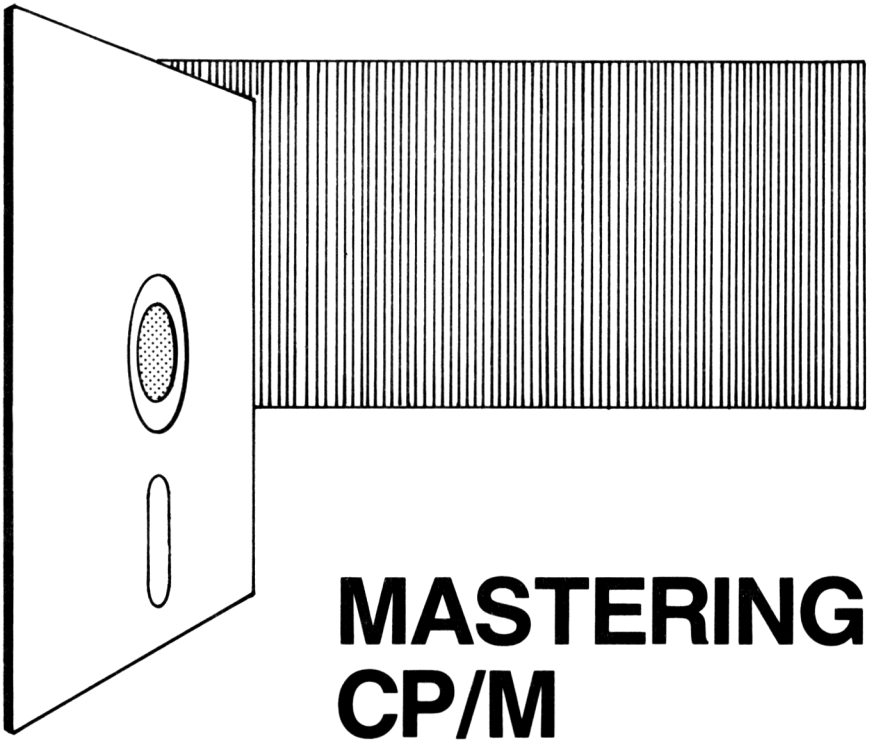
MASTERING CP/M

Alan R. Miller

SYBEX

MASTERING CP/M



MASTERING CP/M

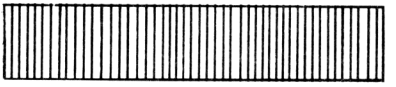
ALAN R. MILLER

SYBEX®

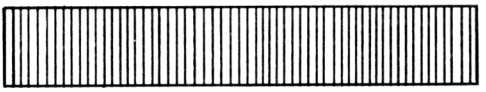
Berkeley • Paris • Düsseldorf

Cover design by Daniel Le Noury
Technical illustrations, book design, and layout by Marlyn Amann

Library of Congress Catalog Card Number: 82-62006
ISBN 0-89588-068-7
Printed in the United States of America
10 9 8 7 6 5 4



CONTENTS



PREFACE

CP/M has become the standard operating system for Z80, 8080, and 8085 microcomputers. As a consequence, there are a large number of programs that run under CP/M. These include assemblers, editors, spelling checkers, compilers for the engineering languages BASIC, Pascal, FORTRAN, and APL, as well as general business packages.

Some CP/M programs can be run automatically so that only a minimal knowledge of CP/M is necessary. However, other programs require a greater understanding of the operating system. In either case, certain routine tasks, such as formatting new disks and making backup copies of important disks, require a working knowledge of the operating system. Unfortunately, it is difficult to learn the operation of CP/M from the documentation that is provided. There are introductory books on the subject,* but these do not discuss the inner workings of CP/M in great detail. Furthermore, there are numerous inconsistencies and idiosyncracies in the operation of CP/M that are waiting to trap the unwary programmer.

---

*See R. Zaks, *The CP/M Handbook with MP/M,* Berkeley: Sybex, 1980.

I have been working with CP/M from its inception (version 1.3). Consequently, I have developed many techniques for improving its usefulness by altering parts of CP/M itself and by writing auxiliary assembly language programs. This book describes what I have learned. It is a guide for the person who wants a deeper knowledge of the inner workings of CP/M.

Although the operation of each program is described, the reader should have some prior experience with 8080 assembly language programming. Further information on assembly language programming can be found in *8080/Z80 Assembly Language* and *Programming the Z80.** To gain the fullest benefit of the book, it will be necessary to have a computer with CP/M, a system editor, a macro assembler such as MAC or MACRO-80, and an assembly language debugger such as SID or DDT.

The book begins with a detailed description of the organization and operation of CP/M. The topics include the system parameter area, TPA, CCP, BDOS, and BIOS. Use of the built-in commands, control characters, and transient programs is also covered. Routine tasks such as formatting new disks and making backup copies are discussed in Chapter 2. The operation of COPY, SYSGEN, and SAVEUSER are also considered, leading to the discussion of procedures for altering the CP/M system and saving the altered version on disk. In Chapter 3 we actually alter the BIOS to incorporate the IOBYTE feature.

The powerful concept of macros is introduced in Chapter 4. Macros for comparing, moving, and filling regions of memory are the foundation of a macro library. The use of BDOS for performing console input and output is implemented with macros in Chapter 5. Several executable programs are written.

Chapters 6 and 7 describe the CP/M disk file system. The macro library is expanded with BDOS operations for reading and writing disk files, and additional executable programs are written. The final chapter presents the details of the CP/M disk directory. A general utility program is written that can be used to display the disk parameters, a block allocation map, and a detailed presentation of the directory.

The appendices contain all the reference material needed to write 8080 and Z80 assembly language programs. Appendix A identifies the ASCII codes in decimal, hexadecimal, and octal. Appendix B presents a 64K-byte memory map. Appendices C and D summarize the 8080 instruction set alphabetically and numerically, respectively. Appendices E and F

*A. R. Miller, *8080/Z80 Assembly Language: Techniques for Improved Programming,* New York: Wiley, 1981.
R. Zaks, *Programming the Z80,* Berkeley: Sybex, 1980.

give the entire Z80 instruction set according to the official Zilog mnemonic, with E being ordered alphabetically and F numerically. Those instructions common to the 8080 set are marked with an asterisk.

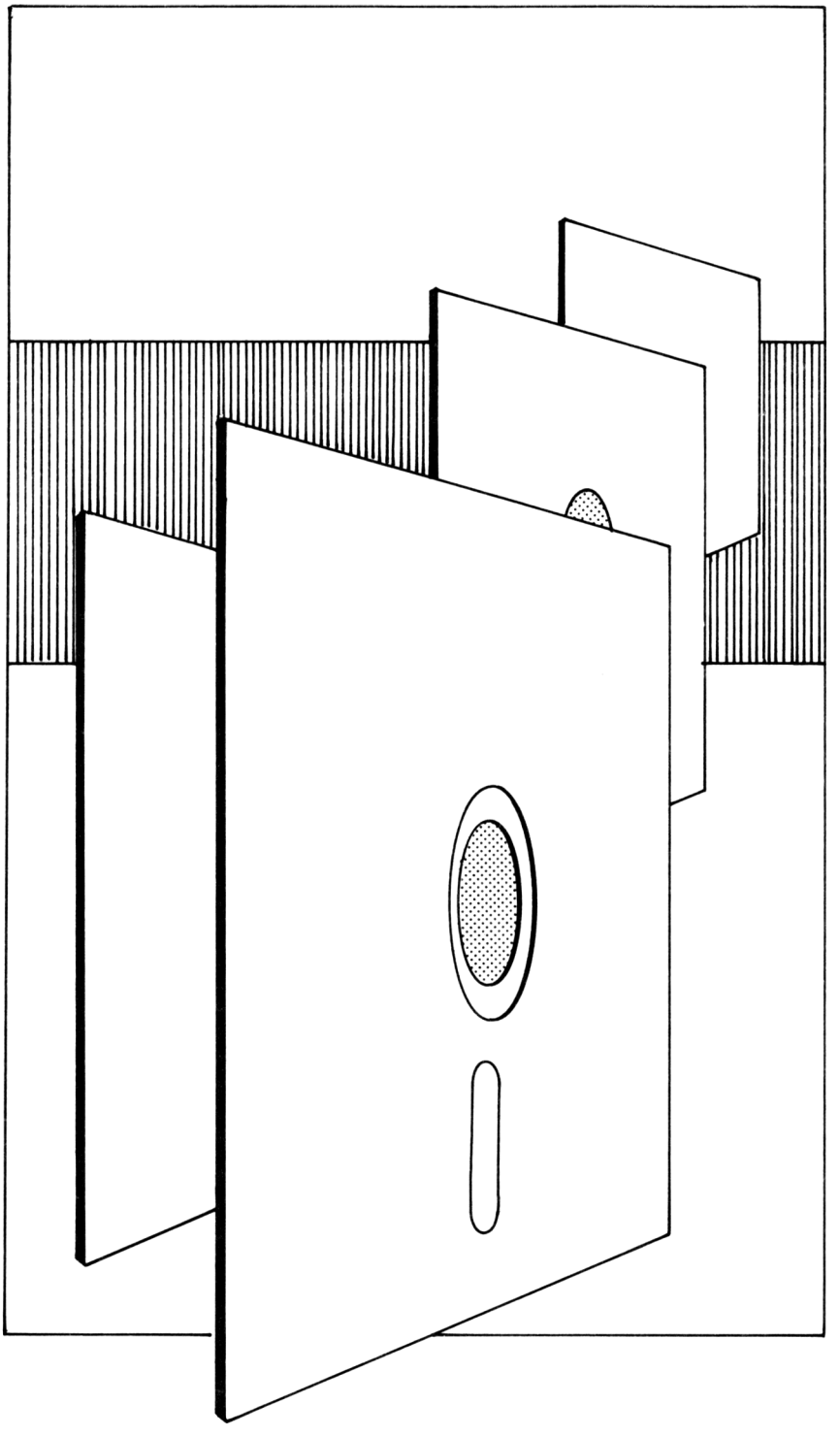
The 8080 instruction set is discussed in detail in Appendix G, including potential pitfalls and interesting techniques. The Z80 mnemonic is also referenced. Appendix H gives a detailed description of the Z80 instruction set. Appendix I summarizes the CP/M BDOS calls.

