable
13	unsigned char	Receive state variable
14	unsigned char	Last received NR
15	unsigned char	Length of network header in xmits
16–47	char[]	Network header being used

DIR_CLOSE_ADAPTER

This command will shut down the adapter and terminate all outstanding CCBs. If the adapter was opened with a lock code, the lock code must be placed in the first two bytes of the `parameter` field of this CCB. Trying to close the adapter with an invalid lock code results in a return code of 0x05.

DIR_CLOSE_DIRECT

This command applies to OS/2 only.

DIR_DEFINE_MIF_ENVIRONMENT

This routine allows you to write a NetBIOS emulator which processes NetBIOS NCBs using your own code. See IBM (1988) if you are interested in this area.

DIR_INITIALIZE

This command initializes the adapter, resetting all tables and buffers and causing the adapter to run the bring-up tests. If this command is executed while the adapter is already open, all outstanding CCBs are lost. The parameters are passed indirectly (using the **parameter** pointer in the CCB) in the following format:

Offset	Parameter	Type
0	bring_ups	(out) unsigned int
2	sram_address	(in/out) unsigned int
4	reserved	4 bytes
8	adptr_chk_exit	(in) far pointer
12	netw_status_exit	(in) far pointer
16	pc_error_exit	(in) far pointer

bring_ups: If the adapter detects an error during bring-up tests, it returns a value of 0x07 for this CCB and sets the **bring_up** field to the specific error detected. Bring-up error

	codes are included in the next section.
`sram_address:`	This field is a segment value. If a nonzero number is included in this field on command execution, the adapter will locate shared RAM at the specified address. A zero as input causes the adapter to use the default values (0xD800 for the primary adapter, 0xD400 for the secondary). Upon command completion, this field is set to the actual shared RAM segment.
`adptr_chk_exit:`	This field is a function pointer to your interrupt function you want the adapter to call if it detects an adapter hardware error during execution. If the field is zero, no user function will be called.
`netw_status_exit:`	This field is a function pointer to your interrupt function you want the adapter to call if it detects a network problem. If the field is zero, no user function will be called.
`pc_error_exit:`	This field is a function pointer to your interrupt function you want the adapter to call if it detects a PC hardware or operating system error.

DIR_INTERRUPT

This command forces an adapter interrupt, but performs no action.

DIR_MODIFY_OPEN_PARMS

This command allows you to temporarily modify many values set when the adapter was initially opened. The parameters are restored with the **`DIR_RESTORE_OPEN_PARMS`** command. This command will fail if a receive command is outstanding. After this command has been completed, it may not be used again until after a **`DIR_RESTORE_OPEN_PARMS`** has been issued. The parameters are passed indirectly (using the **`parameter`** pointer in the CCB) in the following format:

Offset	Parameter	Type
0	dir_buf_size	(in) unsigned int
2	dir_pool_blocks	(in) unsigned int
4	dir_pool_address	(in) far pointer
8	adptr_chk_exit	(in) far pointer
12	netw_status_exit	(in) far pointer
16	pc_error_exit	(in) far pointer
20	open_options	(in) unsigned int

`dir_buf_size:`	The size of each buffer in the direct buffer pool, including overhead. The minimum is 80, and the length must be a multiple of 16. The default is 160.

dir_pool_blocks:	The length of the direct buffer pool in terms of 16-byte blocks.
dir_pool_address:	Far pointer to the beginning of the buffer pool.
adptr_chk_exit:	This field is a function pointer to your interrupt function you want the adapter to call if it detects an adapter hardware error during execution. If the field is zero, no user function will be called.
netw_status_exit:	This field is a function pointer to your interrupt function you want the adapter to call if it detects a network problem. If the field is zero, no user function will be called.
pc_error_exit:	This field is a function pointer to your interrupt function you want the adapter to call if it detects a PC hardware or operating system error.
open_options:	Various options, each represented by one bit. A bit turns the option on, 0 turns it off. Bit 15 is the leftmost bit.

The **open_options** bit fields have the following meanings:

bit	meaning
15	Wrap interface. All user transmissions will be wrapped around as received data without going on the network.
14	Disable hard error. Prevents network hard errors from causing interrupts.
13	Disable soft error. Prevents network soft errors from causing interrupts.
12	Pass adapter MAC frames. Causes all adapter class MAC frames which are not recognized by the adapter to be passed to the application (they are ignored by default). MAC frames are covered in the next chapter.
11	Pass attention MAC frames. Causes all attention MAC frames to be passed to the adapter. Multiple identical attention MAC frames will only result in one frame being passed to your application (the first one). By default, these frames are not passed to the application.
10	Reserved. Set to zero.
9	Pass parameter table. If the adapter is already open and this bit is set, all fields will be filled with the current values being used by the adapter.
8	Contender. This option allows the adapter to participate in contention for token ring media access protocols if necessary.

7	Pass beacon MAC frames. Passes all unique beacon MAC frames to the application.
6	Reserved. Set to zero.
5	Remote program load. This bit prevents your adapter from joining the ring until at least one other adapter is up on the ring.
4	Token release. This bit turns off the early token release option of the newer 16/4 adapters. This capability is discussed in the token ring network hardware section.
0-3	Reserved. Set to zero.

DIR_OPEN_ADAPTER

This command opens the adapter. The parameters were discussed in detail in Section 4.2.

DIR_OPEN_DIRECT

This command applies to OS/2 only.

DIR_READ_LOG

This command reads and resets the direct logs. The parameters are passed indirectly (using the `parameter` pointer in the CCB) in the following format:

Offset	Parameter	Type
0	log_id	(in) unsigned int

2	log_buf_length	(in) unsigned int
4	log_buf_addr	(in) far pointer
8	log_act_length	(out) unsigned int

log_id:	Identifies the log to read, as follows: 0x0000 – read adapter error log; 0x0001 – read direct interface error log; 0x0002 – read both logs.
log_buf_length:	The length of your buffer space you have allocated for the log buffer.
log_buffer_addr:	A pointer to the log buffer space that you have allocated for this purpose.
log_act_length:	The actual length of the data transferred to your log buffer. If this length is greater than **log_buf_length**, the remaining data is simply discarded by the adapter.

For the adapter, the log format is as follows:

Bytes	Type	Meaning
0	unsigned char	Line errors
1	unsigned char	Internal errors
2	unsigned char	Burst errors
3	unsigned char	A/C errors
4	unsigned char	Abort delimiter

5	1 byte	Reserved
6	unsigned char	Lost frames
7	unsigned char	Receive congestion
8	unsigned char	Frame copied errors
9	unsigned char	Frequency errors
10	unsigned char	Token errors
11	3 bytes	Reserved

For the direct interface, the log format is as follows:

Bytes	Type	Meaning
0-3	unsigned long	Number of frames transmitted
4-7	unsigned long	Number of frames received
8-11	unsigned long	Number of frames discarded (no receive outstanding)
12-15	unsigned long	Number of times data was lost
16-17	unsigned int	Numbers of buffers available in buffer pool

DIR_RESTORE_OPEN_PARMS

This command restores the default parameters to the adapter after they have been temporarily reset using **DIR_MODIFY_OPEN_PARMS.**

DIR_SET_EXCEPTION_FLAGS

This command only applies to OS/2.

DIR_SET_FUNCTIONAL_ADDRESS

This command allows you to modify the functional address for the adapter. The four bytes of the **parameter** field contain a bit pattern representing the *bits to change*, not the actual new address. The bits field is formated as a 4-byte character array. The least significant and most significant bit are ignored. For example, 0xFFFFFFFF will reset all address bits, while 0x00000060 will reset bits 5 and 6.

DIR_SET_GROUP_ADDRESS

This command allows you to set a group address which the adapter will use to receive messages. The **parameter** field contains 4-byte character array which represents the group address which should be added.

DIR_SET_USER_APPENDAGE

This command allows you to modify the interrupt functions set when the adapter was opened or initialized. The parameters are passed indirectly (using the **parameter** pointer in the CCB) in the following format:

Offset	Parameter	Type
0	adptr_chk_exit	(in) far pointer
4	netw_status_exit	(in) far pointer
8	pc_error_exit	(in) far pointer

adptr_chk_exit: This field is a function pointer to your interrupt function you want the adapter to call if it detects an

	adapter hardware error during execution. If the field is zero, no user function will be called.
`netw_status_exit:`	This field is a function pointer to your interrupt function you want the adapter to call if it detects a network problem. If the field is zero, no user function will be called.
`pc_error_exit:`	This field is a function pointer to your interrupt function you want the adapter to call if it detects a PC hardware or operating system error.

An NULL pointer for any field will restore that function pointer to the address being used before the last call to **DIR_SET_USER_APPENDAGE.**

DIR_STATUS

This command returns adapter status information. The parameters are passed indirectly (using the **`parameter`** pointer in the CCB) in the following format:

Offset	**Parameter**	**Type**
0	encoded_addr	(out) char[6]
6	node_address	(out) char[6]
12	group_address	(out) char[4]
16	functional_addr	(out) char[4]

20	max_sap	(out) unsigned char
21	open_sap	(out) unsigned char
22	max_station	(out) unsigned char
23	open_station	(out) unsigned char
24	avail_station	(out) unsigned char
25	adapter_config	(out) unsigned char
26	microcode_level	(out) char[10]
36	adapter_parms_addr	(out) far pointer
40	adapter_mac_addr	(out) far pointer
44	tick_cntr_addr	(out) far pointer
48	last_ntwk_status	(out) unsigned int

encoded_addr:	The permanent address encoded by the manufacturer on the adapter.
node_address:	The adapter's network address, which will equal the encoded_addr unless modified (for example, during **DIR_OPEN_ADAPTER**).
group_address:	The adapter's group address.
functional_addr:	The adapter's functional address.
max_sap:	The maximum number of SAPs allowed for this adapter.
open_sap:	The number of SAPs which are currently open.
max_station:	The maximum number of link access stations allowed for this adapter (across all SAPs).

open_station:	The number of link access stations which are currently open.
avail_station:	The number of link access stations which are available. Link access stations are considered unavailable if the either are already open or if they were reserved when a SAP was opened.
adapter_config:	A bit field in which bit 7 indicates if this is an original PC network adapter, bit 4 indicates if the early token release capability of the 16/4 adapter is turned on, and bit 0 indicates the adapter's data rate (0 = 4, 1 = 16 Mbps).
microcode_level:	A number representing the release of the adapter microcode.
adapter_parms_addr:	The address of the adapter's memory containing adapter parameters. This memory cannot be written by an application program.
adapter_mac_addr:	The address of the adapter's memory containing MAC buffers. This memory cannot be written by an application program.
tick_cntr_addr:	The address of an unsigned long containing the number of 100-milli-

	second intervals that have ellapsed since the last **DIR_INITIALIZE** command.
last_ntwk_addr:	The most recent network status change.

DIR_TIMER_CANCEL

This command cancels a timer set with the **DIR_TIMER_SET** command. The **parameter** field of the CCB contains a far pointer to the CCB block used to start the timer.

DIR_TIMER_CANCEL_GROUP

This command cancels a group of timers. The **parameter** field of the CCB contains a far pointer to an interrupt function. The interrupt function is the function which **DIR_TIMER_SET** was instructed to call when the timer expired. All timers using this interrupt function as their exit function will then be canceled.

DIR_TIMER_SET

This command starts a timer. The first two bytes of the **parameter** field contain an unsigned integer indicating the number of timer ticks to wait. The valid range is 0 through 13,107. Each timer tick is .5 seconds. When the command completes, the interrupt function pointed to in the **post** field of the CCB is called.

PDT_TRACE_ON

This command starts an interrupt trace for all adapter traffic. The trace capability stores all CCBs which are started, all CCBs which are completed, all NCBs which are started, and all adapter interrupts of the PC. Timer interrupts are stored and a count is output when a non-timer interrupt occurs. The parameters are passed indirectly (using the **parameter** pointer in the CCB) in the following format:

Offset	Parameter	Type
0	table_length	(in) unsigned int
2	current_off	(out) unsigned int
4	start_tick_0	(out) unsigned long
8	stop_tick_0	(out) unsigned long
12	start_tick_1	(out) unsigned long
16	stop_tick_1	(out) unsigned long
20	reserved	12 bytes
31	table	char[???]

table_length:	The length of the trace table you have setup in your memory space. The minimum value is 256 bytes. Each entry in the trace table is 16 bytes long.
current_off:	This is the offset into the trace table. This field is updated continuously as the trace table is updated. The trace table wraps when full.

start_tick_0:	This is the value of the timer 0 tick counter when the trace started.
stop_tick_0:	This is the value of the timer 0 tick counter when the trace stopped.
start_tick_1:	This is the value of the timer 1 tick counter when the trace started.
stop_tick_1:	This is the value of the timer 1 tick counter when the trace stopped.
table:	This is the space where the actual trace table entries are made. This array must be equal is size to the number specified in **table_length**.

Four trace entry formats are used (and will be intermixed in the trace table). Each entry is 16 bytes long. The valid ranges for byte 0 of each entry are different for each of the four formats, allowing the type of interrupt to be determined. The four valid formats are

CCB Trace Entry

Byte	Meaning	
0	Adapter number (0/1)	
1	Bit flags	7 = adapter initialized
		6 = initialize in process
		5 = adapter opened
		4 = open in process
		3 = SRB busy
		2 = Block bit on

	1 = always 0
	0 = no adapter found
2	CCB command
3	Return code
4-7	SS:SP registers
8-11	Pointer to interrupted application program code
12-15	Pointer to CCB

Adapter Interrupt Entry

Byte	Meaning
0	Interrupt status register processor (ISRP) even bit flags
	7 = always 1
	6 = always 1
	5 = reserved
	4 = programmable timer interrupt
	3 = error interrupt
	2 = access interrupt
	1 = always 1
	0 = adapter number (0/1)
1	ISRP odd-bit field
	7 = reserved
	6 = adapter check
	5 = SRB response
	4 = ASB free
	3 = ARB command
	2 = SSB response
	1 = reserved
	0 = reserved
2	Command code of interrupt
3	Return code
4-7	SS:SP registers

8-11	Address of interrupted application code
12-15	CCB address, or zero if interrupt not result of CCB

Adapter Timer Interrupt Entry

Byte	Meaning
0	0xD2 for primary adapter, 0xD3 for secondary
1	0x00
2-3	Total timer interrupts (both adapters)
4-7	SS:SP registers
8-11	Address of the interrupted application code
12-15	CCB address if this interrupt causes a DIR_TIMER_SET command to be completed, 0x00 otherwise

NCB Trace Entry

Byte	Meaning
0	0x0F when NCB first issued 0x1F when executing user post routine 0x2F when returning from post routine
1	adapter number (0/1)
2	NCB command
3	return code
4-7	SS:SP registers
8-11	address of interrupted application program code
12-15	pointer to NCB

PDT_TRACE_OFF

This command turns off the adapter trace capability. There are no parameters.

PURGE_RESOURCES

This command applies to OS/2 only.

READ

This command applies to OS/2 only.

READ_CANCEL

This command applies to OS/2 only.

RECEIVE

This command receives all types of DLC frames and places them in the buffer pool for use by your application. If an interrupt handler is defined in the receive command, the command remains active and continues to receive frames until specifically canceled. If no interrupt handler is defined, the command terminates upon completion and must be started again after a frame is received. The parameters are passed indirectly (using the **parameter** pointer in the CCB) in the following format:

Offset	Parameter	Type
0	station_id	(in) unsigned int
2	user_length	(in) unsigned int
4	received_data	(in) far pointer
8	first_buffer	(out) far pointer
12	options	(in) unsigned char

station_id: The station_id to receive frames for. This is the number returned by **DLC_OPEN_SAP** or **DLC_OPEN_STATION**. In addition, the following station_ids are automatically established when the adapter is opened:

0x0000 to receive MAC and non-MAC frames

0x0001 to receive MAC frames

0x0002 to receive non_MAC frames

user_length: This allows you to tell the adapter to reserve a space at the start of each buffer for user data. The length of this user space is specified in this field.

received_data: This is a far pointer to a function which should be called each time a frame is received. When this interrupt handler is called, the DS:SI registers are set to point to the CCB and the ES:BX registers are are set to point to the first received buffer.

first_buffer: The address of the first received buffer.

`options:`	Bit flags with the following meanings:
	7 – (on) store all MAC frames contiguously and in their entirety; (off) store MAC frame headers in buffer 1, remaining frame data in the second buffer
	6 – same as 7, but applies to non-MAC frames
	5 – place all data in second buffer, leaving first buffer empty except for header information
	0-4 reserved, set to zero

The buffer format for noncontiguous receipts (the default) is as follows:

Buffer One format

Offset	**Type**	**Meaning**
0	far pointer	Pointer to next buffer, or NULL for last
4	unsigned int	Length of entire received frame
6	unsigned int	Number of frame bytes in this buffer
8	unsigned int	Offset from buffer start to user field

10	unsigned int	Length of user field
12	unsigned int	Receiving station_id
14	unsigned char	Option byte used in receive
16	unsigned int	Buffers remaining
18	unsigned char	Frame status field from frame
19	unsigned char	Adapter number (0/1)
20	unsigned char	LAN header length
21	unsigned char	DLC header length
22	char[32]	LAN header
54	char[4]	DLC header
58	char[???]	User space defined by user_length
??	char[???]	Received data

Note that the received data *always* starts at the value user_offset plus user_length.

Buffer Two format

Offset	**Type**	**Meaning**
0	far pointer	Pointer to next buffer, or NULL for last
4	unsigned int	Length of entire receive frame
6	unsigned int	Number of frame bytes in this buffer
8	unsigned int	Offset from buffer start to user field
10	unsigned int	Length of user field
12	char[???]	User space defined by user_length
??	char[???]	Received data

RECEIVE_CANCEL

This command cancels an outstanding receive command for a station_id. The station_id you are interested in is placed in the first two bytes of the **parameter** field of the CCB as an unsigned int.

RECEIVE_MODIFY

This command receives data and puts some of the data into a local user buffer (not one assigned from the SAP buffer pool). This command is not normally used by application programmers. If necessary, refer to the suggested readings for command specifics.

TRANSMIT_I_FRAME

This command transmits one or more buffers using a connection over a link access station. All buffers are transmitted (and must fit within) one frame. This size is limited based on the *least capable* adapter you will encounter on your network. The limitations are 2025 for the original adapters, 4441 for the 16/4 adapters running at 4 Mbps, and 17937 for the 16/4 adapters running at 16 Mbps. The adapter software automatically adds the frame's DLC and LAN header, so each buffer contains actual data only. The parameters are passed indirectly (using the **parameter** pointer in the CCB) in the following format:

Offset	Parameter	Type
0	station_id	(in) unsigned int
2	transmit_fs	(out) unsigned char
3	reserved	1 byte
4	xmit_queue_one	(in) far pointer
8	xmit_queue_two	(in) far pointer

12	buffer_len_one	(in) unsigned int
14	buffer_len_two	(in) unsigned int
16	buffer_one	(in) far pointer
20	buffer_two	(in) far pointer

The adapter will send all data pointed to by the transmit queues/buffers in the following order:

1. Transmit queue one will be used.
2. Transmit queue two will be used.
3. Buffer one will be used.
4. Buffer two will be used.

You may put your outgoing data in any combination of the four queues/buffers, although applications typically do all transmissions using buffer one only. The field descriptions are as follows:

station_id:	The station_id of the link station as returned by **DLC_OPEN_STATION.**
transmit_fs:	The frame status field values after the frame has made a complete circuit around the ring.
xmit_queue_one:	The first of a linked list of transmit buffers. Each buffer starts with a far pointer to the next buffer (or NULL for the last buffer), followed by

- A 2-byte reserved field.

- An unsigned int giving the lenght of actual data in this buffer.
- An unused unsigned int for user use.
- An unsigned int giving the size of user space.
- A character array equal in size to the user space just specified.
- The actual data.

xmit_queue_two:	The first of another linked list of transmit buffers using the same format as transmit queue one.
buffer_len_one:	The length of the first transmit buffer.
buffer_len_two:	The length of the second transmit buffer.
buffer_one:	The address of the first buffer of data to transmit. The buffer contains data only.

buffer_two:	The address of the second buffer of data to transmit. The buffer contains data only.

Transmit queue one and two buffers can be obtained from the SAP buffer pool using **buffer_get**. If this approach is used, transmit queue two buffers will be automatically freed (using **buffer_free**) after the frame is successfully transmitted. This is *not* true of transmit queue one.

TRANSMIT_TEST_CMD

This command transmits a test command frame with the poll bit set. This command is normally not used by application programs. Refer to the Suggested Readings for further information.

TRANSMIT_UI_FRAME

This command transmits a datagram over a SAP (not a link access station). The application program is responsible for providing the LAN header, although the adapter will add the DLC header. All buffers are transmitted (and must fit within) one frame. This size is limited based on the *least capable* adapter you will encounter on your network. The limitations are 2025 for the original adapters, 4441 for the 16/4 adapters running at 4 Mbps, and 17937 for the 16/4 adapters running at 16 Mbps. The parameters are passed indirectly (using the **parameter** pointer in the CCB) in the following format:

Offset	Parameter	Type
0	station_id	(in) unsigned int
2	transmit_fs	(out) unsigned char
3	rsap	(in) unsigned char

4	reserved	8 bytes, set to zero
12	buffer_len_one	(in) unsigned int
14	buffer_len_two	(in) unsigned int
16	buffer_one	(in) far pointer
20	buffer_two	(in) far pointer

`station_id:`	The station_id of the link station as returned by **`DLC_OPEN_SAP`**.
`transmit_fs:`	The frame status field values after the frame has made a complete circuit around the ring.
`rsap:`	The remote SAP number you wish the data sent to. Note that this number is *not* necessarily equal to the remote station_id associated with this SAP.
`buffer_len_one:`	The length of the first transmit buffer.
`buffer_len_two:`	The length of the second transmit buffer.
`buffer_one:`	The address of the first buffer to transmit. This buffer *must* contain the LAN header only.
`buffer_two:`	The address of the second buffer of data to transmit. The buffer contains data only.

The LAN header has the following format:

Offset	Type	Meaning
0	unsigned char	AC byte, added by adapter
1	unsigned char	FC byte, added by adapter
2	char[6]	Destination node address (12 digits)
8	char[6]	Your node address, added by adapter
14	char[??]	0-16 bytes of routing information

The only field you are required to enter is the destination node address. You need to enter 1, 2, or 3 intermediate node addresses (routing information) if the frame must be routed through 1, 2, or 3 gateway nodes to different token rings. The remaining fields are filled in by your adapter automatically.

TRANSMIT_XID_CMD

This command transmits an XID command with the pool bit set on. This command is normally not used by an application program. Refer to the Suggested Readings for command specifics.

TRANSMIT_XID_RESP_FINAL

This command transmits an XID response with the final bit on. This command is normally not used by an application program. Refer to the Suggested Readings for command specifics.

TRANSMIT_XID_RESP_NOT_FINAL

This command transmits an XID response without the final bit on. This command is normally not used by an application program. Refer to the Suggested Readings for command specifics.

4.8 Summary of DLC Return Codes

CCB command return codes are returned in the **`retcode`** field of the Command Control Block structure. If you are using the **`int_adapter()`** function defined earlier in this chapter, the return code is also placed in the global variable **`net_error`**. A return of 0x00 is always a valid return without error. A return code of 0xFF indicates that the command is continuing to operate. A return of anything else indicates some type of error. The specific meaning of each possible return code is as follows:

Code	Name	Description	Action
0x00	SUCCESS	Operation completed normally.	None.
0x01	INVALID_COMMAND	The command code passed in the CCB was invalid.	Use a valid code.
0x02	DUPLICATE_COMMAND	Only one command of this type can be outstanding at a time, but you tried to execute a second one.	Wait for the earlier command to complete.
0x03	ADAPTER_OPEN	This command requires that the adapter be closed, but the adapter is already open.	Close the adapter.
0x04	ADAPTER_CLOSED	This command requires that the adapter be open, but the adapter is closed.	Open the adapter.
0x05	MISSING_PARAM	A required parameter was not provided.	Check your input parameters to be sure that no required parameters are coded to zero.

0x06	INVALID_OPTIONS	An invalid option was provided, or a combination of options is invalid.	Check your option lists and try again.
0x07	UNRECOVERABLE_F AILURE	The adapter has been closed because of an unrecoverable error condition.	Determine the cause of the error, correct the error if necessary, then initialize and open the adapter.
0x08	UNAUTHORIZED_PR IORITY	The requested access priority is not authorized for this adapter.	Use a lower priority. 0 is always valid.
0x09	NOT_INITIALIZED	The adapter must be initialized for this command to work, and it has not been initialized.	Initialize the adapter.
0x0A	USER_CANCEL	The command was successfully cancelled per user request.	None.
0x0B	CLOSE_CANCEL	The adapter was closed while this command was in progress.	Determine why the adapt-er was closed.
0x0C	NOT_OPEN_SUCCESS	The command completed, alth-ough the adapter is not opened.	None.
0x10	NETBIOS_FAILURE	NetBIOS was accessed but it is not loaded, or one or more NetBIOS parameters used during the adapter open command was invalid.	If you will be using NetBIOS, close the adapter, correct the prob-lems, then reopen the adapter.
0x11	TIMER_ERROR	A timer value for timer_set or timer_cancel is not in the range of 0-13107, or you tried to cancel a timer which was never set.	Correct and try again.
0x12	WORK_AREA_OVERF LOW	The available work area has over-flowed. The work area includes the adapter's internal memory and the application provided work space.	Reduce the values for max_station and/or max_sap. You can also increase the memory made available to the adapter to match the value returned in the work_len_act field.
0x13	INVALID_LOG_ID	The requested log_id is not de-fined.	Correct and retry.

0x14	INVALID_RAM	The shared RAM segment or size is invalid.	Adjust the value. Values must often be even multiples of 16.
0x15	LOG_OVERFLOW	The buffer allocated for the log was too small, resulting in the loss of some statistics. The information that overflowed is permanently lost if the command indicated reset.	Be sure to use a buffer size which is large enough.
0x16	BUFFER_TOO_LARGE	The requested buffer size cannot be satisfied using the SAP buffer pool.	Increase the SAP buffer pool size or decrease the requested buffer size.
0x17	NETBIOS_OPERATIONAL	Attempt to change a NetBIOS parameter which is currently being used by NetBIOS.	Close, then reopen the adapter.
0x18	INVALID_BUFFER_LENGTH	The specified SAP buffer size is invalid.	The size must be at least 80 bytes and a multiple of 16.
0x19	NO_BUFFERS	Inadequate buffers remain to satisfy the request.	Retry with fewer buffers or wait for more buffers to become available.
0x1A	USER_LENGTH_TOO_LARGE	The user requested area is too large for the available buffer sizes.	Reduce the user length field value.
0x1B	PARAMATER_INVALID	The CCB parameter field pointer is invalid. This can be caused by the pointer pointing into the PC system interrupt vector area or being too near the end of the segment which will cause wrap-around for some of the fields.	Correct and retry.
0x1C	INVALID_POINTER	A pointer within a parameter table is invalid.	Correct and retry.
0x1D	INVALID_CCB_ADAPTER	The ccb_adapter value is outside of the prescribed range.	Correct and retry.
0x20	LOST_DATA_NO_BUFFERS	There were no available buffers in the SAP's buffer pool when a frame was received, resulting in lost data. This error only occurs for connectionless transmissions.	Free some buffers (buffer_free), then retry.

0x21	LOST_DATA_BUFFER_OVERFLOW	There was inadequate space in the SAP's buffer pool to hold a received frame. Part of the frame will be lost. This message only occurs for connectionless transmissions.	Free some buffers, then retry.
0x22	TRANSMIT_ERROR	The frame was not successfully transmitted.	Check the FS byte in the CCB to determine the cause of failure.
0x23	FRAME_ERROR	A frame error was detected during transmission. This may indicate that corrupted data was received by the destination.	Application specific.
0x24	UNAUTHORIZED_MAC	An attempt was made to send a MAC frame which this adapter was not authorized to do. Possible causes include an invalid source class, an attempt to send a MAC fram or a SAP, or an attempt to send a MAC frame on the PC Network (not token ring).	Adjust the source class value and try again.
0x25	MAX_XMIT_CMDS	128 transmit commands are already cued for this station.	Wait for some commands to complete.
0x27	LINK_NOT_AVAILABLE	An error was detected over a connection, causing the connection to be closed.	Try to re-open the connection using dlc_connect_station.
0x28	INVALID_FRAME_LENGTH	The frame length is to short to contain header information or too long for the transmit buffer. If you are using a connection, this error also causes the connection to enter a disconnected state.	Modify the frame length. For connection, re-open the connection.
0x30	INADEQUATE_RCV_BUFFERS	There were an inadequate number of receive buffers allocated when the adapter was opened.	Free up RAM using open adapter parameters.
0x32	NODE_ADDRESS	The defined node address is invalid.	The node address contains an unallowed bit or number.
0x33	INVALID_REC_BUF_LEN	The receive buffer length is over the allowed maximum, less than the allowed minimum, or not a multiple of 8.	Adjust and retry.

0x34	INVALID_XMIT_BUF_LEN	The transmit buffer length is over the allowed maximum, less than the allowed minimum, or not a multiple of 8.	Adjust and retry.
0x40	INVALID_STATION_ID	The station_id either does not exist or is not valid for this particular command.	Be sure that you are using the SAP or link access station station_id as assigned by the adapter.
0x41	PROTOCOL_ERROR	Attempt to connect a link station while the link is disconnected or closed (you must first open it), or to transmit over a connection which is not connected.	Correct your application code.
0x42	PARAMETER_TOO_LARGE	One or more parameters exceed the maximum allowed.	Correct and retry.
0x43	INVALID_SAP	The SAP value is either invalid or already in use.	Invalid SAPs are the null, global, and group sap. Correct and retry.
0x44	INVALID_ROUTE	The routing field is too short, larger than 18 bytes, or an odd number of bytes long.	Correct and retry.
0x45	INVALID_GROUP_REQUEST	An attempt was made to join a nonexistent group.	Correct and retry.
0x46	INADEQUATE_LINK_STATIONS	When opening a SAP, this error indicates that the adapter has inadequate link stations remaining to satisfy the request. When opening a station, this error indicates that all assigned link stations for this SAP are already in use.	Correct and retry.
0x48	LINK_STATION_OPEN	An attempt was made to close a SAP which has one or more link stations open.	Close the link stations prior to closing the SAP.
0x49	GROUP_SAP_FULL	The group SAP is currently full.	Application specific.
0x4A	SEQUENCE_ERROR	The station is closing or establishing a connection while you are attempting to execute a command.	Wait for the command to complete before trying your comand.
0x4B	STATION_CLOSE_NO_ACK	The station closed without remote acknowledgment.	Application specific.