All of the assembly language programs given in this book were developed on a Z80 microcomputer fitted with three 5-inch disks (drives A, B, and C) and two 8-inch disks (drives D and E). The operating system was the Lifeboat 2.2 version of CP/M. The source programs were written with MicroPro's WordStar word processing program. The programs were assembled with both the Digital Research MAC assembler and the Microsoft MACRO-80 assembler.

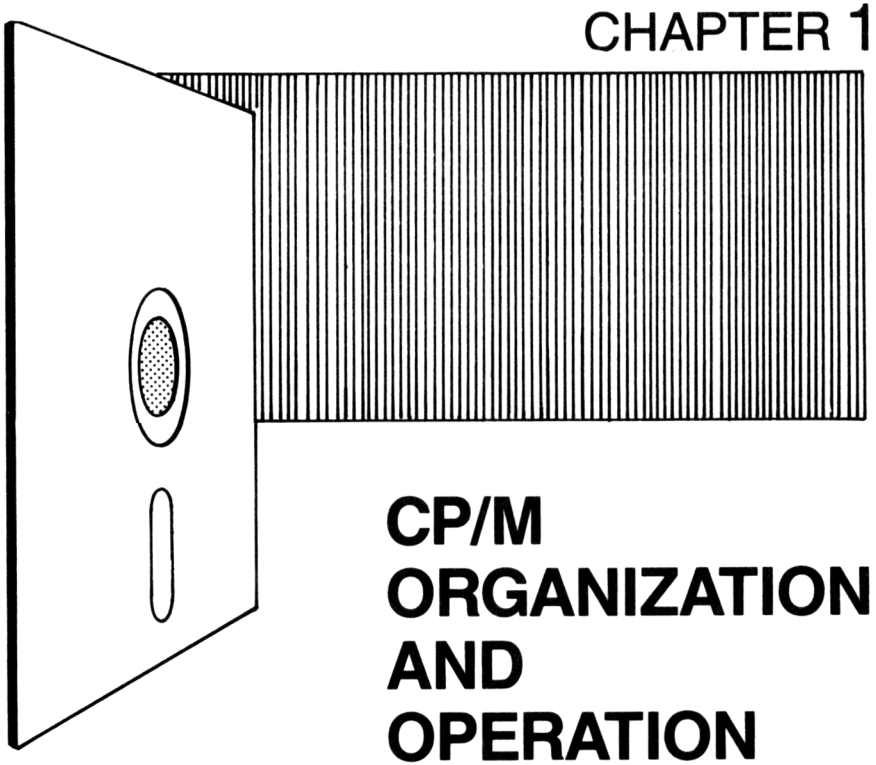
The manuscript was created and edited on the same Z80 computer with WordStar. The manuscript was proofed with Spellguard, a spelling checker, and Grammatik, a syntax checker. The assembly language source programs have been incorporated directly into the manuscript from the original source files. The computer printouts that appear were also incorporated magnetically into the manuscript. This was accomplished by altering the CP/M operating system so that printer output was written into a block of memory. The block was then saved as a disk file. (This technique is described in Chapter 3.) The final manuscript was submitted to the publisher in a magnetic form compatible with the photocomposer.

I am sincerely grateful to Barbara Gordon, editor of the manuscript, for all of her helpful suggestions. I also want to thank Douglas Hergert, Jim Compton, Joe Sharp, and Eric Novikoff for reviewing the manuscript and making additional suggestions. John Wiley & Sons kindly gave permission to reproduce Appendices A–F and H from my book *8080/Z80 Assembly Language: Techniques for Improved Programming.*

Alan R. Miller
Socorro, New Mexico
September 1982



CHAPTER 1



CP/M ORGANIZATION AND OPERATION

INTRODUCTION

The purpose of this first chapter is to review the organization and operation of the CP/M operating system. First we will discuss the various parts of CP/M—the system parameter area, the transient program area, the console command processor, the basic disk-operating system, and the basic input/output system. Then we will summarize the operation of CP/M, including the use of built-in commands, control characters, and transient programs. (Additional details on these subjects can be found in *The CP/M Handbook*.*) After reviewing standard executable programs such as STAT and PIP, we will create a new command, CONTIN, and look at how and why it works.

---

*R. Zaks, *The CP/M Handbook with MP/M,* Berkeley: Sybex, 1980.

MEMORY ORGANIZATION

The hardware of a computer can be divided logically into several parts. These include the central processing unit (CPU), the main or random access memory (RAM), and the peripherals, such as the console, printer, phone modem, and disks. The disk-operating system (DOS) is a software program that coordinates the operation of the computer. CP/M is the most widely used DOS for the 8080, 8085, and Z80 CPUs. Let us review the operation of CP/M.

The 8080, 8085, and Z80 CPUs are very similar. The concepts developed in this chapter apply equally to all three. The CPUs can directly address a maximum of 64K bytes of RAM (actually 2^{16} or 65,536 bytes). Each byte of RAM is assigned an address from 0 to 65,535. CP/M divides this memory into five regions. Beginning with the lowest memory address, the regions are as follows:

- *The system parameter area.* This area contains key items of information such as the current disk and user number, peripheral assignments, the addresses of the basic input/output system and the basic disk-operating system, the restart locations, and the default buffer.
- *The transient program area* (TPA). This is the working area of memory. Executable programs and their data are located here.
- *The console command processor* (CCP). This area contains the programs for the built-in commands DIR, ERA, REN, SAVE, TYPE, and USER.
- *The basic disk-operating system* (BDOS). This area contains the general programs for the operation of peripherals.
- *The basic input/output system* (BIOS). This area contains the customized routines that operate the actual peripherals.

The BDOS and BIOS regions are known collectively as the full disk-operating system (FDOS). The five regions of RAM are summarized in Figure 1.1.