0x4C	OUTSTANDING_COMMANDS	Attempt to close a link station while outstanding commands are queued.	Wait until commands complete or issue a reset.
0x4D	NO_CONNECTION	The link station could not establish a connection.	Verify rsap values, routing information, the remote adapter address, and physical connectivity, then try again.
0x4F	INVALID_ADDRESS	The remote address is not valid because the high bit is set to 1 which indicates a group address, but a group address is not allowed for this command.	Correct the remote address.

Other types of data which the adapter passes to your application are covered individually:

Adapter Status Parameter Table

This information is returned in response to a DIR_STATUS COMMAND:

Offset	Name	Type	Meaning
0	phys_addr	char[4]	Adapter physical address
4	up_node_addr	char[6]	Address of next node in ring
10	up_phys_addr	char[4]	Physical address of next node
14	poll_addr	char[6]	Last poll address
20	auth_env	char[2]	Authorized environment
22	acc_priority	char[2]	Transmit access priority
24	source_class	char[2]	Source class authorization
26	att_code	char[2]	Last attention code
28	source_addr	char[6]	Last source address
34	beacon_type	char[2]	Last beacon type
36	major_vector	char[2]	Last major vector
38	netw_status	char[2]	Network status
40	soft_error	char[2]	Soft error timer value
42	fe_error	char[2]	Front end error counter
44	local_ring	char[2]	Ring number
46	mon_error	char[2]	Monitor error code
48	beacon_transmit	char[2]	Beacon transmit type
50	beacon_receive	char[2]	Beacon receive type
52	frame_correl	char[2]	Frame correlation save
54	beacon_naun	char[6]	Beaconing station NAUN
60	reserved	char[4]	

64	beacon_phys	char[4]	Beaconing station physical address

Frame Status Byte

After each frame makes a circuit around the ring, the frame status (FS) byte can be examined. Some values and their meanings are

0xCC	The frame was copied
0x00	No adapter recognized the address
0x88	The destination adapter saw the frame but didn't copy it.

Bring-up Error Codes

Code	Meaning
0x0020	Diagnostics could not execute
0x0022	ROM diagnostics failed
0x0024	Shared RAM diagnostics failed
0x0026	Processor instruction test failed
0x0028	Processor interrupt test failed
0x002A	Shared RAM interface register diagnostics failed
0x002C	Protocol handler diagnostics failed
0x0040	Adapter's programmable timer for the PC system failed
0x0042	Cannot write to shared RAM
0x0044	Cannot read from shared RAM
0x0046	Allowed to write into shared RAM read-only area
0x0048	Initialization timed out

4.9 Suggested Reading

IBM (1988), *Local Area Network Technical Reference*, Research Triangle Park, NC: International Business Machines Corporation.

IBM (1987), *Token-Ring Network Architecture Reference,* Research Triangle Park, NC: International Business Machines corporation.

Poo, Gee-Swee and Wilson Ang (1989), "Data Link Driver Program Design for the IBM Token Ring Metwork PC Adapter", *Computer Communications*, Vol. 12, no. 5, (October), pp. 266-272.

5. Register Direct Programming

It is possible to program the adapter without requiring that *any* adapter support software be loaded to provide such "fluff" as DLC control, NetBIOS support, or heaven forbid, BIOS redirectors. The question you must ask yourself is *why* would anyone in his or her right mind want to do this? I must confess that it is interesting to have an understanding about how the adapter works when you strip away the insulating shells, and there *is* a certain macho pride in feeling like you can do it if you really need to . . . but let's stop at that point and not *really* try to do things the hard way. With that warning in mind, this chapter will explain how the adapter works at the lowest possible level and will provide sufficient information to give you a good head start if you ever find an application that absolutely requires you to work at this level. We will not try to present detailed code examples or sample applications for this level of programming. If you need to "make it work", use this chapter as a starting point; then read Chapter seven of IBM (1988) about 15 times and it will start to make sense.

5.1 Talking to the Adapter

Communication between your application and the adapter is accomplished using three mechanisms:

1. The adapter supports programmed I/O (PIO) ports at fixed memory locations. In Turbo C these ports can be accessed using **`inport()`**, **`inportb()`**, **`outport()`**, and **`outportb()`** functions. PIO ports are discussed further in Section 5.2.

2. The adapter supports memory-mapped I/O (MMIO), which is accessed as fixed addresses relative to a starting address which *can* change. MMIO addresses can be accessed in Turbo C using **`peek()`**, **`peekb()`**, **`poke()`**, and **`pokeb()`**. MMIO addresses are actually mapped to RAM/ports on the adapter. MMIO is discussed in Section 5.2.

3. The adapter supports shared RAM in your application's address space. This shared RAM is used for passing control blocks and actual data back and forth. Shared RAM is discussed in Section 5.5.

5.2 Programmed I/O

You can perform three functions using PIO with ordinary PC adapters:

1. Control adapter interrupts.

2. Determine the starting address of the MMIO area and the current interrupt level.

3. Control adapter resets.

5.2.1 Controlling Adapter Interrupts

You can enable interrupts for all installed adapters (primary and alternate) with a write to address 0x02Fn, where "n" is the desired interrupt level. Valid interrupt levels are 0 through 3, with the meaning of each discussed in Section 5.2.2. For example, to enable interrupts using interrupt level 0 for all installed adapters, you would write

```
outportb(0x02F0, 1);
```

The actual value output (1, in this case) is irrelevant. The simple act of writing anything is what performs the desired action.

Similarly, you can enable interrupts for just the primary adapter by writing to address 0x0A23, or just the alternate adapter by writing to address 0x0A27. In this case, the interrupt level cannot be changed.

5.2.2 Determining MMIO Starting Location

The primary and alternate adapter will each have an independent memory mapped I/O (MMIO) area, and each will be located at a different location. To determine the starting address for the MMIO for the primary adapter, you read from a port located at 0x0A20 (the address is 0x0A24 for the alternate adapter). The byte value can be read as follows:

```
unsigned char  byte;

byte = inportb(0x0A20);
```

Bits 2 through 7 indicate the starting address of the MMIO area (bits 0 and 1 will be discussed momentarily). The following code converts the byte value returned into a far pointer to the start of the MMIO:

```
void             *mmio;
unsigned int     segment;

byte &= 0x03;          /* mask lower two bits */
segment = byte;        /* convert to integer */
segment <<= 7;         /* left shift by 7 */
mmio = MK_FP( (segment, 0);
```

Before you start writing me letters, yes *I know* that there are much more efficient ways to do the same thing I do in this code fragment. In all code examples in this chapter, I am describing the algorithm using straightforward, crude, often inefficient C code to make it very clear what is going on. If you have read this far, you are probably a better C programmer than I anyway, and you will not have any difficulties taking my examples and making them more efficient to your hearts content!

You can also mask off bits 2 through 7 of the byte and use the remaining two bits (bits 0 and 1). These two bits tell you the current interrupt level set for the adapter as follows:

Value	**PC I/O Bus**	**Micro Channel**
0	IRQ2	IRQ2
1	IRQ3	IRQ3
2	IRQ6	IRQ10
3	IRQ7	IRQ11

We will see how to use these numbers later when using interrupts to communicate with the adapter.

5.2.3 Controlling Adapter Resets

You can force the adapter to enter a reset mode which is similar to the power-on state. To force the adapter to enter the reset mode, write to address 0x0A21 (or 0x0A25 for the alternate adapter). The adapter will then stay in this reset mode until you write to the adapter reset release port (0x0A22 for the primary adapter, 0x0A26 for the alternate).

5.3 Memory Mapped I/O

Allright, the two key things you've used the PIO to learn are the starting location of the MMIO area and the interrupt level of the adapter. Let's put that information to some use. We will start with the MMIO segment, determined as described in Section 5.2.2.

In that section we combined the segment with a zero offset to determine the starting location of the MMIO area. To actually perform MMIO functions, we need to modify the offset as follows:

- Bits 0 - 4 select a particular register of interest.
- Bits 5 and 6 select the operation to perform on the register.
- Bits 7 and 8 select the area of interest within the MMIO area.

Let's talk about each of these categories individually, starting with bits 7 and 8. These bits operate as follows:

- 00 selects the attachment control area. This is the normal selection.

- 01 is reserved.
- 10 selects the adapter identification area A containing the adapter encoded address.
- 11 selects the adapter identification area B containing test patterns.

In general, you will only be concerned with the attachment control area, as this is where the MMIO registers are located. Bits 5 and 6 allow you to perform four operations on these registers:

- 11 is used to read from a register.
- 00 is used to write to a register.
- 10 is used to bitwise OR a byte with a register.
- 01 is used to bitwise AND a byte with a register.

Bits 0 through 4 select the register you are interested in. There are 18 registers, as follows (9 pairs of even and odd):

- RRR (shared RAM relocation registers) – even and odd
- WRBR (write region base registers) – even and odd
- WWCR (write window close registers) – even and odd
- WWOR (write window open registers) – even and odd
- ISRP (interrupt status registers – PC system) – even and odd

- ISRA (interrupt status registers – adapter) – even and odd

- TCR (timer control registers) – even and odd

- TVR (timer value registers) – even and odd

- SRPR (shared RAM paging registers) – even and odd

RRR even uses bit pattern 0000, RRR odd uses bit pattern 0001, WRBR even uses bit pattern 0010, WRBR odd uses bit pattern 0011, and so on. We will discuss the purpose of these registers next, but first a hint about accessing them. The following example shows a convenient way to address various registers assuming that the variable **segment** was previously set to point to the top of the MMIO area:

```
#define RRR_EVEN        0x00
#define RRR_ODD 0x01
#define WRBR_EVEN       0x02
#define WRBR_ODD        0x03
#define WWCR_EVEN       0x04
#define WWCR_ODD        0x05
#define WWOR_EVEN       0x06
#define WWOR_ODD        0x07
#define ISRP_EVEN       0x08
#define ISRP_ODD        0x09
#define ISRA_EVEN       0x0A
#define ISRA_ODD        0x0B
#define TCR_EVEN        0x0C
#define TCR_ODD 0x0D
#define TVR_EVEN        0x0E
#define TVR_ODD 0x0F
#define SRPR_EVEN       0x10
#define SRPR_ODD        0x11

#define READ            0x60
#define OR              0x40
#define AND             0x20
#define WRITE   0x00

unsigned int    segment;
unsigned char   byte;

/* write 0x00 to RRR_EVEN */
pokeb(segment, RRR_EVEN | WRITE, 0x00);

/* read TVR_EVEN into byte */
byte = peekb(segment, TVR_EVEN | READ);
```

Now let's talk about each of the registers individually, starting with the RRR registers. RRR_EVEN is used to set the starting address of shared RAM (this register is unused for PCs with the Micro Channel bus). Bits 1 through 7 of this register map to bits 13 through 19 of the shared RAM address, so writing 0x02 to this register sets the shared RAM address to 8K, 0x04 to 16K, 0x06 to 24K, and so on. The shared RAM address boundary set using this register must be an even multiple of the shared RAM size, which brings us to register RRR_ODD.

RRR_ODD is used to determine the amount of shared RAM used by your adapter. Bits 2 and 3 of this register can be read to determine the shared RAM size as follows:

- 00 for 8K
- 01 for 16K
- 10 for 32K
- 11 for 64K

You must remember to mask the remaining bits prior to doing your comparison.

We've now used the MMIO's RRR registers to determine the amount of shared RAM supported by the adapter and to set the shared RAM starting location to a value we have allocated from the global heap. Note that because the shared RAM must begin on a fixed boundary, you will normally need to allocate more memory than required (using **`malloc()`**), then use as your address the first valid address within the allocated block of memory. Don't forget to save your original pointer so that the memory block can be freed when you are done. The next question is, How is the shared memory controlled to prevent the adapter and our application program from simultaneously accessing the same memory area? The answer is the write management register pairs.

Although your application can always read data from anywhere in the shared RAM area, there are only two regions where writes are allowed (writes to other areas generate a PC access error interrupt). These two regions are called the *write region* and the *write window*. The write region base register (WRBR) points to the start of the write region. The top of the write region is the end of the shared RAM block which was setup. Similarly, the write window wpen register (WWOR) points to the start of the write window within the shared RAM, while the write window close register (WWCR) points to the end of the write window. Either the write region or the write window (or both) may zero size (closed). If any of these registers has a value of zero, the associated window is closed for all writing. If the value is nonzero, you must convert the register value into an actual address as described next.

Recall that each of these three registers (WRBR, WWCR, and WWOR) is actually a pair of registers, one even and one odd. The even and odd register values are combined to produce a 16-bit *offset* into the shared RAM. The even register contains the most significant byte of this offset, while the odd register contains the least significant byte of the offset.

The interrupt status registers (ISRA and ISRP) are used by the adapter to interrupt your application and by your application to interrupt the adapter. These registers are covered in depth in Section 5.4.

Three timer registers are used by your application:

- TCR_EVEN is used to control the timer.
- TCV_ODD is used to select a countdown timer initial value.

- TCV_EVEN contains the actual value of the timer.

TCR_EVEN contains 6 bits for your use. The bits have the following meanings:

- Bit 2: PC system interlock. This bit is set when the adapter wants to prevent your application from accessing any of the timer registers while critical functions are being performed.

- Bit 3: PC system programmable timer count status. This bit is set by the adapter when the countdown timer contains a nonzero value.

- Bit 4: PC system programmable timer overrun status. This bit is set by the adapter when the countdown timer expires and is not reset by your application.

- Bit 5: PC system programmable timer count gate. This bit is used by the application program to control the countdown timer. Writing a one to this location starts the countdown timer counting. Writing a zero pauses the timer. Writing a one when the timer has already expired (reached zero) causes the timer to be reloaded and restarted.

- Bit 6: PC system programmable timer reload mode. If this bit is one, the timer is automatically reloaded when it expires. If this bit is zero, the timer must be manually reloaded using bit 5.

- Bit 7: PC system programmable timer interrupt mask. If this bit is one, the timer will interrupt your application when the countdown timer expires. If this bit is zero, the timer will not interrupt your application and you must manually check the timer values periodically. The discussion of ISRP and ISRA in the next section discuss the process of interrupting your application in more detail.

The timer value registers (even and odd) contain timer values in 10 millisecond increments. The timer value is initially written to TVR_ODD (and changed by writing a value to TVR_ODD. It must then be transferred to TVR_EVEN and started when you want it to commence counting. This is accomplished using the appropriate bits in TCR_EVEN, as discussed earlier.

The shared RAM page register (even) is used for paging of shared RAM to and from your PC-accessible memory (the odd register is not used). This register is only used on computers supporting RAM paging. For details, refer to IBM (1988).

5.4 Interrupt Status Registers

Your application and the adapter communicate using interrupts. These interrupts are initiated via the interrupt status register adapter (ISRA) and the interrupt status register PC (ISRP). The ISRA_ODD register is used by your application to interrupt the adapter. To interrupt the adapter, a specific bit is written to the ISRA_ODD address using a **pokeb()** call. To understand these interrupts fully, we must first look ahead and examine how you communicate commands and data to the adapter.

You application write three types of data into the shared RAM: Data Holding Buffers (DHBs), System Request Blocks (SRBs), and Adapter Status Blocks (ASBs). You read four types of data from shared RAM: System Status Blocks (SSBs), Adapter Request Blocks (ARBs), Receive Buffers (RBs), and SAP and Link Station Control

Blocks. The exact nature of each of these blocks will be covered in the next section.

With this in mind, the following bits may be used:

- Bit 5 indicates that you have placed a new command in the SRB and are ready for the adapter to process the command.

- Bit 4 indicates that you have placed a response (an ASB) in the shared RAM which is available for the adapter's use.

- Bit 3 indicates that you are ready to put an SRB in the shared RAM, but that a previous command is still pending. The adapter will then interrupt you when the previous command is completed.