The System Parameter Area

The system parameters, shown in Figure 1.2, begin at address 0. The first three bytes (bytes 0 to 2) contain a jump into the BIOS warm-start entry. When this instruction is executed, CP/M is restarted. This causes a

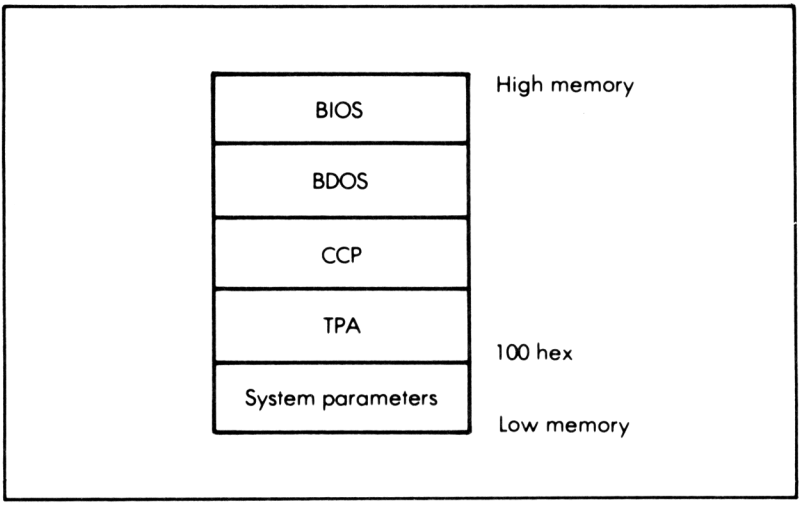


Figure 1.1: Memory Partitions of the CP/M Operating System

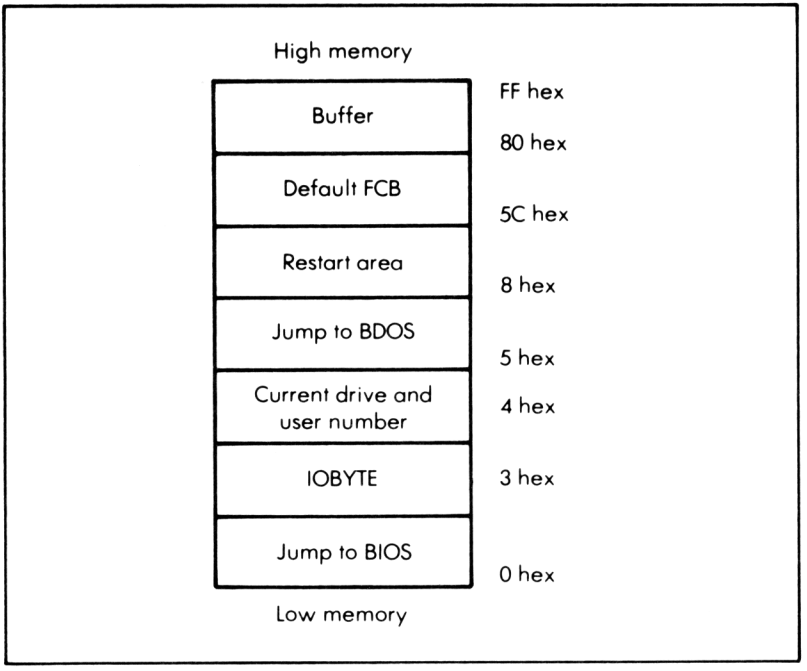


Figure 1.2: The System Parameter Area: 0 to FF Hex

fresh copy of the CCP and the BDOS to be copied into memory from the system disk. The disks are also reset at this time.

The fourth byte of the system parameter area (address 3) is called the IOBYTE. It indicates the current memory assignments of the four logical peripherals: console, reader, punch, and list (printer). The next location, address 4, contains two items: the current disk drive and the current user number. Beginning at address 5, the next three bytes contain a jump into the BDOS. This instruction is executed when console, printer, and disk operations are desired.

The region from address 0 to 38 includes eight locations referenced by the 8080 instructions RST 0 through RST 7:

Instruction	*Hex address*
RST 0	0
RST 1	8
RST 2	10
RST 3	18
RST 4	20
RST 5	28
RST 6	30
RST 7	38

These instructions generate subroutine calls to the corresponding memory locations. The instructions can be activated by hardware interrupts as well as by normal subroutine calls. RST 6 and RST 7 are used by the debuggers DDT and SID. Execution of an RST 0 instruction will perform a warm start, because it causes a branch to address 0.

When a program is executed from the command level of CP/M, one or two file names may be given on the command line. For example, along with the command EDIT, the user might include two parameters:

```
EDIT FIRST.FIL B:SECOND.FIL
```

The region of memory beginning at 5C hex is called the *default* file control block (DFCB) because the CCP automatically selects this region for the file control block area. A file control block is a 32-byte block describing each disk file. The CCP takes the first parameter, FIRST.FIL in this example, for the first FCB. The CCP also initializes a second FCB starting at 6C hex. The second parameter, B:SECOND.FIL in this example, is used this time.

The region from 80 to FF hex is a general buffer area. The *command line tail,* all characters typed after the command itself, is placed in this

region. In the above example, the command line tail is the two file names. This region is also used as the default area for transferring data to and from disks.

The system parameter area is described further in Chapter 3.

The TPA and the CCP

The transient program area usually contains the largest portion of the memory. Beginning at 100 hex, it is the region where executable programs reside.

The console command processor contains instructions for processing commands typed from the console. The area of memory belonging to the CCP is not needed after an executable program has begun operations. Consequently, executable programs may enlarge the TPA to overlap this region. A warm start at the conclusion of the program will reload the CCP along with the BDOS.

The BDOS and the BIOS

The basic disk-operating system contains the device-independent routines for interacting with the console, printer, and disk drives. This region is generally the same for all CP/M computers. We will study its operation in detail in Chapters 5, 6, and 7.

The basic input/output system contains instructions for operating the peripheral devices: the console, printer, phone modem, disks, and so forth. Each BIOS must be customized for the particular set of physical devices actually attached to the computer. Therefore, the BIOS for two identical computers will be different if different peripherals are used. We will learn more about the BIOS in Chapters 2 and 3.

OPERATION OF CP/M

When CP/M is first started up (booted), the CCP, the BDOS, and the BIOS are copied into memory from the system disk (usually drive A). This operation is called a cold start. After loading the system into memory, the cold-start loader transfers control to BIOS. BIOS then fills in the system parameter area at addresses 0 to 7. This includes the jump to BIOS (addresses 0 to 2), the IOBYTE (address 3), the current disk drive and user number (address 4), and the jump to BDOS (bytes 5 to 7).

At this point CP/M displays a prompt symbol to indicate that it is ready to accept a command from the console:

```
A>
```

Built-in Commands

CP/M can control up to 16 separate disks. These are designated by the first 16 letters of the alphabet (A – P). The letter A in the prompt indicates that disk drive A is the current or default drive. The console commands that are built into the CCP can be executed at this time. The following are built-in commands:

Command	*Function*
DIR	List the disk directory
ERA	Erase a disk file
REN	Rename a disk file
SAVE	Create a new disk file from memory
TYPE	Display an ASCII file on console
USER	Change the user number

These commands may not be preceded by a disk name, because they are not associated with any particular disk drive. In other words, the command

```
A:DIR
```

is improper. Some of these commands, however, may take parameters that are disk names. For example:

```
DIR A:
```

The command DIR is given to obtain a listing of the files on the default disk. The listing is arranged across the screen in four columns. All of the files are displayed if no parameter is given.

A whole *class* of files can be selected by using one of the *ambiguous symbols,* * or ?. For example, the command

```
DIR *.ASM
```

will show the names of all files on the default disk that have the type ASM. The asterisk refers to all possible combinations of characters, including blanks. The double asterisk, *.*, refers to all of the files on the disk. For many CP/M commands, the asterisk can be used as an ambiguous file name.

The question mark is used to indicate a single ambiguous character,

including a blank. Thus, the file name SORT?.BAS refers collectively to the following files:

```
SORT1.BAS
SORT2.BAS
SORT3.BAS
SORT.BAS
```

A file or group of files can be erased with the command ERA. Ambiguous symbols may be used (carefully!) in the parameter of ERA. For example:

```
ERA NEW.ASM
ERA *.ASM
ERA *.*
```

The third example erases all files. In this case, however, CP/M asks for verification of the command before erasing all the files on a disk:

```
ALL (Y/N)?
```

You must enter a Y if you want to continue.

We can use the REN command to rename individual files. REN requires two *unambiguous* file names; that is, the * and ? symbols must not be used. The new file name is given first, followed by an equal sign and the old file name. For example, the command

```
REN NEW.ASM=OLD.ASM
```

changes the name of OLD.ASM to NEW.ASM.

The SAVE command makes a disk file from a memory image. SAVE takes two parameters. The first parameter is the number of 256-byte blocks to be saved. The second parameter is the file name. For example, the command

```
SAVE 4 NEWFIL
```

creates a disk file called NEWFIL from the first 1K bytes of the transient program area.

An ASCII disk file can be viewed on the console with the TYPE command. The single parameter must be an unambiguous file name. Scrolling can be stopped by typing control-S. (The task is terminated if any other key is pressed during scrolling.) Scrolling is resumed by pressing any key, but it is wise to use control-S so that you do not unintentionally terminate the command.

The user number can be changed with the USER command. The single

parameter is a decimal number from 0 to 15. CP/M can keep track of 16 different users, numbered from 0 to 15. User 0 is normally selected when CP/M is initialized. Whenever a new disk file is created, it is coded with the current user number. Therefore, each disk file is associated with a particular user number. Only files belonging to the current user are normally accessible.

Control Characters

Several console keys have special meanings to CP/M; following are the control-character commands:

Command	*Function*
control-C	Perform a warm start
control-E	Move to next line
control-H	Back up cursor to previous character
control-I	Tab key
control-J	Execute line (line feed)
control-M	Execute line (carriage return)
control-P	Engage or disengage printer (toggle)
control-R	Reprint current line
control-S	Freeze scrolling
control-U	Cancel current line, start new line
control-X	Cancel current line, restart line

Warm Start

A warm start is performed when control-C is typed and the cursor is in the *first* position of a line. This action is similar to a cold start. The CCP and BDOS are copied from disk drive A into memory. The jumps into BIOS and BDOS at the beginning of memory are also reinitialized, but the memory image of the BIOS is not altered. The current disk drive and drive A are logged in at this time.

When CP/M first accesses a disk drive, it makes a copy of the disk directory and certain characteristics of the disk. You can observe this operation with floppy disks by accessing each one in turn. For example, when you give the command

```
B:
```

the head of disk drive B will be loaded. This is usually indicated by a red

activity light on the front of the disk drive. The system prompt will change to B>. If you have additional drives, you can go to each one in turn by giving its name followed by a colon. If you return to drive A with the command

```
A:
```

the system prompt will change back to A>. However, no disk activity will be apparent because drive A has already been logged in. CP/M assumes that the diskette has not been changed since the last time drive A was accessed.

The disk directory is not reread on subsequent references to a disk. Thus, if you remove a floppy diskette from a drive and replace it with another diskette, you should perform a warm start with the control-C command. This forces a reading of the directory of the new disk. If you neglect to perform a warm start after changing diskettes, CP/M may be able to read the disk. However, if you try to write on this disk, CP/M will refuse to perform the write operation and will issue an error message:

```
BDOS ERROR ON A: R/O
```

CP/M will wait until you type a carriage return. It then automatically performs a warm start, even though you have not typed control-C. Because the new disk is read at this time, it is now possible to write on it.

Transient Programs

The number of commands built into the CCP is limited. Consequently, additional operations are provided by separate, executable programs that are stored as COM files on one of the disk drives. We execute these programs by typing the disk drive and the file name (without the extension COM). The drive name may be omitted if the program is on the default drive. When the name of a transient program is entered, CP/M copies the file from disk into memory and then branches to it.

Programs stored on disk are referenced by a file name. A CP/M file name consists of a primary name and an extension. The primary name contains from one to eight alphabetic or numeric (alphanumeric) characters. Characters other than the letters A – Z and the digits 0 – 9 can be used in certain cases, but it is better to avoid them if you are unsure. For some applications a file type or file name extension is required. In other cases it is not. If an extension is required, it contains from one to three characters. The file type is usually a mnemonic suggesting the nature of

the file. For example:

Extension	*Meaning*
ASM	Assembly language file
COM	Executable (command) file
HEX	Hexadecimal file
BAK	Backup file
BAS	BASIC file
FOR	FORTRAN file
PAS	Pascal file
REL	Relocatable binary file
ASC	ASCII text file
LST	ASCII listing file

Several transient programs are supplied with the CP/M operating system. These are independent executable programs that have a file type of COM:

File name	*Program function*
ASM	Assembler
DDT	Debugger
DUMP	Program to examine executable files
ED	System editor
LOAD	Convert HEX file to COM file
MOVCPM	Change CP/M size
PIP	Copy files
STAT	View disk directory in detail
SYSGEN	Copy system tracks to another disk
SUBMIT	Process a collection of commands
XSUB	Extension of SUBMIT

Following are other common executable programs that are available commercially:

File name	*Program function*
BADLIM	Program to isolate bad disk sectors
COPY	Track-to-track copier
FILEFIX	Disk utility program to undelete files
FORMAT	Initialize a disk
MAC	Digital Research macro assembler
MACRO-80	Microsoft macro assembler
MBASIC	Microsoft BASIC

SAVEUSER	Lifeboat utility to save BIOS on disk
SID	Digital Research symbolic debugger
WS	WordStar text editor

Many other executable programs can be purchased or written. We will write many useful programs in this book.

FIRST EXECUTABLE PROGRAM

Let us begin by creating a very simple executable program. Boot CP/M if it is not already in place. When the process is complete there will be a prompt of

```
A>
```

on the console. Give the built-in DIR command to see what programs are available on drive A. If the executable program STAT.COM appears in the listing, execute it with the command

```
STAT *.*
```

Like DIR, the STAT command produces a listing of all programs on the disk. In addition, STAT arranges the files in alphabetical order.

If you have a printer, turn it on. Then type control-P (to send console output to the printer) and give the STAT command again. The printer will duplicate the output of the console. Type control-P again to disengage the printer. Tear off the printer output and place this directory listing into the diskette envelope for future reference.

STAT gives additional information about the files on the disk. Consider, for example, the following lines:

```
 58  8K 1 R/O A:PIP.COM
266 34K 3 R/W A:WSOVLY1.OVR
```

The first line indicates that the file PIP.COM is located on drive A. This file can only be read (indicated by R/O); that is, it is write protected. Furthermore, the program consists of 58 (128-byte) records that are stored in 8K bytes. The file is referenced by one physical extent.

Smaller files can be referenced by a single disk-directory entry called a physical extent (a 16K block of space on the diskette). If more than one extent is needed for a file, all the extents have the same file name. However, only one of these entries is shown in the STAT and DIR listings.

The second file in the above example, WSOVLY.OVR, is also located on drive A. This file can be both read and written over (indicated by

R/W); it is not write protected. It contains 266 records stored in 34K bytes and requires three physical extents. At the end of the STAT listing, the remaining space on the diskette is given.

Often it is convenient to have a method of returning to a previous command after a warm start has been performed. We will create such a method now. Give the built-in command

```
SAVE 0 CONTIN.COM
```

This puts a new directory entry on the drive currently logged in. However, because the file size is specified as zero, no actual data are saved. If you execute STAT again, the remaining space on the disk should be the same as before. The listing will indicate that the new entry, CONTIN.COM, has zero bytes. We will find that this empty "program" is actually very valuable.

Whenever the command CONTIN is given, CP/M will attempt to load the corresponding program, CONTIN.COM, then branch to the beginning of the TPA at 100 hex. Because the program CONTIN has no data, this command simply restarts the previous program. To see how this works, give the command

```
PIP
```

This will direct CP/M to load PIP.COM into memory and branch to it at 100 hex. PIP responds with an asterisk. You will normally give PIP a command at this point. But in this case, simply type a carriage return. This action will terminate PIP, returning control back to CP/M. CP/M performs a warm start and gives the system prompt, awaiting another command. Now type

```
CONTIN
```

Because this dummy program has no data, the effect is simply to branch to address 100 hex, the beginning of PIP. The memory image of PIP is still intact, so the PIP star should appear again. You can verify this by giving PIP a command. For example, type

```
PIP2.COM=PIP.COM[V]
```

PIP will make a copy of itself calling the new copy PIP2.COM. The parameter V enclosed in brackets causes PIP to verify that the new copy is correct.

This technique will work with many but not all executable programs. For example, it will not work with STAT because data areas are not properly reinitialized when the program is restarted. It will, however, work with MBASIC.

Let us investigate this phenomenon with Microsoft BASIC. Execute BASIC by typing its name. Then write the following BASIC program:

```
10 FOR I = 1 TO 9
20  PRINT I; I*I, 1/I, SQR(I)
30 NEXT I
40 END
```

Try out the program with the command RUN. This program only exists in memory, so you will lose it if you leave BASIC. Therefore, you will normally want to make a permanent copy with the BASIC command

```
SAVE "FIRST"
```

But suppose that you inadvertently typed the BASIC command SYSTEM before saving your program (try it). You will find that you are back at the CP/M system level. Apparently you have lost your BASIC program. Now give the command