- Bit 2 indicates that you are ready to put an ASB in the shared RAM, but that a previous ASB is still pending. The adapter will then interrupt you when the previous ASB is copied.

The ISRA_EVEN register bits provide current adapter status information. These bits are normally not used by an application program, but their meaning is as follows:

- Bit 7 – Internal parity error (on adapter's internal bus)

- Bit 6 – Timer interrupt pending

- Bit 5 – Access interrupt (attempt by *adapter* to access illegal address)

- Bit 4 – Adapter microcode problem (microcode dead-man timer expired)
- Bit 3 – Adapter processor check status
- Bit 2 – Reserved
- Bit 1 – Adapter hardware interrupt mask (prevents internal interrupts)
- Bit 0 – Adapter software interrupt mask (prevents internal software interrupts)

The ISRP registers are used by the adapter to interrupt your application. The actual interrupt will occur as a hardware interrupt using the IRQ number available at MMIO address 0x0A20 (primary adapter) or 0x0A24 (alternate adapter). The selected IRQ number is mapped to an MS-DOS interrupt number by taking the IRQ number and adding 0x08 (i.e., IRQ0 = MS-DOS interrupt 0x08, IRQ1 = MS-DOS interrupt 0x09, etc.). Prior to activating the token ring adapter, you must ensure that the MS-DOS interrupt vector for the appropriate interrupt number is set to *your* interrupt handler. The interrupt handler is simply a function declared to be of type `void interrupt`. The procedure for changing the normal interrupt processing is as follows:

1. Determine the MS-DOS interrupt number by first finding (or setting) the adapter's internal IRQ number.
2. Use `getvect()` to read and store the current value for this interrupt. The stored value will be a far pointer to the current interrupt processing code.

3. Use **`setvect()`** to modify the current value for this interrupt to your own interrupt function. **`setvect()`** is passed the interrupt number of interest and a far pointer to your interrupt function.

4. Within your interrupt function, use values stored in ISRP_ODD and ISRP_EVEN (discussed next) registers to determine if the interrupt was generated by the adapter for you.

5. If the interrupt was for you, process the interrupt expeditiously and return. If the interrupt was not for you, call the *original* interrupt code returned from **`getvect()`**; then return.

As we just mentioned, most of the bits in ISRP_EVEN and ISRP_ODD are designed to let you know if an interrupt was for you, and if so, what the nature of the interrupt was. Starting with ISRP_ODD, the meaning of appropriate bits is

- Bit 6 – Adapter check. The adapter has encountered a serious problem and has closed itself. There are procedures, described in IBM (1988), for determining the cause of the problem.

- Bit 5 – SRB response. The adapter has accepted an SRB request and set the return code within the SRB.

- Bit 4 – ASB free. The adapter has read the ASB and this area can be safely reused. This interrupt is only used if your aplication has set the ASB free request bit in ISRA_ODD or if an error was detected in your response.

- Bit 3 – ARB command. The adapter has given you a command for action. The command is located in the ARB area of shared memory.
- Bit 2 – SSB response. The adapter has posted a response to your SRB (the response is located in the SSB area of shared memory).
- Bit 1 – Bridge frame forward complete.

Within ISRP_EVEN, the following bits are used to describe interrupt conditions:

- Bit 4 – Timer interrupt. The TVR_EVEN timer has expired.
- Bit 3 – Error interrupt. The adapter has had an internal error.
- Bit 2 – Access interrupt. You have attempted to write to an invalid area of shared RAM or an invalid register within the MMIO.

In addition, the following bits within ISRP_EVEN can be turned on or off by you to control the interrupt processing:

- Bit 7 – If 0 the adapter will issue a CHCK, if 1 an IRQ. This should normally be set (by you) to 1.
- Bit 6 – Interrupt enable. If 0, no interrupts will occur. If 1, interrupts will occur normally. Normally set to 1.
- Bit 0 – Primary or alternate adapter. Set to zero if this

adapter is the primary adapter, 1 if this adapter is the alternate adapter.

We've kind of danced around the terms DHBs, SRBs, ASBs, etc., alluding to the fact that they are areas within the shared RAM. Now that you understand how ISRP and ISRA registers are used to communicate (via interrupts) back and forth between the adapter and your application, we are ready to discuss the structure of shared RAM.

5.5 Shared RAM

There are four formatted control blocks used for communication between the adapter and your application:

1. The System Request Block (SRB)
2. The System Status Block (SSB)
3. The Adapter Request Block (ARB)
4. The Adapter Status Block (ASB)

The System Request Block is used to pass a command and its associated parameters from your application to the adapter. The SRB is functionally identical to the Net Control Blocks and Command Control Blocks discussed in earlier chapters. If the command is completed immediately, the return values will be passed back in the SRB space. If the command is accepted but not completed, the SRB return code field is set to 0xFF.

The System Status Block is used when the adapter accepts a command but does not complete the command immediately. The SSB is used to pass back return values when the SRB command is finally completed.

The Adapter Request Block is used by the adapter to communicate with your application. If the ARB contains information only, your application should note the information, then inform the adapter that the ARB has been read. If the ARB asks for some type of response, you notify the adapter that the ARB has been read and pass your response to the adapter using the ASB (discussed next).

The Adapter Status Block is used by your application to respond to an ARB issued by the adapter.

All four blocks are located in the previously identified shared RAM, but where? You begin by initializing the adapter using PIO and MMIO operations. After the adapter is initialized, the WRBR register tells you the base of the write region offset within the shared RAM. This is where you place your first SRB, a command to open the adapter. After the adapter is open, the response returned by the adapter tells you the location of the four block areas within the shared RAM (i.e., the SRB location, SSB location, ARB location, and ASB location). You store these four addresses and use them until the adapter is closed (due to a DIR_CLOSE_ADAPTER, DIR_CONFIGURE_BRIDGE_RAM, DIR_OPEN_ADAPTER, or error condition which causes the adapter to automatically close). You must then repeat the procedure and store the shared RAM addresses. In summary, the steps involved in opening the adapter are

1. Issue an adapter reset PIO command.

2. Delay for at least 50 milliseconds to ensure that the adapter responds.

3. Issue an adapter release PIO command.

4. Set the interrupt enable bit (bit 6) in the ISRP_EVEN register.

5. Wait for the adapter to interrupt you (via ISRP_ODD bit 5). This takes between one and three seconds. At that point you can can use the WRBR to determine the address of a SRB containing diagnostic information regarding the adapter, if necessary.

6. Use the shared RAM segment address combined with the WRBR offset to post a DIR_OPEN_ADAPTER command. The specifics of this command are covered in the following section.

7. When the DIR_OPEN_ADAPTER command completes (you will be interrupted with ISRP_ODD bit 5), read bytes 6 through 15 in the SRB to determine the value for ASB_address, SRB_address, ARB_address, and SSB_address. These addresses are the offset from the start of shared RAM to the area used for reading/writing the specified block of data. Unless specifically mentioned, the internal structure of the SRB and SSB are identical.

Now that the mechanics of communicating with the adapter are more or less clear, it is time to cover some specifics. This involves looking at the ASB, SRB, ARB, and SSB blocks in more detail. As you read the following section, you will find that the block structures (and the parameters they contain) are very similar to the structure of the associated CCB as discussed in the previous chapter.

5.6 Adapter Command Blocks

You control the adapter using command blocks called System Request Blocks. The general procedure for issuing a command to the adapter is as follows:

1. The appropriate SRB structure is filled with parameters and moved into the SRB area of shared RAM.

2. ISRA_ODD bit 5 is set to interrupt the adapter. The adapter checks the validity of the SRB contents and either

    - Completes the command, sets a return code other than 0xFF, and interrupts the PC using ISRP_ODD bit 5.

    - Performs initial processing only, sets the return code to 0xFF, and provides a command correlator. The PC will normally *not* be interrupted at this point. An exception is that if you have told the adapter that you have another SRB to send (using ISRA_ODD bit 3), the adapter will interrupt you using ISRP_ODD bit 5 to confirm that it has performed initial processing on the SRB.

3. For some commands, the adapter may then request further data using the ARB and DHB blocks (and interrupts to tell you about the request). The PC system uses the ASB command block to respond to these commands.

4. When a command is completed that was started (i.e., the return code was set to 0xFF in the SRB), the adapter puts the final return code in the SSB and interrupts the PC using ISRP_ODD bit 2.

5. After you read the data from an SSB, you inform the adapter that you are done reading it by setting ISRA_ODD bit 0.

The following commands are available using the adapter direct interface. For common commands, the command specifics are included. For less common commands, the command is summarized and interested readers are referred to IBM (1988) for more details. In most cases the parameters are identical (in name and function) to the parameters used for the identical CCB discussed in the previous chapter. In addition, the valid return codes (and their meaning) is identical to the CCB return codes covered in the last chapter.

Important note: At this point, it is necessary to discuss an anomaly in the way the adapter looks at the world versus the way in which Intel microprocessors (including the IBM PC/AT line) look at the world. The adapter stores (and reads) words in a byte-reversed format relative to Intel microprocessors. This means that you must use a macro (or some other method) of switching the byte order for all word values read from or written to shared RAM when communicating with the adapter. This new data type (a byte reversed unsigned integer) will be refered to as a R_WORD for the remainder of this chapter.

5.6.1 DIR_CLOSE_ADAPTER (0x04)

This command is used to close the adapter. The command does not return until completed. The SSB and SRB structure definition is

```
struct
{
        unsigned char       command;
        unsigned char       reserved;
        unsigned char       retcode;
};
```

5.6.2 DIR_INTERRUPT (0x00)

This command forces an adapter interrupt, but performs no operation. The command does not return until completed. The SSB and SRB structure definition is

```
struct
{
        unsigned char       command;
        unsigned char       reserved;
        unsigned char       retcode;
};
```

5.6.3 DIR_MODIFY_OPEN_PARMS (0x01)

This command is used to modify the open_options parameters for the adapter, normally temporarily. The format of the open_options field is covered under DIR_OPEN_ADAPTER. The command does not return until completed. The SSB and SRB structure definition is

```
struct
{
        unsigned char       command;
        unsigned char       reserved1;
        unsigned char       retcode;
        unsigned char       reserved2;
        R_WORD              open_options;
};
```

5.6.4 DIR_OPEN_ADAPTER (0x03)

This command is used to open the adapter for normal ring communications (or for adapter loopback testing). The command does not return until completed. The SSB structure definition is

```
struct
{
        unsigned char       command;
        unsigned char       reserved1;
        R_WORD              open_options;
        char                node_address[6];
        char                group_address[4];
        char                funct_address[4];
        R_WORD              num_rcv_buf;
        R_WORD              rcv_buf_len;
        R_WORD              DHB_length;
        unsigned char       num_DHB;
        unsigned char       reserved2;
        unsigned char       dlc_max_sap;
        unsigned char       dlc_max_sta;
        unsigned char       dlc_max_gsap;
        unsigned char       dlc_max_gmem;
        unsigned char       dlc_t1_tick_one;
        unsigned char       dlc_t2_tick_one;
        unsigned char       dlc_ti_tick_one;
        unsigned char       dlc_t1_tick_two;
        unsigned char       dlc_t2_tick_two;
        unsigned char       dlc_ti_tick_two;
        char                product_id[18];
};
```

The parameters are identical to the similarly named parameters discussed for DIR_OPEN_ADAPTER in Chapter four. One discrepancy is that the transmit buffers from Chapter four are called DHB buffers when using the adapter direct interface.

The SRB response for this command is not identical to the SSB. The format of the SRB response is

```
struct
{
        unsigned char       command;
        unsigned char       reserved1;
        unsigned char       retcode;
        char                reserved2[3];
        R_WORD              open_error_code;
        R_WORD              ASB_address;
        R_WORD              SRB_address;
        R_WORD              ARB_address;
        R_WORD              SSB_address;
};
```

The four addresses are offsets from the beginning of the shared RAM area to the start of the specified block.

5.6.5 DIR_READ_LOG (0x08)

This command reads log data and resets the adapter error counters. The command does not return until completed. The SSB structure is identical to the SRB structure and is defined as

```
struct
{
        unsigned char       command;
        unsigned char       reserved1;
        unsigned char       retcode;
        char                reserved2[3];
        unsigned char       log_data[14];
};
```

Upon return, the 14 bytes of log data have the following meaning:

```
log_data[0] = line errors
log_data[1] = internal errors
log_data[2] = burst errors
log_data[3] = a/c errors
log_data[4] = abort delimiters
log_data[5] = reserved
log_data[6] = lost frames
log_data[7] = receive congestion count
log_data[8] = frame copied errors
log_data[9] = frequency errors
log_data[10] = token errors
log_data[11-13] = reserved
```

5.6.6 DIR_RESTORE_OPEN_PARMS (0x02)

This command restores the adapter parameters to their values prior to calling DIR_MODIFY_OPEN_PARMS. This command does not return until complete. The open_options field is set (by you) to the values stored in the adapter prior to calling DIR_MODIFY_OPEN_PARMS. The structure of the SRB and SSB are identically defined as follows

```
struct
{
        unsigned char       command;
        unsigned char       reserved1;
        unsigned char       retcode;
        unsigned char       reserved2;
        R_WORD              open_options;
};
```

5.6.7 DIR_SET_FUNCT_ADDRESS (0x07)

This command is used to set the functional address for the adapter to receive messages. Bits 31, 1, and 0 of the functional address are ignored. This command does not return until complete. The structure of the SRB and SSB are identically defined as follows

```
struct
{
        unsigned char       command;
        unsigned char       reserved1;
        unsigned char       retcode;
        unsigned char       reserved2;
        unsigned char       funct_address[4];
};
```

5.6.8 DIR_SET_GROUP_ADDRESS (0x06)

This command is used to set the group address for the adapter to receive messages. This command does not return until complete. The structure of the SRB and SSB are identically defined as follows

```
struct
{
        unsigned char       command;
        unsigned char       reserved1;
        unsigned char       retcode;
        unsigned char       reserved2;
        unsigned char       group_address[4];
};
```

5.6.9 DLC_CLOSE_SAP (0x16)

This command is used to close a SAP (see Chapter four for a discussion of SAPs). This command does not return until complete. The structure of the SRB and SSB are identically defined as follows

```
struct
{
        unsigned char       command;
        unsigned char       reserved1;
        unsigned char       retcode;
        unsigned char       reserved2;
        R_WORD              station_id;
};
```

5.6.10 DLC_CLOSE_STATION (0x1A)

This command is used to close a station (see Chapter four for a discussion of stations). This command returns immediately (with a return code of 0xFF) if the initial values are valid, and then interrupts your application with the final return code when the station is successfully closed. The structure of the SRB and SSB are identically defined as follows

```
struct
{
        unsigned char      command;
        unsigned char      cmd_correlate;
        unsigned char      retcode;
        unsigned char      reserved2;
        R_WORD             station_id;
};
```

The **cmd_correlate** field is used to provide a unique identifier for this command block so that you will be able to identify the block later when the command actually completes.

5.6.11 DLC_CONNECT_STATION (0x1B)

This command is used to establish a connection with a remote adapter via an already opened station. This command returns immediately (with a return code of 0xFF) if the initial values are valid, then interrupts your application with the final return code when the connection is successfully established. The structure of the SRB and SSB are identically defined as follows

```
struct
{
        unsigned char       command;
        unsigned char       cms_correlate;
        unsigned char       retcode;
        unsigned char       reserved2;
        R_WORD              station_id;
        char                routing_info[18];
};
```

The **cmd_correlate** field is used to provide a unique identifier for this command block so that you will be able to identify the block later when the command actually completes.

5.6.12 DLC_FLOW_CONTROL (0x1D)

This command is used to control the flow of information into your application via the adapter. This command returns only upon completion. The structure of the SRB and SSB are identically defined as follows

```
struct
{
        unsigned char       command;
        unsigned char       reserved1;
        unsigned char       retcode;
        unsigned char       reserved2;
        R_WORD              station_id;
        unsigned char       flow_options;
};
```

Flow option bits are defined as follows:

Bit 7	Exit (bit value of one) or enter (bit value of zero) a local busy state.
Bit 6	If this bit is one, then a zero in bit 7 will reset the local busy state. If this bit is zero, then a zero in bit 7 will reset the system set busy state (buffer busy).
Bits 5-0	Reserved

5.6.13 DLC_MODIFY (0x1C)

This command is used to modify the adapter parameters. The command does not return until completed. The SSB and SRB are identically defined as

```
struct
{
        unsigned char      command;
        unsigned char      reserved1;
        unsigned char      retcode;
        char               reserved2;
        R_WORD             station_id;
        unsigned char      timer_t1;
        unsigned char      timer_t2;
        unsigned char      timer_ti;
        unsigned char      maxout;
        unsigned char      maxin;
        unsigned char      maxout_incr;
        unsigned char      max_retry_count;
        unsigned char      access_priority;
        unsigned char       sap_gsap_mem;
        unsigned char       gsaps[13];
};
```

The parameters are identical to the similarly named parameters discussed for DLC_OPEN_STATION in Chapter four. The group sap member list (gsaps) contains 0 to 13 items (the exact number is specified by **sap_gsap_mem**. If the low-order bit of a SAP value is 1 then membership in that group SAP is canceled, a 0 indicates that the group SAP should be joined.

5.6.14 DLC_OPEN_SAP (0x15)

This command is used to open a SAP. The command does not return until completed. The SSB and SRB are identically defined as

```
struct
{
        unsigned char       command;
        unsigned char       reserved1;
        unsigned char       retcode;
        char                reserved2;
        R_WORD              station_id;
        unsigned char       timer_t1;
        unsigned char       timer_t2;
        unsigned char       timer_ti;
        unsigned char       maxout;
        unsigned char       maxin;
        unsigned char       maxout_incr;
        unsigned char       max_retry_count;
        unsigned char       gsap_max_mem;
        R_WORD              max_i_field;
        unsigned char       sap_value;
        unsigned char      sap_options;
        unsigned_char      station_count;
        unsigned char      access_priority;
        unsigned char      sap_gsap_mem;
        unsigned char      gsaps[13];
};
```

The parameters are identical to the similarly named parameters discussed for DLC_OPEN_STATION in Chapter four. The group SAP member list (gsaps) contains 0 to 8 items with the exact number specified by **sap_gsap_mem**. If the low-order bit of a SAP value is 1 then membership in that group SAP is canceled, a 0 indicates that the group SAP should be joined.

5.6.15 DLC_OPEN_STATION (0x19)

This command is used to open a link access station. The command does not return until completed. The SSB and SRB are identically defined as

```
struct
{
        unsigned char      command;
        unsigned char      reserved1;
        unsigned char      retcode;
        char               reserved2;
        R_WORD             station_id;
        unsigned char      timer_t1;
        unsigned char      timer_t2;
        unsigned char      timer_ti;
        unsigned char      maxout;
        unsigned char      maxin;
        unsigned char      maxout_incr;
        unsigned char       max_retry_count;
        unsigned char       rsap_value;
        R_WORD              max_i_field;
        unsigned char       station_options;
        unsigned char       reserved;
        char                remote_address[6];
};
```

The parameters are identical to the similarly named parameters discussed for DLC_OPEN_STATION in Chapter four.

5.6.16 DLC_REALLOCATE (0x17)

This command is used to increase or decrease the number link stations which a SAP can support. The command does not return until completed. The SSB and SRB are identically defined as

```
struct
{
        unsigned char       command;
        unsigned char       reserved1;
        unsigned char       retcode;
        char                reserved2;
        R_WORD              station_id;
        unsigned char       option_byte;
        unsigned char       station_count;
        unsigned char       adapter_count;
        unsigned char       sap_count;
};
```

The option byte can be zero to make more link stations available to a SAP, or one to make less link stations available to a SAP. The field station_count is the number of link stations to add or delete. The adapter returns the **`retcode`** field and also sets the field **`adapter_count`** equal to the number of link stations available for the adapter (not allocated to a SAP) and sets **`sap_count`** equal to the number of link stations available for this SAP.

5.6.17 DLC_RESET (0x14)

This command is used to reset either one SAP and all of its link stations or all SAPs and their link stations. The command does not return until completed. The SSB and SRB are identically defined as

```
struct
{
        unsigned char        command;
        unsigned char        reserved1;
        unsigned char        retcode;
        char                 reserved2;
        R_WORD               station_id;
};
```

If **`station_id`** is 0x0000, then all SAPs and their link stations will be reset. Otherwise, the specified SAP and its associated link stations will be reset.

5.6.18 DLC_STATISTICS (0x1E)

This command reads (and optionally resets) statistics for a link station. The command does not return until completed. The SSB and SRB are identically defined as

```
struct
{
        unsigned char        command;
        unsigned char       reserved1;
        unsigned char       retcode;
        char                reserved2;
        R_WORD              station_id;
        R_WORD              counters_addr;
        R_WORD              header_addr;
        unsigned char       header_length;
        unsigned char       reset_option;
};
```

The counters_addr is an offset from the start of the SRB to a table of counters. This table contains the following information:

```
struct
{
        R_WORD              i_frame_xmit_count;
        R_WORD              i_frame_rcv_count;
        unsigned char       i_frame_xmit_err;
        unsigned char       i_frame_rcv_err;
        R_WORD              t1_expired;
        unsigned char       station_rcvd_cmd;
        unsigned char       station_sent_cmd;
        unsigned char       station_prmy_state;
        unsigned char       station_scdy_state;
        unsigned char       station_vs;
        unsigned char       station_vr;
        unsigned char       station_va;
};
```

Most fields are self-explanatory. The `station_rcvd_cmd` and `station_sent_cmd` variables are the *last* command sent or received, not the total. The `station_prmy_state` variable is a bit field with the following meanings:

- Bit 7: Link closed
- Bit 6: Disconnected
- Bit 5: Disconnecting
- Bit 4: Link opening
- Bit 3: Resetting
- Bit 2: FRMR sent
- Bit 1: FRMR received
- Bit 0: Link opened

The **`station_scdy_state`** variable is a bit field with the following meanings:

- Bit 7: Checkpointing
- Bit 6: Local busy (user set)
- Bit 5: Local busy (buffer set)
- Bit 4: Remote busy
- Bit 3: Rejection
- Bit 2: Clearing
- Bit 1: Dynamic window algorithm running

- Bit 0: Reserved (may be 0 or 1)

The header_addr is an offset within the SRB to a copy of the LAN header being used.

5.6.19 TRANSMIT_DIR_FRAME (0x0A)

This command is used to transmit a MAC frame on the token ring network. The command completes as soon as the validity of the SRB has been checked. The format for the SRB is as follows

```
struct
{
        unsigned char       command;
        unsigned char       cmd_correlate;
        unsigned char       retcode;
        unsigned char       reserved;
        R_WORD              station_id;
};
```

When the adapter is ready for you to transfer the actual data to it, it will request the data using a TRANSMIT_DATA_REQUEST ARB. This command is covered later.

After the command completes, the adapter returns an SSB in the following format

```
struct
{
        unsigned char       command;
        unsigned char       cmd_correlate;
        unsigned char       retcode;
        unsigned char       reserved;
        R_WORD              station_id;
        unsigned char       transmit_error;
    };
```

5.6.20 TRANSMIT_I_FRAME (0x0B)

This command is used to transmit data over a connection using a link access station. The procedure and syntax is identical to that for **TRANSMIT_DIR_FRAME** covered in Section 5.6.21.

5.6.21 TRANSMIT_UI_FRAME (0x0D)

This command is used to transmit a datagram using a SAP. The procedure and syntax is identical to that for **TRANSMIT_DIR_FRAME** covered in Section 5.6.21.

5.6.22 TRANSMIT_XID_CMD (0xOE)

This command is used to transmit an XID command. This command is normally used by the link station protocol driver, not an application program. The procedure and syntax is identical to that for **TRANSMIT_DIR_FRAME** covered in Section 5.6.21.

5.6.23 TRANSMIT_XID_RESP_FINAL (0x0F)

This command is used to transmit an XID response (final). This command is normally used by the link station protocol driver, not an application program. The procedure and syntax are identical to that for **TRANSMIT_DIR_FRAME** covered in Section 5.6.21.

5.6.24 TRANSMIT_XID_RESP_NOT_FINAL (0x10)

This command is used to transmit an XID response (not final). This command is normally used by the link station protocol driver, not an application program. The procedure and syntax is identical to that for **TRANSMIT_DIR_FRAME** covered in Section 5.6.21.

5.6.25 TRANSMIT_TEST_CMD (0x11)

This command is used to transmit a test command. The procedure and syntax is identical to that for **TRANSMIT_DIR_FRAME** covered in Section 5.6.21.

5.6.26 DLC_STATUS (0x83)

This command is issued (via interrupt) from the adapter to your application. The command is found in the ARB area of shared RAM and must be acknowledged via ISRA_ODD bit 1. This command requires no response. The adapter is informed of the presence of a response by setting ISRA_ODD bit 4. This command indicates that there has been a change in DLC status. The ARB contains the following information:

```
struct
{
        unsigned char       command;
        char                reserved[3];
        R_WORD              station_id;
        R_WORD              status;
        char                FRMR_data[5];
        unsigned char       access_priority;
        char              remote_address[6];
        unsigned char     rsap_value;
};
```

status is a bit field where each bit has the following meaning:

- Bit 15: Link lost
- Bit 14: DM or DISC received or DISC acknowledged
- Bit 13: FRMR received
- Bit 12: FRMR sent
- Bit 11: SABME received for an open link station
- Bit 10: SABME received, link station opened
- Bit 9: Remote station has entered local busy state
- Bit 8: Remote station has left local busy state
- Bit 7: Ti timer has expired
- Bit 6: DLC counter overflow
- Bit 5: Access priority reduced

5.6.27 RECEIVED_DATA (0x81)

This command is used by the adapter to tell you that a data frame has been received. The format for the ARB is

```
struct
{
        unsigned char       command;
        char                reserved[3];
        R_WORD              station_id;
        R_WORD              receive_buffer;
        unsigned char       lan_header_length;
        unsigned char       dlc_hdr_length;
        R_WORD              frame_length;
        unsigned char       ncb_type;
};
```

The field **receive buffer** is the offset to the first receive buffer in shared RAM. The buffer format is

- Two reserved bytes
- 2-byte R_WORD offset to next buffer + 2
- One reserved byte
- 1-byte FS/Address match (last buffer only)
- 2-byte buffer length (length of data)
- Frame data (n bytes)

The fields **lan_header_length** and **dlc_hdr_length** are the length (in the first buffer) of the LAN and DLC header. The field **ncb_type** can take on any one of the following values based on the frame type:

- 0x02: MAC frame

- 0x04: I frame
- 0x06: UI frame
- 0x08: XID command poll
- 0x0A: XID response final
- 0x0C: XID response not_final
- 0x10: TEST response final
- 0x12: TEST response not_final
- 0x14: Other or unidentified

You must copy the data from the shared RAM to your local memory and then inform the adapter that the data has been transfered by providing a return code to the adapter in an ASB, copying it to the appropriate area of shared RAM, and setting ISRA_ODD bit 4. The format of the ASB used by you to send a message back to the adapter is

```
struct
{
        unsigned char       command;
        char                reserved1;
        unsigned char       retcode;
        char                reserved2;
        R_WORD              station_id;
        R_WORD              receive_buffer;
};
```

The **retcode** field can be 0x00 for success, or 0x20 for "Lost data on receive, no buffers available."

5.6.28 RING_STATUS_CHANGE (0x84)

The adapter uses this ARB to indicate a change in the network status. The format of the ARB is as follows:

```
struct
{
        unsigned char        command;
        char                 reserved[5];
        R_WORD               netw_status;
};
```

netw_status is defined fully in IBM (1988).

5.6.29 TRANSMIT_DATA_REQUEST (0x82)

The adapter informs you that it is ready to receive actual data (in response to a transmit SRB) by sending you a TRANSMIT_DATA_-REQUEST ARB. The format of this ARB is as follows

```
struct
{
        unsigned char        command;
        unsigned char        cmd_correlate;
        char                 reserved[2];
        R_WORD               station_id;
        R_WORD               dhb_address;
};
```

After receiving this ARB, you should read it, then acknowledge it to the adapter using ISRA_ODD bit 1. You then use the Data Hold Buffer (DHB) offset from the ARB to prepare the data for the adapter. This offset is relative to the start of shared RAM. The data written to the DHB address is

- The data only for I frames.
- The entire message (including LAN header) for direct frames (MAC frames).
- For all other frames the format is the LAN header with space reserved for the source address to be inserted by the adapter, followed by three bytes for the adapter to insert the DLC header, followed by the data.

After the data has been transfered to the Data Hold Buffer in shared RAM, an ASB response structure is completed, transfered to the ASB area of shared RAM, and transmitted to the adapter by setting ISRA_ODD bit 4. The ASB structure is defined as follows

```
struct
{
        unsigned char       command;
        unsigned char       cmd_correlate;
        unsigned char       retcode;
        char                reserved;
        R_WORD              station_id;
        R_WORD              frame_length;
        unsigned char       header_length;
        unsigned char       rsap_value;
};
```

5.7 Suggested Readings

IBM (1988), *IBM Local Area Network Technical Reference*, Research Triangle Park, NC: International Business Machines Corporation.

6. Token Ring Adapter Hardware

This chapter briefly discusses the token ring adapter hardware. We begin by discussing the 4-Mbps adapter, then discuss the newer 4/16 Mbps, and finally discuss the adapter cables used to connect adapter cards to the media access unit (MAU).

6.1 4 Mbps Adpaters

The majority of token ring network adapter cards (4 Mbps) are based on the TMS380 chipset. This chipset provides support for token ring networks at the physical and media access control (MAC) level in hardware. The physical layer functions supported are signal coding, clocking, and control of the physical connection to the ring. At the MAC level the chipset supports controlled access to the ring, frame transport service at the MAC level, and error detection. To support these functions, the adapter provides hardware support for

- *LAN processing* to ensure that frames have the correct headers and control information.

- *LAN buffers* for local storage of transmit and receive data awaiting transfer to shared RAM.

- *Host interface* to support interrupt-based communication with the host application.

- *Ring operation and signaling* to support correct ring voltages for signaling, data transmission, and token regeneration.

- *Maintenance and management* to assist in maintaining accurate clocks, detection of faulty ring functioning, and removal of malfunctioning adapters.

The TMS380 chipset consists of five major components:

1. The *system interface* chip controls communication between the adapter and the host PC.

2. The *communications processor* is a microprocessor which executes the MAC processing firmware and controls the on-board buffers.

3. The actual MAC protocol code is stored on the *protocol handler* chip (a ROM).

4 and 5. The *ring interface* and *transceiver* chips interface with the ring itself. Data on the token ring is transmitted in analog format, so these two chips are the interface between the analog world of the ring and the digital world of the remainder of the adapter.