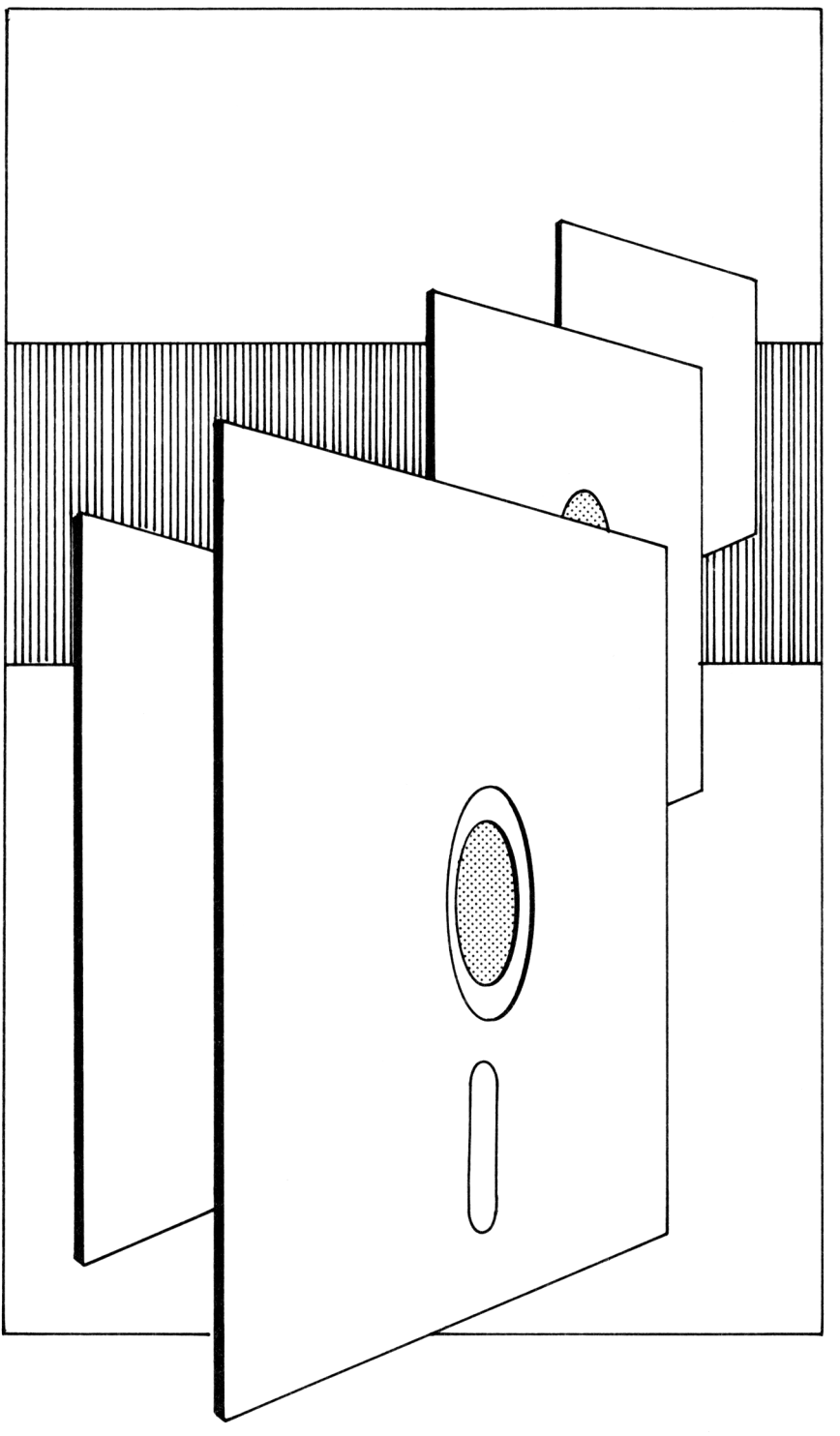
```
CONTIN
```

and you will return to BASIC and the program you wrote with it. Give the command LIST to see that the source program is still there. Then give the RUN command to see that it still works. You can now issue the SAVE command if you want to save your original BASIC program.

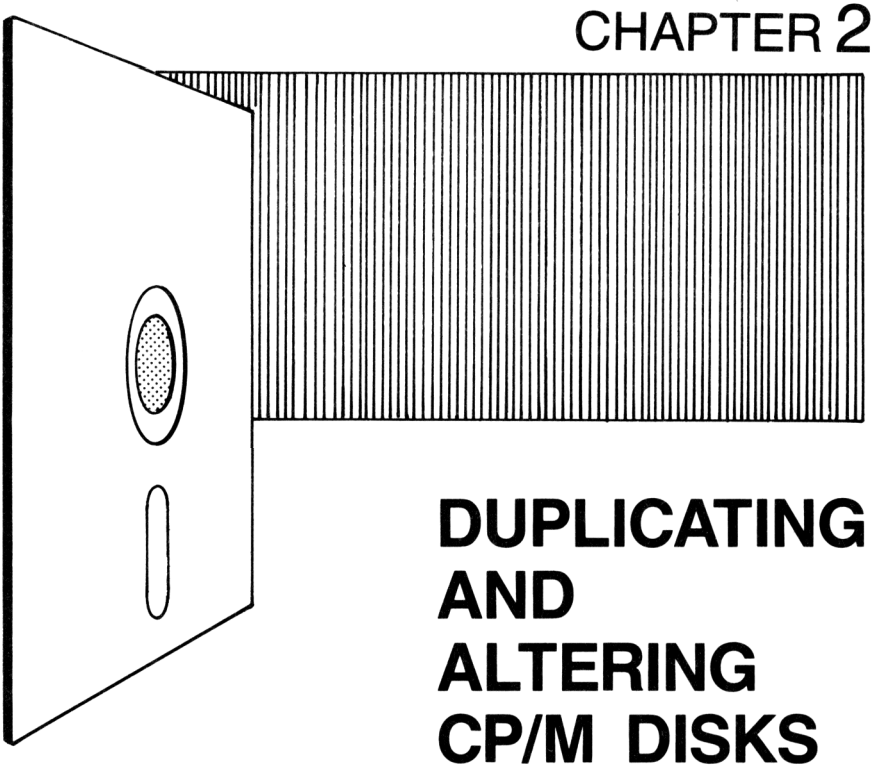
SUMMARY

In this chapter we briefly reviewed the fundamentals of CP/M organization and operation. This included a discussion of the system parameter area, the TPA, the CCP, the BDOS, and the BIOS. The built-in commands, control characters, and some standard executable programs were also considered. We then wrote a short executable program called CONTIN that can be used to restart most executable programs.

In the next chapter we will learn how to copy and alter the disk version of the CP/M system.



CHAPTER 2



DUPLICATING AND ALTERING CP/M DISKS

INTRODUCTION

In Chapter 1 we studied the fundamental CP/M operations and learned how the memory is organized. Because CP/M is a disk-operating system, the disk plays an important role in the operation. Let us therefore focus our attention in this chapter on the organization of the disk.

In this chapter we will learn how to duplicate CP/M disks by formatting a new diskette, copying the data, and copying the system tracks. We then learn how to alter the BIOS or USER routines of the CP/M system, how to assemble and test the new version, and finally how to write a copy of the new version to the disk.

FORMATTING AND DUPLICATING DISKS

Floppy disks are one of the most important devices used to store microcomputer information. The surface of the disk is a magnetic material that is read and altered by a read/write head. (The operation is similar to sound recording with magnetic tape.) Physically, floppy disks are formatted with concentric rings called *tracks*. Each track is divided into regions called *sectors*.

It is common practice to place the CP/M system disk with the executable programs in drive A, and a working disk in drive B. Information can be safely stored on disks provided some precautions are observed. For example, the disks should not be placed near magnetic fields or in a dusty environment. Even when you are careful, an electrical failure during a write operation can result in lost data. Therefore, it is a good idea to make backup copies of all important disks. Let us consider several methods for duplicating the information on disks.

New disks must be formatted before they are used. There are two common floppy-disk sizes—8-inch diameter and 5-inch diameter. In addition, there are disks that are hard or soft sectored, single or double density, and single or double sided. The number of tracks per disk and sectors per track also varies from one version to another.

When you buy floppy disks, you must select the correct diameter (8-inch or 5-inch) and sectoring format (soft or hard). If you require hard-sectored disks, you must also choose the correct number of sectors per track. Many different floppy-disk formats are obtainable from a particular type of disk. Consequently, it will usually be necessary to format a new diskette before it is used for the first time.

Formatting a New Diskette

Floppy disks are formatted by executing a program that is named FORMAT.COM, FORMAT5.COM, or FORMAT8.COM. This program should be included on your original CP/M disk. Format programs have to be specifically tailored to the type of disk you are using. Do not try to format a disk with a program taken from a different computer, because it is not likely to work.

When you use a floppy diskette for the first time, place it into drive B for the formatting operation. In Chapter 1 we saw that a warm start must be performed when changing disks. But this is an exception. Do not perform a warm start after inserting a new, unformatted diskette.

If drive A is not the default drive, give the command 'A:' so that drive A will be the default drive. Be sure that the diskette in drive A contains your

formatting program. Execute this program by typing its name. You may have to answer several questions during program execution. These will deal with whether the diskette should be formatted in single or double density, and whether the drive is single sided or double sided. Some systems can figure these things out automatically, so there may be no questions.

If you open a new box of diskettes, it will be convenient to format all of the disks at once. The FORMAT program is usually written with this in mind. After the first disk has been formatted, change to a fresh disk and press RETURN. The program will usually repeat the previous operation. Remember, do not try to write on a new disk until it has been formatted, or you might get a BDOS error. In the next section we will consider a general technique for making copies of disks using SYSGEN and PIP.

Duplicating a Diskette with SYSGEN and PIP

CP/M disks are partitioned into two regions. These are known as the *system tracks* and the *data tracks.* The system tracks contain the CP/M operating system, including the CCP, the BDOS, and the BIOS. This region of the diskette is not usually accessible. The data tracks are divided into the *directory area* and the *program-storage area.*

Because the system tracks are not normally accessible to the user, the built-in command

```
ERA *.*
```

will erase all of the regular user files stored on the data tracks of the disk, but it will not alter the system tracks.

However, you will need to access the system tracks to make a backup of your system diskette or to alter the BIOS. After a new diskette has been formatted, all of the regular files can be copied from the original diskette to the new one with the PIP program. If the original diskette is on drive A and the new diskette is on drive B, give the command

```
PIP B:=A:*.*[V]
```

Now the new diskette contains all the files from the original diskette. The new diskette can be used in drive B, but it cannot yet be used in drive A. This is because the system tracks, which contain the CCP, BDOS, and BIOS, have not yet been recorded on the new diskette.

A program called SYSGEN can be used to copy the system tracks from one diskette to another. However, SYSGEN cannot do this task directly. It must first copy the system tracks from the source disk into memory.

Then it can copy the memory image to the system tracks of another disk. Let us see how this works.

Execute SYSGEN and follow its directions. There will be slight variations from one version of the program to another, but the general approach is the same. When you execute SYSGEN it might respond with something like this:

```
SYSGEN Version 3.0
Distributed by Lifeboat Associates
for CP/M2 on quad North Star.
Source drive NAME (or RETURN to skip)
```

Enter the name of the source drive but do not include the colon. This is the drive where the original disk is located, normally drive A. SYSGEN repeats your response and then asks for a second carriage return. For example, if you respond with drive A, it will display the following line:

```
Place SOURCE disk on A, then type RETURN
```

When you enter a second carriage return, the response is as follows:

```
Function complete
CP/M image in RAM at 900H is ready to write
or reboot and "SAVE 40 CPMxx.COM"
Destination drive NAME (or RETURN to reboot)
```

During this step SYSGEN copies the CP/M system tracks from the source drive into memory. There are now two copies of CP/M in memory (see Figure 2.1). The working version is at the top of memory and the SYSGEN version is near the bottom, just above SYSGEN itself.

The next step is to write the SYSGEN version of CP/M to the system tracks of a floppy disk. SYSGEN first must know where (on which drive) to write the system. Give the drive name of the new diskette. This will usually be drive B. Again, do not include the colon. SYSGEN responds with the following:

```
Place destination disk on B, then type return
```

When you enter a carriage return SYSGEN copies the system from memory to the system tracks of the new diskette. That completes the process. SYSGEN then prints the following lines:

```
Function complete
Destination drive NAME (or RETURN to reboot)
```

At this point you can place copies of CP/M onto the system tracks of additional formatted diskettes. Remove the new diskette and insert

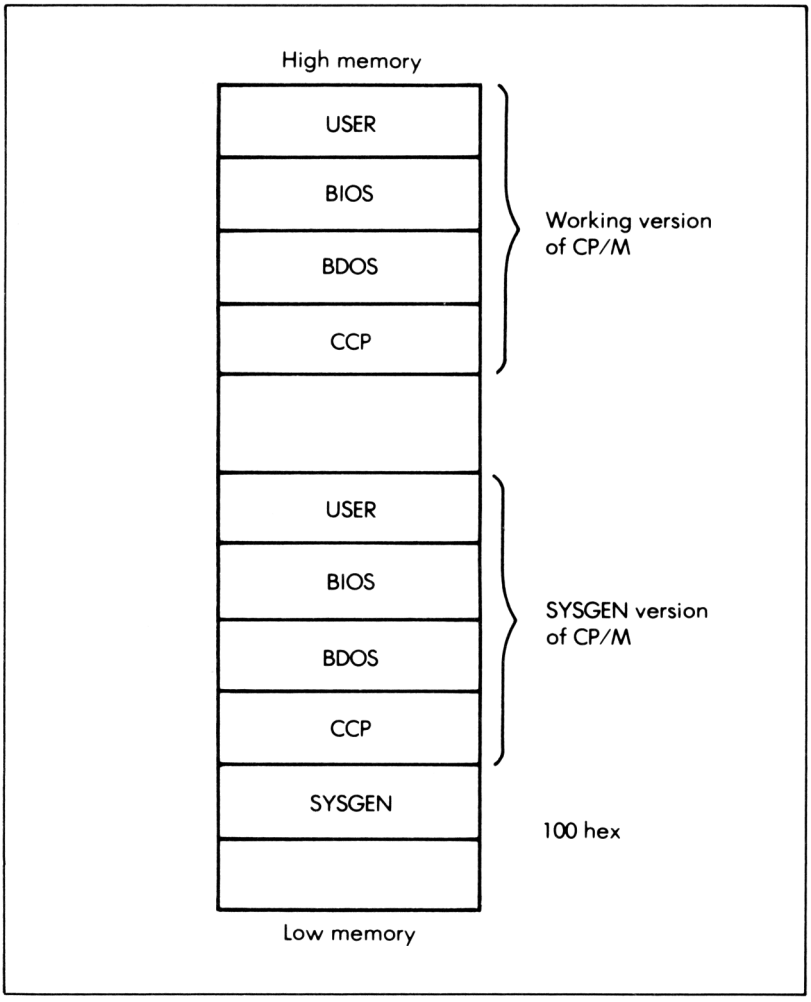


Figure 2.1: The SYSGEN Version and the Working Version of CP/M

another formatted diskette into drive B. Type the letter B and a carriage return. SYSGEN will display the requested drive and ask for another carriage return. SYSGEN will then copy the system from memory to the system tracks of this diskette. In this way you can easily write the system tracks to a number of diskettes, one after the other. If you only want a single copy, simply type a carriage return and the program will terminate. In the next section we will consider another method for duplicating diskettes.

Duplicating a Diskette with Copy

We used PIP and SYSGEN in the previous section to make a duplicate copy of a diskette. In this section we will consider a second method that is simpler and quicker. However, this approach requires a nonstandard program called COPY that may not be provided. Check whether you have an executable program called COPY.COM, COPY5.COM, or COPY8.COM. Usually such a program can perform the three tasks of formatting a new diskette, copying the system tracks, and copying the data tracks all in one operation.

Put the original diskette in drive A and a new, unformatted diskette in drive B. Be careful not to perform a warm start. Execute the COPY program and follow its instructions. Answer the questions about the name of the source and destination drives. In this example, the source drive is A and the destination drive is B. This may be the default option. Before giving the final carriage return, you can change the source disk in drive A if you want to copy a different disk.

If this operation is successful, you have discovered a convenient method of duplicating diskettes. However, the operation may fail if your version of COPY requires a formatted destination disk. Even so, this is a convenient way to copy a disk. Although you must format the new diskette separately, you can copy both the system tracks and the data tracks with the COPY program. We will now learn how to alter the information stored on the system tracks of a disk.

GENERAL PROCEDURE FOR ALTERING THE BIOS

In the previous examples of this chapter we considered methods of duplicating a diskette, including the system tracks. In the next chapter we will want to be able to alter parts of the CP/M system stored on these system tracks. This is an awkward procedure, because the system tracks are not normally accessible. Therefore, in the remainder of this chapter we will discuss three methods for accessing the CP/M system tracks so they can be revised. (Be sure to make a duplicate copy of the system diskette and alter the copy rather than the original.)

The revisions we will perform in the next chapter will be made to the BIOS area of CP/M. The alterations will require an assembly language source program named BIOS.ASM, BIOS.MAC, USER.ASM, or USER.MAC. This program should be provided on your original CP/M diskette. After altering the BIOS source program, we will assemble it, then copy it over the original working version of BIOS. When we are satisfied that the new version performs properly, we will need to save a permanent copy on the system tracks of a floppy disk.

Although we will not actually change BIOS in this chapter, we will cover the necessary steps for moving the altered BIOS into the CP/M system in memory, testing the new BIOS, and saving it on the system tracks of a diskette. These steps are as follows:

1. Alter the BIOS.ASM or USER.ASM source program.
2. Create the corresponding HEX or REL file with the assembler.
3. Copy the HEX or REL file into place with DDT or SID.
4. Try out the new features to see if they work.
5. Copy the new version to the system tracks.