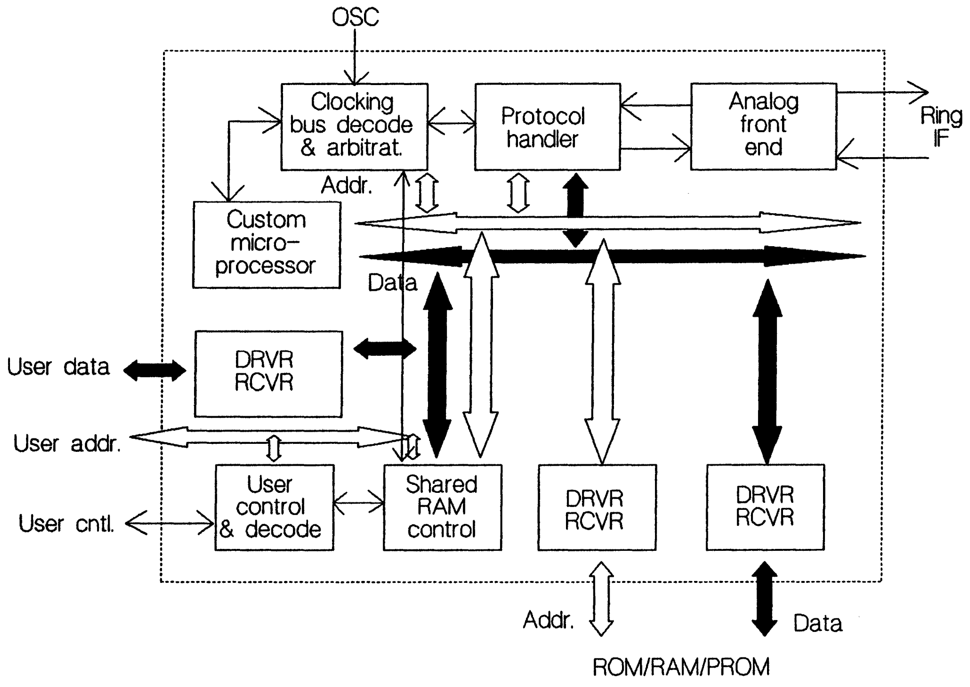


Fig. 6.1 Chip set functional diagram.

6.2 16-Mbps Token Ring Adapter

The newer, 16-Mbps token ring adapters have replaced the five chips required by the original token ring adapter with a single CMOS VLSI module which performs all major LAN adapter functions. Fig. 6.1 is a functional block diagram of the chip. You will notice that all of the functional components found in the earlier chipset are still present within the newer VLSI module. The module is supported by external PROM and RAM modules. The module supports

- Analog data encoding and decoding.

- Address recognition.
- Frame assembly and disassembly.
- Linked buffer list processing.
- Interrupt control.
- Token capture.
- Serialization and deserialization of frames.

In addition, the protocol handler within the module supports state machines to automatically transmit and receive frames. The custom microprocessor shown in the diagram is a 16-bit microprocessor running at 32-MHz and yielding a performance of 3 MIPS. In addition, the microcode within the module has been expanded to support the LLC protocol directly (on chip). For those of you who are into raw numbers, the module contains 106,000 transistors.

6.3 Token Ring Network Adapter Cable

The token ring network adapter uses a cable with a 9-pin D connector at the computer end and a custom 6-pin modular plug at the MAU end. The nine pins at the computer end function as follows:

1. Receive

5. Transmit

6. Receive

9. Transmit

Other pins are unused. The D ring housing is used as shield (ground). For IBM cables, within the cables the wiring is as follows:

- The cables shielding is used for ground.
- The red wire is for pin 1.
- The black wire is for pin 5.
- The green wire is for pin 6.
- The orange wire is for pin 2.

6.4 Suggested Readings

East, W. (1988), "New Developments Lead to Further Integration of a High Performance Token Ring Adapter," *Proceedings of the Networking Technology and Architectures,* (Pinner, UK: Blenheim Online), pp. 89-105.

Lank, K. (1989), "A 16 MBPS Adapter Chip for the IBM Token Ring Local Area Network," *Proceedings of the IEEE 1989 Custom Integrated Circuits Conference*, (May), pp. 11.3.1-11.3.5.

Strole, N. (1989), "Inside Token Ring Version II, according to Big Blue," *Data Communications*, Vol. 18, no. 1, (January), pp. 117-125.

7. Using APPC For Transaction Processing

Many network-oriented software applications are basically transaction oriented. An application "calls up" another application (normally on a different computer), passes a record structure, and requests that some action be performed on that structure. A typical example might be a hospital. A central database containing all patient information might be maintained on an IBM mainframe computer. Local PCs throughout the hospital are used to read/update the information in this central database. For example, the pharmacy might access the patient database to review patient allergies, update the database with the medications the patient is taking, and then later access the patient database to determine the patient's room number so that the medicine can be delivered. The hospital kitchen, switchboard, nurse's stations, and accounting office might also use PCs to access and update this same database. Similar requirements arise from a wide variety of other fields, including airline reservations, point-of-sale systems, and automatic bank teller machines. IBM handles this type of transaction-oriented network environment using the *Application Program-to-Program Communication*, or APPC. This chapter describes the PC version of APPC, known as APPC/PC, and presents some examples illustrating how to use APPC/PC. APPC/PC is sufficiently rich (i.e., complex) that this book cannot do it justice in a single chapter. We can, however, present a sufficient flavor of APPC/PC for you to know if it is worthwhile to pursue the topic using the Suggested Readings.

The following functions are defined in this chapter:

- **`test_appc()`** test for presence of APPC/PC
- **`int_appc()`** process appc command block
- **`ascii_to_ebcdic()`** convert ASCII string to EBCDIC
- **`attach_pu()`** attach a physical unit

7.1 APPC Overview

APPC/PC works over both token ring networks and synchronous data link control (SDLC) connections. Using APPC/PC, it is possible to implement a transaction oriented application which communicates transparently with a wide range of computers, including the following:

- IBM System /370 CICS/VS
- IBM System /370 IMS LU 6.2 Adapter
- IBM System /38
- IBM System /34

- IBM System /1
- IBM System /88
- Other IBM PCs

In general, a connection is established only long enough to complete a transaction. The time required to process a transaction will vary widely, because a transaction can be as short as a record update or as lengthy as a file transfer. When using dial-up phone lines for connectivity, you can set up APPC/PC so that dialing in to the host computer is performed as part of the transaction initiation procedure.

APPC/PC provides a PC based Systems Network Architecture (SNA) programming environment. APPC/PC functions as an SNA logical unit (LU) 6.2 platform and as an SNA physical unit (PU) 2.1 network node. LUs and PUs are discussed further in the following section.

7.2 Addressing in an APPC/PC Environment

There are five addresses you use in an APPC/PC environment:

1. The *network name* is an application defined unique name used for APPC/PC communication. This value must be eight characters long, so names less than eight characters must be blank padded on the right. This name must be known by all other applications wishing to communicate with you.

2. The *physical unit (PU) name* is an 8-character name which is used to tag error messages logged to the system log. This name is normally the same as the network name.

3. The *logical unit (LU) name* which is the same as your network name.

4. Your *LU local address*, which is only used for terminals attached to mainframe computers. This value should be set to zero for PCs.

5. Your *LU adapter address*, which is the 16-byte adapter address (either the ROM address or the address set during adapter configuration).

To complicate the situation further, IBM does not use ASCII for any of these addresses. Each of the addresses must be converted to EBCDIC prior to transmission, and converted back to ASCII for received messages if you intend to display the messages to the user. At least the task is simplified somewhat by a conversion capability built into APPC/PC (discussed later).

7.3 Communicating with APPC/PC

From a network layer perspective, APPC/PC is roughly similar to NetBIOS. Both operate on top of the underlying DLC layer to provide a hardware/network protocol-independent method of communicating with other applications. You will also find that communicating with APPC/PC is roughly similar to working with NetBIOS. You fill in a structure with the command and pass parameters; you call APPC/PC using one of the PC's interrupt vectors; and then you read return values in your structure.

The first thing you must do is test to ensure that APPC/PC is installed and running on your PC. This can be accomplished using the code shown in Code Box 7.1. APPC/PC uses interrupt 0x68. Recall that because each interrupt vector is a far pointer, this interrupt vector is stored at 0x68 * 4. We can read this interrupt vector value, and

```
#include        <dos.h>
#include        "appc.h"

/****************************************************************
*       test_appc() - Test for presence of APPC/PC
*
*       Returns:
*               0 for success (APPC/PC installed)
*               -1 for failure
*
*       History:
*               Original code by William H. Roetzheim, 1990
*****************************************************************/

int             test_appc ()
{
        int     i;
        char    *appc;
        char    *valid = "APPC/PC";

        appc = (char *) getvect(0x68);

        for (i = 0; i < 7; i++)
        {
                if (appc[-9 + i] != valid[i]) return -1;
        }
        return 0;
}
```

Code Box 7.1 **`test_appc()`** function definition.

then determine if APPC/PC is installed by looking at the memory location 9 bytes prior to the address pointed to by the interrupt vector and looking for the string "APPC/PC."

The actual call to APPC/PC involves filling a structure which is unique to each command, pointing the DS:DX register pair to the beginning of this structure, setting the AH register to the command type, then executing an interrupt 0x68. The function **`int_appc()`** shown in Code Box 7.2 shows how this can be done.

The header file **`appc.h`** (shown below) defines each of the appc commands, each of the appc command structures, and all return codes.

```
/* APPC Commands */
#define ALLOCATE                        0x0001
#define ALLOCATE_FAMILY                 0x02
#define CONFIRM                         0x0003
#define CONFIRM_FAMILY                  0x02
#define CONFIRMED                       0x0004
#define DEALLOCATE                      0x0005
#define DEALLOCATE_FAMILY               0x02
```

```
#include        <dos.h>
#include        "appc.h"

extern  int             net_error;

/****************************************************************
*       int_appc() - execute an appc command block
*
*       Parameters:
*               command_block (in) - pointer to command block
*               command_family (in) - command family to execute
*
*       Global:
*               net_error - updated with primary return code value
*
*       Returns:
*               0 for success, net_error for error
*
*       History:
*               Original code by William H. Roetzheim, 1990
*****************************************************************/

int             int_appc(void *command_block, unsigned char command_family)
{
        char    *cb = command_block;

        _DS = FP_SEG(command_block);
        _DX = FP_OFF(command_block);
        _AH = command_family;
        geninterrupt(0x68);
        net_error = * (int * ) (& cb[20]);
        return net_error;
}
```