At this time we will assemble the original version of BIOS. Then we will install it in memory to try it out. Finally, we will copy the "new" version to the system tracks of a floppy disk. Because we have not altered the original BIOS, you should not notice any change in the operation of your CP/M. The purpose of this step is to learn how to test an altered version of BIOS and how to make a permanent copy of it on a system disk. First we must determine the location of the working version of BIOS in memory.

LOCATING THE WORKING VERSION OF BIOS

We saw in Chapter 1 that the BIOS region of CP/M contains the tailor-made routines needed to operate the particular peripherals connected to the computer. These routines will be different from one computer to the next. However, the remainder of CP/M, such as the BDOS and the CCP, is universal—that is, it is independent of any particular computer. There is therefore a series of jump instructions, called *vectors,* at the beginning of the BIOS, which gives the addresses of the important routines within the BIOS. Thus it is possible to change the BIOS instructions without affecting the remainder of CP/M operations.

Sometimes these routines reside in a separate region of memory known as the USER area, a subset of BIOS. In either case, a permanent copy of these routines is present on the system tracks and a temporary working copy is present in memory. We can make alterations to the memory image of these routines to see whether a new version does what we want. Once we are satisfied that the operation is correct, we must make a permanent copy of the new version on the system tracks of drive A.

Let us now use the debugger to locate the working version of BIOS, that is, the version at the top of memory. The jump instruction at the beginning of memory references the BIOS warm-start address. Execute the debugger and give the command

```
L0   (the letter L followed by a zero)
```

The letter L is a mnemonic for list. This command is used to disassemble

8080 instructions, that is, to display them in mnemonic form. The parameter zero is the memory address. The first line of the response might be

```
JMP  D303
```

The symbol JMP is the 8080 mnemonic for an unconditional branch instruction and D303 is the operand, the target of the branch. This instruction refers to the BIOS warm-start entry. However, we want the previous cold-start entry at location D300 hex in this example.

The next step will investigate the working BIOS region in memory. Being careful to substitute 3 less than the value you found, give the command

```
LD300
```

The response will be a series of jump instructions. For example:

```
D300    JMP    D380    (initial cold start)
D303    JMP    D39F    (warm-start reset)
D306    JMP    DA06    (console status)
D309    JMP    DA09    (console input)
D30C    JMP    D4E6    (console output)
D30F    JMP    DA0F    (printer output)
D312    JMP    DA12    (punch output)
D315    JMP    DA15    (reader input)
D318    JMP    D4CA    (beginning of disk routines)
D31B    JMP    D499
D31E    JMP    D4CC
```

We must now determine whether there is a separate USER area in addition to the regular BIOS region. If there is only a BIOS region, then all of the jump vectors will be pointing to nearby memory locations, that is, within about 800 hex of each other. Notice that in the above list the first two vectors branch to locations near the beginning of the BIOS (D380 and D39F hex). However, several of the other vectors refer to an area that is farther away (DA06, DA09, and so forth).

Let us examine this second area with the debugger. If we give the command

```
LDA00
```

the response might be as follows:

```
DA00    JMP    DA1B    (initial cold start)
DA03    JMP    DA41    (warm-start reset)
DA06    JMP    DA7E    (console status)
DA09    JMP    DA96    (console input)
DA0C    JMP    DAB6    (console output)
DA0F    JMP    DACC    (printer output)
```

```
DA12    JMP    DB8B    (punch output)
DA15    JMP    DBD6    (reader input)
```

We have found another set of vectors. In this case, all the vectors refer to the immediate memory region. We have located the auxiliary region known as the USER area, a subset of the BIOS routines.

Notice that there is a one-to-one correspondence between many of the vectors for the BIOS region and the corresponding vectors for the USER area. That is, some of the vectors in the BIOS area refer to the same relative positions in a different memory area. For example, address D306 contains a jump to address DA06. One apparent exception in the above list is the console output vector at address D30C. It references address D4E6. However, if this reference is traced with the system debugger, it will ultimately point to the corresponding address DA0C. Disk routines are not usually placed in the USER area, so we do not expect USER jump vectors beyond XX15 hex.

It is important to determine whether your system has a USER area. If there is no USER area, we will make all the changes in the BIOS region using a source program named BIOS.ASM or BIOS.MAC. But if the USER area exists, we will perform the alterations in that area. The source program will be named USER.ASM or USER.MAC.

ASSEMBLING THE BIOS OR USER SOURCE PROGRAM

In this section we will assemble the original source program for the BIOS or USER routines. (If you cannot find the source program, you will not be able to make the revisions we discuss in Chapter 3.) We will then compare the assembled code with the version used by CP/M. This will ensure that your source program matches the version in use.

Assembling the BIOS or USER Source Program with Digital Research MAC

Look on your original CP/M diskette for a program called BIOS.ASM or USER.ASM, and copy it to a working disk. Look at the beginning of this program with the command TYPE or with the system editor. Locate the ORG instruction that establishes the address of BIOS or USER. Be sure that it matches the value you found in the previous step. For our example, the statement is as follows:

```
ORG    0DA00H    ;beginning of USER
```

On the other hand, the operand of the ORG statement may be an expression such as

```
ORG   MSIZE*400H - 600H
```

In this case, the BIOS location is calculated by the assembler according to the memory size (MSIZE). Locate the EQU statement that defines MSIZE and see if it will give the correct value during assembly. Alternatively, you can inspect the assembly listing to see what value the assembler assigned it.

Assemble the source program with the command

```
MAC   BIOS
```

or

```
MAC   USER
```

This step will generate three files with extensions HEX, SYM, and PRN. The HEX file contains the assembled instructions coded in ASCII hex. The SYM file lists the program symbols and their values. The PRN file gives the original source program with the corresponding addresses and assembled code.

Examine the resulting assembly listing with the TYPE command. Find the jump vectors near the beginning of the listing. Compare the addresses of the jump vectors with the values found by the debugger for the actual working copy of BIOS. If these addresses are different, you must change the operand of the ORG statement (or the value of MSIZE in the operand expression) so that the assembled code matches the value you found for the working version of BIOS.

Also compare the targets of the jump vectors to see if they have the same values as the working version of BIOS. If the vector addresses are correct but the target addresses are different, your source program does not match the working version. It still may be possible to use this version, however.

If the jump vectors in the assembly listing match the values you found for the working version, we can try out the assembled version by copying it into place over the working version of BIOS. Give the command

```
SID BIOS.HEX
```

or

```
DDT USER.HEX
```

This will execute the debugger and direct it to copy the HEX file of BIOS or USER into place. CP/M is now using the "new" version of BIOS. You may want to explore the new version with the L command of the debugger. However, you will find that this command no longer works. The debugger

has loaded BIOS into an address larger than itself. Whenever this happens, the debugger L command is automatically disabled. The solution is simple. Return to CP/M with control-C. Then execute the debugger once again.

Assembling the BIOS or USER Source Program with Microsoft MACRO-80

If you use the Microsoft assembler, it will be a bit more difficult to install the assembled BIOS. One method is to replace the ORG directive with a PHASE directive such as

```
.PHASE    0DA00H    ;absolute code
```

Here the operand 0DA00H is the beginning of BIOS or USER. Notice that the symbol PHASE is preceded by a decimal point.

Be sure that the source file has a type of MAC rather than ASM. Assemble the program with the command

```
M80 =BIOS/L
```

In this example, the L switch will direct the assembler to create a PRN file in addition to the usual REL file.

Inspect the resulting PRN file as described in the previous section. Compare the addresses for the jump vectors in the listing to the location of the working version of BIOS. When they agree, you can install the assembled version using the linking loader Link-80 and the debugger. Start with the command

```
L80 BIOS/E
```

This command will convert the file BIOS.REL into an executable version and place it at the beginning of the TPA. The E switch causes the loader to exit to the CP/M operating system after it has created the memory image.

The situation is now very unusual. The newly assembled BIOS is sitting in low memory starting at 100 hex. However, the first instruction contains a jump to the beginning of BIOS (DA00 hex in this case). The program we want actually begins at address 103 hex.

Link-80 has displayed three numbers enclosed in brackets. For example:

```
[DA00 39F 3]
```

The first number (DA00 hex) is the address of the beginning of BIOS. The second number is the location of the end of the TPA memory image of BIOS. The third number, 3, is the program size, that is, the number of 256-byte blocks needed to save the program. Make a disk copy of the memory image by giving the CP/M command

```
SAVE XX BIOS.COM
```

where XX is the number of blocks to save.

Load the new file back into memory with the debugger command

```
SID BIOS.COM
```

Remember, the first three bytes starting at address 100 hex contain an unwanted jump instruction. The actual program begins at address 103 hex. We found that the end of the image is at 39F hex.