Code Box 7.2 `int_appc()` function definition.

```
#define FLUSH                           0x0006
#define FLUSH_FAMILY                    0x02
#define GET_ATTRIBUTES                  0x0007
#define GET_ATTRIBUTES_FAMILY           0x02
#define GET_TYPE                        0x0008
#define GET_TYPE_FAMILY                 0x02
#define POST_ON_RECEIPT                 0x0009
#define POST_ON_RECEIPT_FAMILY          0x02
#define PREPARE_TO_RECEIVE              0x000A
#define PREPARE_TO_RECEIVE_FAMILY       0x02
#define RECEIVE_AND_WAIT                0x000B
#define RECEIVE_AND_WAIT_FAMILY         0x02
#define RECEIVE_IMMEDIATE               0x000C
#define RECEIVE_IMMEDIATE_FAMILY        0x02
#define REQUEST_TO_SEND                 0x000E
#define REQUEST_TO_SEND_FAMILY          0x02
#define SEND_ERROR                      0x0010
#define SEND_ERROR_FAMILY               0x02
#define TEST                            0x0012
#define TEST_FAMILY                     0x02
#define WAIT                            0x0013
#define WAIT_FAMILY                     0x02
#define CNOS                            0x0015
#define CNOS_FAMILY                     0x06
#define ACCES_LU_LU_PW                  0x0019
#define ACCESS_LU_LU_PW_FAMILY          0x02
#define CONVERT                         0x001A
#define CONVERT_FAMILY                  0x251
```

```
#define DISPLAY                         0x001B
#define DISPLAY_FAMILY                  0x01
#define TRANSFER_MS_DATA                0x001C
#define TRANSFER_MS_DATA_FAMILY         0x05
#define ATTACH_PU                       0x0020
#define ATTACH_PU_FAMILY                0x01
#define ATTACH_LU                       0x0021
#define ATTACH_LU_FAMILY                0x01
#define CREATE_TP                       0x0023
#define CREATE_TP_FAMILY                0x01
#define TP_STARTED                      0x0024
#define TP_STARTED_FAMILY               0x03
#define TP_ENDED                        0x0025
#define TP_ENDED_FAMILY                 0x04
#define SYSLOG                          0x0026
#define SYSLOG_FAMILY                   0x01
#define DETACH_PU                       0x0027
#define DETACH_PU_FAMILY                0x01
#define GET_ALLOCATE                    0x0028
#define GET_ALLOCATE_FAMILY             0x03
#define TP_VALID                        0x0029
#define TP_VALID_FAMILY                 0x04
#define CHANGE_LU                       0x002A
#define CHANGE_LU_FAMILY                0x03
#define ACTIVATE_DLC                    0x002B
#define ACTIVATE_DLC_FAMILY             0x01

typedef unsigned int    RWORD;  /* byte reversed integer */
typedef         unsigned long   RLONG;  /* byte reversed long */

/* structure definitions for APPC/Commands */
struct  access_lu_lu_pw
{
        char                    reserved[12];
        unsigned int            command;
        char                    lu_id[8];
        char                    lu_name[8];
        char                    partner_lu_name[8];
        char                    partner_lu_fully_qualified_lu_name[17];
        char                    password_available;
        char                    password[8];
};

struct  activate_dlc
{
        char                    reserved1[12];
        unsigned int            command;
        char                    reserved2[6];
        RLONG                   return_code;
        char                    dlc_name[8];
        char                    adapter_number;
};

struct  allocate
{
        char                    reserved1[12];
        unsigned int            command;
        char                    verb_extension_code;
        char                    reserved2[5];
        RWORD                   primary_return_code;
        RLONG                   secondary_return_code;
        char                    tp_id[8];
        RLONG                   conv_id;
        char                    conversation_type;
        char                    sync_level;
        char                    reserved3[2];
        char                    return_control;
        char                    reserved4[8];
        char                    partner_lu_name[8];
        char                    mode_name[8];
        unsigned char           tp_name_length;
        char                    tp_name[64];
        char                    security;
        char                    reserved5[11];
        unsigned char           password_length;
        char                    password[10];
        unsigned char           user_id_length;
        char                    user_id[10];
        unsigned int            pip_data_length;
        far                     *pip_data;
};


struct  attach_lu
{
        char                    reserved1[12];
        unsigned int            command;
        char                    reserved2[6];
        RLONG                   return_code;
        unsigned int            offset_to_partner_lu_record_length_field;
        char                    lu_name[8];
        char                    lu_id[8];
        unsigned char           lu_local_address;
        unsigned char           lu_session_limit;
        far                     *create_tp_exit;
        char                    reserved3[4];
        far                     *system_log_exit;
        char                    reserved4[4];
        unsigned char           max_tps;
        unsigned char           queue_dpeth;
        far                     *lu_lu_password_exit;
        char                    *reserved5[4];
        unsigned int            total_length_of_partner_lu_records;

        struct
        {
                unsigned int    length_of_this_partner_lu_record;
                unsigned int    offset_to_start_of_mode_records;
                char            partner_lu_name[8];
                unsigned char   partner_lu_security_capabilities;
                unsigned char   partner_lu_session_limit;
                unsigned int    partner_lu_max_mc_send_ll;
                char            partner_lu_dlc_name[8];
                unsigned char   partner_lu_adapter_number;
                unsigned char   length_of_partner_lu_address;
                char            partner_lu_adapter_address[16];
                unsigned int    total_length_of_all_mode_name_records;
                struct
                {
                        unsigned int            length_of_this_mode_name_record;
                        char                    mode_name[8];
                        unsigned int            ru_size_high_bound;
                        unsigned int            ru_size_low_bound;
                        unsigned char           mode_max_negotiable_session_limit;
                        unsigned int            pacing_size;
                } modename [MAX_MODE];
        } partner_lu [MAX_PARTNER_LU];
};
```

```
struct  attach_pu
{
        char                    reserved1[12];
        unsigned int            command;
        char                    reserved2[6];
        RLONG                   return_code;
        char                    reserved3[2];
        unsigned char           version;
        unsigned char           release;
        char                    net_name[8];
        char                    pu_name[8];
        char                    reserved4[8];
        far                     *system_log_exit;
        char                    reserved5[4];
        unsigned char           return_control;
};


struct change_lu
{
        char                    reserved1[12];
        unsigned int            command;
        char                    reserved2[6];
        RLONG                   return_code;
        char                    reserved3[2];
        far                     *lu_id_create_tp_exit;
        char                    reserved4[4];
        far                     *system_log_exit;
        char                    reserved5[4];
        unsigned char           max_tps;
        unsigned char           queue_allocates;
        far                     *lu_lu_password_exit;
        char                    reserved6[4];
};


struct cnos
{
        char                    reserved1[12];
        unsigned int            command;
        char                    reserved2[6];
        RWORD                   primary_return_code;
        RLONG                   secondary_return_code;
        char                    lu_id[8];
        char                    reserved3[8];
        char                    partner_lu_name[8];
        char                    mode_name[8];
        unsigned char           mode_name_select;
        unsigned char           partner_lu_mode_session_limit;
        unsigned char           min_conwinners_source;
        unsigned char           min_conwinners_target;
        unsigned char           auto_activate;
        char                    reserved4;
        unsigned char           termination_settings;
};


struct  confirm
{
        char                    reserved1[12];
        unsigned int            command;
        unsigned char           verb_extension_code;
        char                    reserved2[5];
        RWORD                   primary_return_code;
        RLONG                   secondary_return_code;
        char                    tp_id[8];
        RLONG                   conv_id;
        unsigned char           request_to_send_received;
};


struct confirmed
{
        char                    reserved1[12];
        unsigned int            command;
        unsigned char           verb_extension_code;
        char                    reserved2[5];
        RWORD                   primary_return_code;
        RLONG                   secondary_return_code;
        char                    tp_id[8];
        RLONG                   conv_id;
};


struct convert
{
        char                    reserved1[12];
        unsigned int            command;
        char                    reserved2[6];
        RLONG                   return_code;
        unsigned char           direction;
        unsigned char           character_set;
        unsigned int            length;
        char                    *source;
        char                    *target;
};


struct create_tp
{
        char                    reserved1[12];
        unsigned int            command;
        char                    reserved2[6];
        RLONG                   sense_code;
        char                    tp_id[8];
        char                    lu_id[8];
        RLONG                   conv_id;
        unsigned char           type;
        unsigned char           sync_level;
        char                    reserved3;
        unsigned char           transaction_program_name_length;
        char                    tpn[64];
        char                    reserved4[6];
        unsigned int            length_of_error_log_data_to_return;
        far                     *pointer_to_error_log_data_to_return;
        char                    partner_lu_name[8];
        unsigned int            length_of_fully_qualified_partner_lu_name;
        char                    partner_fully_qualified_lu_name[17];
        char                    mode_name[8];
        char                    reserved5[12];
        unsigned char           length_of_password;
        char                    password[10];
        unsigned char           length_of_user_id;
        char                    user_id;
        unsigned char           already_verified;
};
```

```
struct deallocate
{
        char                    reserved1[12];
        unsigned int            command;
        unsigned char           verb_extension_code;
        char                    reserved2[5];
        RWORD                   primary_return_code;
        RLONG                   secondary_return_code;
        char                    tp_id[8];
        RLONG                   conv_id;
        char                    reserved3;
        unsigned char           type;
        unsigned int            length_of_error_log_data;
        far                     *address_of_error_log_data;
};


struct  detach_lu
{
        char                    reserved1[12];
        unsigned int            command;
        char                    reserved2[6];
        RLONG                   sense_code;
        char                    lu_id[8];
        char                    reserved3;
};


struct  detach_pu
{
        char                    reserved1[12];
        unsigned int            command;
        char                    reserved2[6];
        RLONG                   sense_code;
        char                    type;
};


struct  display
{
        char                    reserved1[12];
        unsigned int            command;
        char                    reserved2[6];
        RLONG                   sense_code;
        char                    reserved3[2];
        char                    lu_id[8];
        char                    partner_lu_name[8];
        char                    mode_name[8];
        unsigned char           lu_session_limit;
        unsigned char           mode_max_negotiable_session_limit;
        unsigned char           current_session_limit;
        unsigned char           min_negotiated_winner_limit;
        unsigned char           min_negotiated_loser_limit;
        unsigned char           active_session_count;
        unsigned char           active_conwinner_session_count;
        unsigned char           active_conloser_session_count;
        unsigned char           session_termination_count;
        unsigned char           termination_settings;
};


struct  flush
{
        char                    reserved1[12];
        unsigned int            command;
        unsigned char           verb_extension_code;
        char                    reserved2[5];
        RWORD                   primary_return_code;
        RLONG                   secondary_return_code;
        char                    tp_id[8];
        RLONG                   conv_id;
};


struct  get_allocate
{
        char                    reserved1[12];
        unsigned int            command;
        char                    reserved2[6];
        RLONG                   sense_code;
        char                    reserved3[2];
        char                    lu_id[8];
        unsigned char           type;
        far                     *pointer_to_create_tp_record;
};


struct  get_attributes
{
        char                    reserved1[12];
        unsigned int            command;
        unsigned char           verb_extension_code;
        char                    reserved2[5];
        RWORD                   primary_return_code;
        RLONG                   secondary_return_code;
        char                    tp_id[8];
        RLONG                   conv_id;
        char                    lu_id;
        char                    reserved3;
        unsigned char           sync_level;
        char                    mode_name[8];
        char                    own_net_name[8];
        char                    own_lu_name[8];
        char                    partner_lu_name[8];
        unsigned char           length_of_partner_fully_qualified_lu_name;
        char                    partner_fully_qualified_lu_name[17];
        char                    reserved4;
        unsigned char           length_of_user_id;
        char                    user_id[10];
};


struct  get_type
{
        char                    reserved1[12];
        unsigned int            command;
        char                    reserved2[6];
        RWORD                   primary_return_code;
        RLONG                   secondary_return_code;
        char                    tp_id[8];
        RLONG                   conv_id;
        char                    type;
};


struct  post_on_receipt
{
        char                    reserved1[12];
        unsigned int            command;
        char                    reserved2[6];
        RWORD                   primary_return_code;
        RLONG                   secondary_return_code;
        char                    tp_id[8];
        RLONG                   conv_id;
        unsigned int            max_length;
        unsigned char           fill;
};

struct  prepare_to_receive
{
        char                    reserved1[12];
        unsigned int            command;
        unsigned char           verb_extension_code;
        char                    reserved2[5];
        RWORD                   primary_return_code;
        RLONG                   secondary_return_code;
        char                    tp_id[8];
        RLONG                   conv_id;
        unsigned char           type;
        unsigned char           locks;
};

struct  receive_and_wait
{
        char                    reserved1[12];
        unsigned int            command;
        unsigned char           verb_extension_code;
        char                    reserved2[5];
        RWORD                   primary_return_code;
        RLONG                   secondary_return_code;
        char                    tp_id[8];
        RLONG                   conv_id;
        unsigned char           what_received;
        unsigned char           fill;
        unsigned char           request_to_send_received;
        unsigned int            max_length;
        unsigned int            data_length;
        far                     *data_ptr;
};

struct receive_immediate
{
        char                    reserved1[12];
        unsigned int            command;
        unsigned char           verb_extension_code;
        char                    reserved2[5];
        RWORD                   primary_return_code;
        RLONG                   secondary_return_code;
        char                    tp_id[8];
        RLONG                   conv_id;
        unsigned char           what_received;
        unsigned char           fill;
        unsigned char           request_to_send_received;
        unsigned int            max_length;
        unsigned int            data_length;
        far                     *data_ptr;
};
```

```
struct request_to_send
{
        char                        reserved1[12];
        unsigned int                command;
        unsigned char               verb_extension_code;
        char                        reserved2[5];
        RWORD                       primary_return_code;
        RLONG                       secondary_return_code;
        char                        tp_id[8];
        RLONG                       conv_id;
};


struct send_data
{
        char                        reserved1[12];
        unsigned int                command;
        unsigned char               verb_extension_code;
        char                        reserved2[5];
        RWORD                       primary_return_code;
        RLONG                       secondary_return_code;
        char                        tp_id[8];
        RLONG                       conv_id;
        unsigned char               request_to_send_received;
        char                        reserved3;
        unsigned int                data_length;
        far                         *data_ptr;
};


struct  send_error
{
        char                        reserved1[12];
        unsigned int                command;
        unsigned char               verb_extension_code;
        char                        reserved2[5];
        RWORD                       primary_return_code;
        RLONG                       secondary_return_code;
        char                        tp_id[8];
        RLONG                       conv_id;
        unsigned char               request_to_send_received;
        unsigned char               type;
        char                        reserved3[4];
        unsigned int                log_data_length;
        far                         *log_data;
};


struct tp_ended
{
        char                        reserved1[12];
        unsigned int                command;
        char                        reserved2[6];
        RLONG                       return_code;
        char                        reserved3[2];
        char                        tp_id[8];
};


struct tp_started
{
        char                        reserved1[12];
        unsigned int                command;
        char                        reserved2[6];
        RLONG                       return_code;
        char                    reserved3[2];
        char                    lu_id[8];
        char                    tp_id[8];
};


struct tp_valid
{
        char                    reserved1[12];
        unsigned int            command;
        char                    reserved2[6];
        RLONG                   return_code;
        char                    reserved3[2];
        char                    tp_id[8];
        far                             *create_tp_ptr;
};


/* function prototypes */
char    *attach_lu(char *local_lu, char *remote_lu[], char *remote_address[]);
int     test_appc (void);
int     int_appc(void *command_block, unsigned char command_family);
char    *ascii_to_ebcdic (char *ascii);
```

7.4 Sending a Transaction in APPC

To establish an outgoing connection in APPC/PC, five APPC/PC commands are used:

1. **`attach_pu()`** defines a local APPC physical unit (PU).

2. **`attach_lu()`** defines a local APPC logical unit (LU) for your use. Although a given PC will normally only have one PU, it is possible for the same computer to have multiple LUs. For example, if you wish to establish full-duplex communication with a remote host, you would normally use two LUs (one for send, one for receive).

3. **`cnos()`**, which stands for *change number of sessions*, is used to establish or teardown an actual connection. Establishing a connection involves raising the session limit for a given partner LU from 0 to 1 and telling if the connection should be a transmit connection or a

receive connection. Other parameters (not discussed) come into play if you have multiple simultaneous sessions and you want negotiations to be performed between APPC/PC nodes for network resources.

4. **`activate_dlc()`** is used to activate APPC over the token ring network adapter. If you were also using an SDLC adapter, you would need an additional call to **`activate_dlc()`** to activate the SDLC adapter.

5. **`allocate()`** is used to reserve buffers and prepare the connection for use by the application.

Code Box 7.3 shows sample code for **`attach_pu()`**. This code is included for illustrative purposes only to demonstrate how typical APPC/PC commands are executed. The end result of this entire procedure will be a valid **`tp_id`** (transaction program ID) and **`conv_id`** (conversation ID). The transaction program ID is an 8-character string (blank padded), while the conversation ID is a 4-character string (blank padded). Both strings are EBCDIC (not ASCII), and neither is null terminated.

For some commands, you may wish to modify a parameter which is defined as a RWORD or RLONG. These parameters are unsigned integer or long integer variables *where they byte (and word) order is reversed from that used in the IBM-PC*. For example, you may have noticed that all the return values passed back from APPC/PC are returned in byte (and for longs, word) reversed order. You may wish to modify **`int_appc()`** to reverse the return values prior to storing the value in **`net_error`**.

After you have established a connection (called a conversation), actually sending the transaction is relatively simple. You use the **`req_send`** command to request permission to send data, the send the data using **`send_data`**. The data is EBCDIC and consists of two

```
#include        <string.h>
#include        "appc.h"

extern  int             net_error;

/*******************************************************************
*       attach_pu - attach physical unit to network
*
*       Parameters:
*               network (in) - network name, 8 characters w/blank padding
*
*       Returns:
*               0 for success, net_error for failure
*
*       History:
*               Original code by William H. Roetzheim, 1990
********************************************************************/

int     attach_pu(char *network)
{
        struct  attach_pu       apu;

        memset (&apu, 0, sizeof (struct attach_pu));
        apu.command = ATTACH_PU;
        memcpy (apu.net_name,ascii_to_ebcdic (network), 8);
        memcpy (apu.pu_name, apu.net_name, 8);
        apu.system_log_exit = (void *) 0xFFFFFFFF;      /* none */
        apu.return_control = 0x00;                      /* complete */
        int_appc(&apu, ATTACH_PU_FAMILY);
        return net_error;
}
```

Code Box 7.3 attach_pu() function definition.

bytes specifying the length (in reversed order from the IBM-PC) followed by the actual data in any format you desire.

Part of establishing the conversation involves specifying a transaction program name. This name is specified in EBCDIC and is up to 64 characters long. When the receiving computer receives a transaction, it uses this name to load the program specified by this transaction program name. For example, if the transaction program name was `command.com` the computer would look for and load command.com when the conversation was established; The data passed as an actual transaction is then given as input to the newly loaded program.

When you are done, the conversation must be terminated. This involves freeing up resources (`deallocate`), calling `cnos()` to change the number of sessions from 1 to 0, and calling **detach_lu()** and **detach_pu()** to terminate the conversation.

7.5 Receiving Transactions Using APPC/PC

To a large extent, the receiving program's tasks are the mirror image of the sending program's tasks. You still must use the functions **attach_pu, attach_lu, cnos,** and **activate_dlc** to establish a connection. In this case, the **partner_lu** related fields might include multiple partners, one for each remote system from which you may be receiving transactions. When transactions arrive, they can either be queued or you can be notified of their arrival via an interrupt. A structure is available to you which includes the name of the transaction program (in EBCDIC) which should be executed. If the transaction program requested exists, you issue a **tp_valid** to confirm the conversation to the remote computer. You then use the **receive_and_wait** or **receive_immediate** command to receive the actual data for the transaction program. You then use the **tp_ended** command to tell APPC/PC that the program is done, calling **cnos()** to change the number of sessions from 1 to 0 and calling **detach_lu()** and **detach_pu()** to terminate the conversation.

7.6 Summary of APPC/PC Commands

The following table presents a summary of all APPC/PC commands. The columns have the following meanings:

1. **Command** – The command name. These names are defined in **appc.h**. These are the values to use for the command structure's **command** field prior to calling APPC for processing. The command family (to be placed in register AH) is also defined in APPC.h.

2. **Inputs** – The fields within the command structure which are used as input.

3. **Outputs** – The fields within the command structure which are modified by the command during processing.

4. **Summary** – A brief description of the command function.

Command	Inputs	Outputs	Summary
ACCESS_LU_LU_PW	command lu_id lu_name partner_lu_name partner_fully_qualified_lu-_name	password_available password	Request (by APPC/PC) for you to provide a password for a specified partner LU.
ACTIVATE_DLC	dlc_name adapter_number	return_code	Activates a DLC adapter. This command must be issued for each DLC adapter installed. DLC adapters include the Token Ring Adapter and the SDLC adapter.
ALLOCATE	command verb_extension_code tp_id conversation_type sync_level return_control partner_lu_name mode_name tp_name_length tp_name security user_id_length user_id pip_data_length pip_data	primary_return_code secondary_return_code conv_id	Allocates a session between the local LU and a remote LU and identifies the remote transaction program the local LU wishes to talk to.

ATTACH_LU	command offset_to_partner_lu_re- cord_length_field lu_name lu_local_address lu_session_limit create_tp_exit system_log_exit max_tps queue_depth lu_lu_password_exit total_length_of_partne- r_lu_records length_of_this_partner- _lu_record offset_to_start_of_mod- e_records partner_lu_name partner_lu_security_cap- abilities partner_lu_session_limit partner_lu_max_mc_sen- d_ll partner_lu_dlc_name partner_lu_adapter_num- ber length_of_partner_lu_ada- pter_address partner_lu_adapter_add- ress total_length_of_all_mo- de_name_records length_of_this_mode_- name_record mode_name ru_size_high_bound ru_size_low_bound mode_max_negotiable_- session_limit pacing_size	return_code lu_id	Creates a local LU with the specified parameters.
ATTACH_PU	command net_name pu_name system_log_exit return_control	return_code version release	Defines a local physical unit with the specified parameters.

CHANGE_LU	command lu_id create_tp_exit system_log_exit max_tps queue_allocates lu_lu_password_exit	return_code	Alters specified parameters for an existing local LU.
CNOS	command lu_id partner_lu_name mode_name mode_name_select partner_lu_mode_session_limit min_conwinners_source min_conwinners_target auto_activate termination_settings	primary_return_code secondary_return_code	Establishes a session for a given LU to LU conversation.
CONFIRM	command verb_extension_code tp_id conv_id	primary_return_code secondary_return_code request_to_send_received	Sends a request for confirmation to a remote transaction program and waits for a reply.
CONFIRMED	command verb_extension_code tp_id conv_id	primary_return_code secondary_return_code	Sends a confirmation reply to a remote transaction program (partner).
CONVERT	command direction character_set length source target	return_code	Converts between ASCII and EBCDIC.
DEALLOCATE	command verb_extension_code tp_id conv_id type length_of_error_log_data log_data	primary_return_code secondary_return_code	Terminates the specified conversation.
DETACH_LU	command lu_id	return_code	Terminates a local LU.
DETACH_PU	command type	return_code	Terminates the local PU.

DISPLAY	command lu_id partner_lu_name mode_name	return_code lu_session_limit partner_lu_session_limit mode_max_negotiable-_session_limit current_session_limit min_negottiated_winn-er_limit min_negotiated_loser_-limit active_session_count active_conwinner_sess-ion_count active_conloser_sessio-n_count session_termination_count termination_settings	Returns the current parameters associated with a local LU.
FLUSH	command verb_extension_code tp_id conv_id	primary_return_code secondary_return_code	Flushes the send buffer for the local LU.
GET_ALLOCATE	command lu_id type	return_code pointer_to_create_tp_-record	Returns the next incomming allocate request which has been queued.
GET_ATTRIBUTES	command verb_extension_code tp_id conv_id	primary_return_code secondary_return_code lu_id sync_level mode_name own_net_name own_lu_name partner_lu_name partner_fully_qualified_lu-_name length_of_user_id user_id	Returns parameters describing a specified conversation.
GET_TYPE	command tp_id conv_id	return_code type	Tells you whether a conversation is basic or mapped.
POST_ON_RECEIPT	command verb_extension_code tp_id conv_id max_length fill	primary_return_code secondary_return_code	During a conversation, instructs APPC/PC to interrupt you when it receives the next incomming data buffer.

PREPARE_TO_RECEIVE	command verb_extension_code tp_id conv_id type locks	primary_return_code secondary_return_code	Changes a basic conversation from the send state to the receive state.
RECEIVE_AND_WAIT	command verb_extension_code tp_id conv_id fill max_length data_ptr	primary_return_code secondary_return_code what_received request_to_send_received data_length	Waits for data to arrive for a specified conversation, then places the incomming data into the application designated buffer area.
RECEIVE_IMMEDIATE	command verb_extension_code tp_id conv_id fill max_length	primary_return_code secondary_return_code what_received request_to_send_received data_length	If information is available, receives the information. If no information is available, returns.
REQUEST_TO_SEND	command verb_extension_code tp_id conv_id	primary_return_code secondary_return_code	Tells the partner LU that the local LU wishes to enter a send state.
SEND_DATA	command verb_extension_code tp_id conv_id data_length data_ptr	primary_return_code secondary_return_code request_to_send_received	Sends one data record to a partner LU.
SEND_ERROR	command verb_extension_code tp_id conv_id type log_data_length log_data	primary_return_code secondary_return_code request_to_send_received	Tells the partner LU that an error was detected.
TEST	command verb_extension_code tp_id conv_id test	primary_return_code secondary_return_code	Tests the specified conversation to determine if the conversation has been posted or if a request_to_send has been received.

TP_ENDED	command tp_id	return_code	Tells APPC/PC that the specified transaction program has exited (terminated).
TP_STARTED	command lu_id	return_code tp_id	Tells APPC/PC that a transaction program has successfully been started and requests APPC/PC to assign a tp_id to the new proram. Programs automatically started as a result of an incoming allocate do not require a call to tp_started.
TP_VALID	command tp_id create_tp_ptr	return_code	Tells APPC/PC that the program named in an incoming allocate exists and is valid.
TRANSFER_MS_DATA	command data_type verb_options data_length data	return_code	Transfers network management information between nodes.

7.7 Primary Return Codes

Code	Name	Description	Action
0x0000	OK	Command completed normally.	None.
0x0001	PARAMETER_CHECK	A parameter is bad in the command structure.	Check the secondary return code for the specific parameter which is bad.
0x0002	STATE_CHECK	Attempt to increase the session limit (without starting at zero).	The session limit must first be set to zero, then modified.
0x0003	ALLOCATION_ERROR	Conversation could not be allocated.	Check secondary return code for reason.

0x0005	DEALLOCATE_ABEND	The remote transaction program cancelled the conversation unexpectedly.	Check the transaction program for errors.
0x0006	DEALLOCATE_ABEN-D_PROG	The remote transaction program cancelled the conversationunexpectedly and set the deallocate flag to "prog".	Check the transaction program for errors.
0x0007	DEALLOCATE_ABEN-D_SVC	The remote transaction program cancelled the conversationunexpectedly and set the deallocate flag to "svc".	Check the transaction program for errors.
0x0008	DEALLOCATE_ABEN-D_TIMER	The remote transaction program cancelled the conversationunexpectedly and set the deallocate flag to "timer".	Check the transaction program for errors.
0x0009	DEALLOCATE_NORM-AL	The conversation was terminated normally.	None.
0x0014	UNSUCCESSFUL	The program specified return control immediate but APPC was not able to allocate the conversation because no sessions were available.	Re-issue using return control when session activated.
0x000A	DATA_POSTING_BL-OCKED	The APPC internal space is full.	Issue a receive_immediate or receive_and_wait to empty some APPC buffers.
0x000B	POSTING_NOT_ACT-IVE	Posting is not active for the specified conversation and you tested the conversation.	Issue post_on_receipt before testing the conversation.
0x000C	PROG_ERROR_NO-_TRUNC	The remote program detected a transmission error but no logical record was affected.	Check the secondary_-return_code for the error type.

0x000E	PROG_ERROR_PURG-ING	The remote program detected a transmission error and logical records were affected (and purged).	Check the error and prepare to re-transmit the data.
0x000D	PROG_ERROR_TRUNC	The remote program detected a transmission error and logical records were truncated.	Check the error.
0x000F	CONV_FAILURE_RE-TRY	A temporary failure caused an abnormal termination.	Establish the connection again.
0x0010	CONV_FAILURE_NO-_RETRY	Conversation was abnormally terminated.	Normally indicates a hardware problem.
0x0011	SVC_ERROR_NO_TR-UNC	The remote program issued a svc error but did not truncate any logical records.	Check the secondary_-return_code for the error type.
0x0012	SVC_ERROR_TRUNC	The remote program detected a transmission error and logical records were truncated.	Check the error.
0x0013	SVC_ERROR_PURG-ING	The remote program issued a svc error and is purging one or more logical records which were received in error.	Check the error and prepare to retransmit the data.
0x0014	UNSUCCESSFUL	There is nothing to receive.	None.
0x0018	CNOS_PARTNER_REJ-ECT	The partner LU rejected the CNOS request.	Check the secondary return code for the reason.
0x0019	CONVERSATION_TY-PE_MIXED	Use of both basic and mapped commands in one conversation.Issue only one type of verb.	
0xF004	INCOMPLETE	The issued command was suspended without completing.	To avoid deadlock, issue verbs on any other transaction programs desired, then issue get_allocate to empty the incoming queue, then re-issue this

			command without cnanging any parameter.
0xF005	INCOMPLETE_ALTERED_VERB	A command that was returned as incomplete was changed and re-issued (or you issued a new command to a transaction program with an incomplete command outstanding).	Modify your code to properly handle incomplete commands.
0xFFFF	INVALID_VERB	The command code is wrong.	Check the command code and the command family (in register AH).
0x0012	SVC_ERROR_TRUNC	The remote program detected a transmission error and logical records were truncated.	Check the error.

7.8 Suggested Reading

IBM (1987), *APPC/PC User Application Interface*, Document number GG24-3025-0, Boca Raton, FL: International Business Machine Corporation.

IBM (1986), *Advanced Program-to-Program Communication for the IBM Personal Computer Programming Guide* - 2nd edition, Raleigh, NC: IBM product 84X0561.

IBM (1986), *An Introduction to Programming for APPC/PC*, Document number GG24-3034, Raleigh, NC: International Business Machine Corporation.

Appendix A
Glossary

Advanced Program-to-Program Communication A set of protocols that provides communication capabilities between computer programs, often on diverse hardware.

alias An alternate name that you can be known by on the network.

alignment error The number of frames received with excessive or missing bits causing a CRC error.

application program interface The set of commands used by the application program to communicate with a lower level process in general (or APPC in particular in this book).

bind Establish an LU 6.2 session.

buffer Memory area temporarily reserved for use in performing input/output operations.

collisions When a transmitting adapter detects any type of line noise during transmission of a frame, the adapter stops transmitting and registers a collision.

contention loser In APPC, the LU that must request and receive permission from the session partner LU to allocate a session.

contention-loser polarity The designation that an LU is the contention loser for a session.

contention winner The LU that can allocate a session without requesting permission from the session partner LU.

contention-winner polarity The designation that an LU is the contention winner for a session.

control verb Commands an application subsystem issues under APPC to set up the hardware and software to perform a remote transaction.

conversation The communication between two transaction programs under APPC.

conversation type Under APPC, either basic or mapped.

cyclic redundancy check An error detection algorithm using a cyclic algorithm.

datagram A single data packet delivered with best effort. No retransmissions or automatic resequencing of multiple packets is performed.

deadlock A situation in which two or more processes are waiting for resources held by each other which will never become available. In APPC, deadlock can occur when using transaction processing commands.

duplex Simultaneous two-way independent transmission.

exhausted resources The number of frames discarded because of a lack of memory.

flow control The process of managaging the rate at which data packets or transactions are sent and received.

frame A low level packet of information which is used to implement all higher level protocols.

hot carrier A transmitter locked in transmit mode.

local session number A unique number assigned to each session established by an adapter.

logical unit A set of logical services allowing one user to communicate with each other using sessions.

pacing window size In APPC, the number of RUs that a program can send before getting permission to send more.

point-to-point A connection between exactly two nodes on a network.

virtual connection A transport layer connection between two network nodes that supports reliable data communication.

Appendix B
Acronyms

APPC Advanced Program-to-Program Communication

API Application Program Interface

ASCII American National Standard Code for Information Exchange

BIOS Basic Input Output System

CICS Customer Information Control System

CICS/VS Customer Information Control System for Virtual Storage

CRC Cyclic Redundancy Check

DLC Data Link Controld

EBCDIC Extended Binary-Coded Decimal Interchange Code

FMH	Function Management Header
GDS	General Data Stream
LU	Logical Unit
NAU	Network Addressable Unit
NCB	Network Control Block
NetBIOS	Network Basic Input Output System
NMVT	Network Management Vector Transport
PIP	Program Initialization Parameter
PLU	Primary Logical Unit
PU	Physical Unit
RAM	Random Access Memory
RH	Request/response Header
ROM	Read Only Memory
RU	Request/response Unit
SDLC	Synchronous Data Link Control
SLU	Secondary Logical Unit

SNA	Systems Network Architecture
SSCP	System Services Control Point
SDLC	Synchronous Data Link Control
VTAM	Virtual Terminal Access Method

Appendix C References

Carlo, J. T., and G. R. Samsen (1986), "High-Level Communication Protocols on the Token-Ring Network," *Proceedings of the Localnet '86 Conference*, November 18-20.

East, W. (1988), "New Developments Lead to Further Integration of a High Performance Token Ring Adapter," *Proceedings of the Networking Technology and Architectures* (Pinner, UK: Blenheim Online), pp. 89-105.

Housley, N. (1987), "An IBM Token Ring Backbone Facility," *Networks 87, Proceedings of the European Computer Communications Conference* (Pinner, UK: Online International).

IBM (1988), *Local Area Network Technical Reference*, Research Triangle Park, NC: International Business Machine Corporation.

Lang, K. W. et. al. (1989), "A 16 MBPS Adapter Chip for the IBM Token-Ring Local Area Network," *Proceedings of the IEEE 1989 Custom Integrated Circuits Conference*, May 15-18.

Lank, K. (1989), "A 16 MBPS Adapter Chip for the IBM Token Ring Local Area Network," *Proceedings of the IEEE 1989 Custom Integrated Circuits Conference*, May (New York: IEEE), pp. 11.3.1-11.3.5

Stallings, W. (1987), *Handbook of Computer Communications Standards (Vol. 2): Local Network Standards*, New York: Macmillan.

Stallings, W. (1984), *Local Networks: An Introduction*, New York: Macmillan.

Strole, N. (1989), "Inside Token Ring Version II, according to Big Blue," *Data Communications*, Vol. 18, no. 1, (January) pp. 117-125.

Tanimoto, M., et. al. (1987), "Development of a High Speed Fiber Optic LAN to Connect Heterogeneous Computers in the Department of Information Engineering at Kyoto University," *Sumitomo Electronics Technical Review*, (January) Japan, pp. 131-136.

Index

A C Programmer's Guide to the IBM Token Ring

William H. Roetzheim

This book distills the key elements from the IBM Token Ring Network reference documents and presents it in an easy-to-understand, concise fashion. For most programmers, everything you will ever need to know about the IBM Token Ring Network can now be found in this convenient volume. Programming for the Token Ring environment is covered at the BIOS redirector, NetBIOS, DLC, register direct, and APPC level. Token Ring hardware is described, with a particular emphasis on the interaction between the hardware and your application programs. Dozens of tables and charts provide a convenient reference to all interrupts, functions, and return codes. Each concept is illustrated with complete C functions which serve both as examples and form the basis of a working library to be used over and over. For advanced users, detailed and highly specific references are included to simplify the search for additional details.

PRENTICE HALL, Englewood Cliffs, N.J. 07632

ISBN 0-13-723768-5

90000>

9 780137 237685