Move the image into place with the debugger, being careful to start at address 103 hex rather than address 100 hex. As an example, the move command might look like this:

```
M103,39F,DA00
```

The "new" version of BIOS has now been installed in memory, overlaying the original copy. Of course, we have not yet made any alterations to BIOS or USER.

COPYING THE ALTERED BIOS TO DISK

In the previous example, we assembled the original version of BIOS or USER and copied the assembled version over the working version. In the next chapter, we will add features to the BIOS source program before assembly. We can then test the new features after the assembled version has been installed over the original working version. But if you now turn off the computer, the original version will be loaded next time CP/M is booted. It will be necessary to copy the revised working version of BIOS from memory to the system tracks of a diskette so that you will have a permanent copy. In this example, we have not made any changes to the BIOS. However, let us go through the process of copying the working version to the system tracks to ensure that you understand the process.

There are three different ways to install a revised version of BIOS onto the system tracks of a diskette. We will begin with the easiest method.

Copying BIOS to Disk with SAVEUSER

The simplest method of copying the working version of BIOS to the CP/M system tracks is to run a program called SAVEUSER. However, SAVEUSER is not a regular CP/M program, so you may not have a copy. SAVEUSER directly copies the current working version of the USER area of BIOS to the system tracks of the disk in drive A. To save the current version of USER, type the name SAVEUSER and answer the questions. If you cannot locate a copy of SAVEUSER, then you must use one of the other methods for saving an altered copy of the system.

Copying the Altered BIOS from a HEX File to Disk Using SYSGEN

We saw previously in this chapter that SYSGEN can be used to copy the system tracks from one disk to another. The operation is actually performed in two steps. The system tracks are copied from the source disk into memory, then the memory image is copied to the destination disk.

SYSGEN can also be used to revise the system tracks of a disk. However, in this case the process is stopped in the middle, after the system has been copied from the source disk to the SYSGEN position of memory. The revised copy of BIOS is placed over the SYSGEN position of the original BIOS. Then the revised system is copied to the destination disk with SYSGEN. Let us consider the first part of the SYSGEN operation.

When copying the system tracks from one disk to another, we saw that SYSGEN produces the following message after the system is copied into memory from drive A:

```
CP/M image in RAM at 900H is ready to write
or reboot and "SAVE 40 CPMxx.COM"
Destination drive NAME (or RETURN to reboot)
```

Previously, we gave the name of the destination disk. This time, however, we terminate SYSGEN with a carriage return. Then we save the SYSGEN image (along with SYSGEN itself) as a regular CP/M disk file. In this example, SYSGEN tells us that 40 blocks of 256 bytes are needed to save the system image, but this number may vary from one system to another.

After SYSGEN loads the image into memory, we can simply type a carriage return to go back to the system level. When the prompt symbol A> appears, give the command

```
SAVE 40 CPM2.COM
```

to save the system image as a file named CPM2.COM. This file contains the complete CP/M system and a boot loader if necessary. It also contains a copy of SYSGEN at the beginning.

(Remember that file types are chosen to suggest the nature of a file. CP/M uses the file type COM for executable programs. However, the system image we just saved is not an executable program, because it contains a copy of SYSGEN at the beginning and the remaining parts are in the wrong place. Consequently, it would be more appropriate to choose a file type of SYS. This is possible if you use SID, but DDT requires the extension COM.)

We now have a regular CP/M disk file containing an image of the original CP/M. We must now reload this system image back into memory

with the debugger DDT, so that we can alter it. We are going to load the file into memory starting at 100 hex, because that is the normal working area of memory. Of course, this is not where the system resides when we are using it.

When the debugger is executed, it copies the system image into memory. There are now two copies of the CP/M system in memory (see Figure 2.1). The regular *working* version resides at the top of memory. The duplicate version, generated by the SYSGEN operation, temporarily resides in the TPA just above SYSGEN. We will refer to these two copies as the working version and the SYSGEN version.

The next step is to place the revised copy of the BIOS over the original copy of BIOS in the SYSGEN position. But we first have to determine the address of the BIOS or USER area in the SYSGEN version. The debugger can help find the location.

Execute the debugger with a parameter so it will load a copy of the system image into memory. Give the command

```
DDT CPM2.COM
```

Be careful that CPM2.COM is on the default drive when using DDT. That is, the command

```
A:DDT CPM2.COM
```

is acceptable but the following command is not:

```
DDT B:CPM2.COM
```

This is not a problem with SID.

When the debugger copies the system image into memory starting at 100 hex, it gives three numbers. For example:

```
NEXT PC   END
2900 0100 ACFF
```

Record the number given under the word NEXT (2900 hex in this case). This marks the location of the end of the system image.

The SYSGEN version of the CP/M system we loaded at address 100 hex should be the same as the working version. We determined the location of the working version of BIOS or USER in the previous section. Now we must find the corresponding region for the SYSGEN version. We will use the L command for this purpose.

When the memory image was loaded with the debugger, the NEXT address was displayed on the screen. Because this address references the end of BIOS, start at 100 hex less than this address. If you do not find the vectors, try a smaller address. For example, if the NEXT address was

given as 2900, give the command

```
L2800
```

The response might be as follows:

```
2800   JNZ   DB35
2803   PUSH  H
2804   PUSH  B
2805   LXI   H,DB18
2808   MVI   B,16
280A   MOV   C,M
280B   CALL  DAC4
280E   INX   H
280F   DCR   B
2810   JNZ   DB0A
2813   POP   B
```

We are looking for a set of jump vectors into BIOS or USER. Obviously, this is not it. The jump addresses will be identical to the values we found previously for the working version of BIOS. Repeat the operation with an address that is 100 hex smaller. For example, if we try

```
L2700
```

we might find the following:

```
2700   JMP   DA1B
2703   JMP   DA41
2706   JMP   DA7E
2709   JMP   DA96
270C   JMP   DAB6
270F   JMP   DACC
2712   JMP   DB8B
2715   JMP   DBD6
2718   JMP   DAFC
```

This is the set we are looking for. The addresses of the jump vectors match the USER addresses we found previously.

The next step is to calculate the *offset* (the difference) between the SYSGEN location and the working location of BIOS or USER. We use the H (for hexadecimal arithmetic) command of DDT or SID. This is an undocumented DDT instruction. For this example, the command is

```
H2700,DA00
```

This command subtracts DA00 hex from 2700 hex. The debugger

responds with both the sum and the difference:

```
0100 4D00
```

It is the difference that we want, the second value of 4D00 hex. This is the value we have to add to the address of the regular assembled code (DA00 hex) to place the new BIOS or USER instructions into the proper SYSGEN area (2700 hex).

After altering the BIOS.ASM or USER.ASM program, we assemble it to produce a corresponding BIOS.HEX or USER.HEX file. We need to install this new version in place of the original. The debugger automatically loads the HEX file of instructions in its proper place (over the working version) if the following two commands are given:

```
IUSER.HEX
R
```

(The I command *initializes* an FCB with the file name USER.HEX. The R command *reads* the corresponding disk file into memory.)

However, in this case we want to load the file into the SYSGEN area rather than the working area. We therefore give the R command with the calculated offset:

```
IUSER.HEX
R4D00
```

The debugger will now place the HEX file into the desired SYSGEN area rather than the working area.

At this point we have a copy of the original CP/M system in the SYSGEN position, except that a revised copy of BIOS has replaced the original BIOS. Return to the CP/M system level by typing control-C. You can now save the revised memory image with the command

```
SAVE 40 CPMREV.COM
```

(Be sure to choose a different name than last time so you can distinguish the original version from the revised version.)

Alternatively, you can execute SYSGEN by typing its name. The SYSGEN version of CP/M is already in memory. Therefore, just give a carriage return to the first SYSGEN question:

```
Source drive NAME (or RETURN to skip)
```

This will skip the first part of SYSGEN, which reads the system tracks into memory. Put the desired diskette into drive B, for example, and type the letter B in response to the next question:

```
Destination drive NAME (or RETURN to reboot)
```

SYSGEN then repeats your answer:

```
Place DESTINATION disk on B, then type RETURN
```

When you type another carriage return, the new system image will be written onto the system tracks of the diskette in drive B. Of course, any other drive can be used. Be sure that the disk has been formatted.

You can test whether your alteration of the system tracks has been successful by turning off the computer and booting up using the new diskette.

Copying the Altered BIOS from the Working Version to Disk Using SYSGEN

The third method of writing the system tracks of a diskette is similar to the second method. In this case we do not use a BIOS or USER file directly. Instead, we move a working copy of USER or BIOS down to the SYSGEN position. For the above example, we would load the debugger and the SYSGEN memory image as before with the command

```
DDT CPM2.COM
```

We must determine the BIOS or USER address of the SYSGEN version as we did in the previous section. Then give the M (move) command:

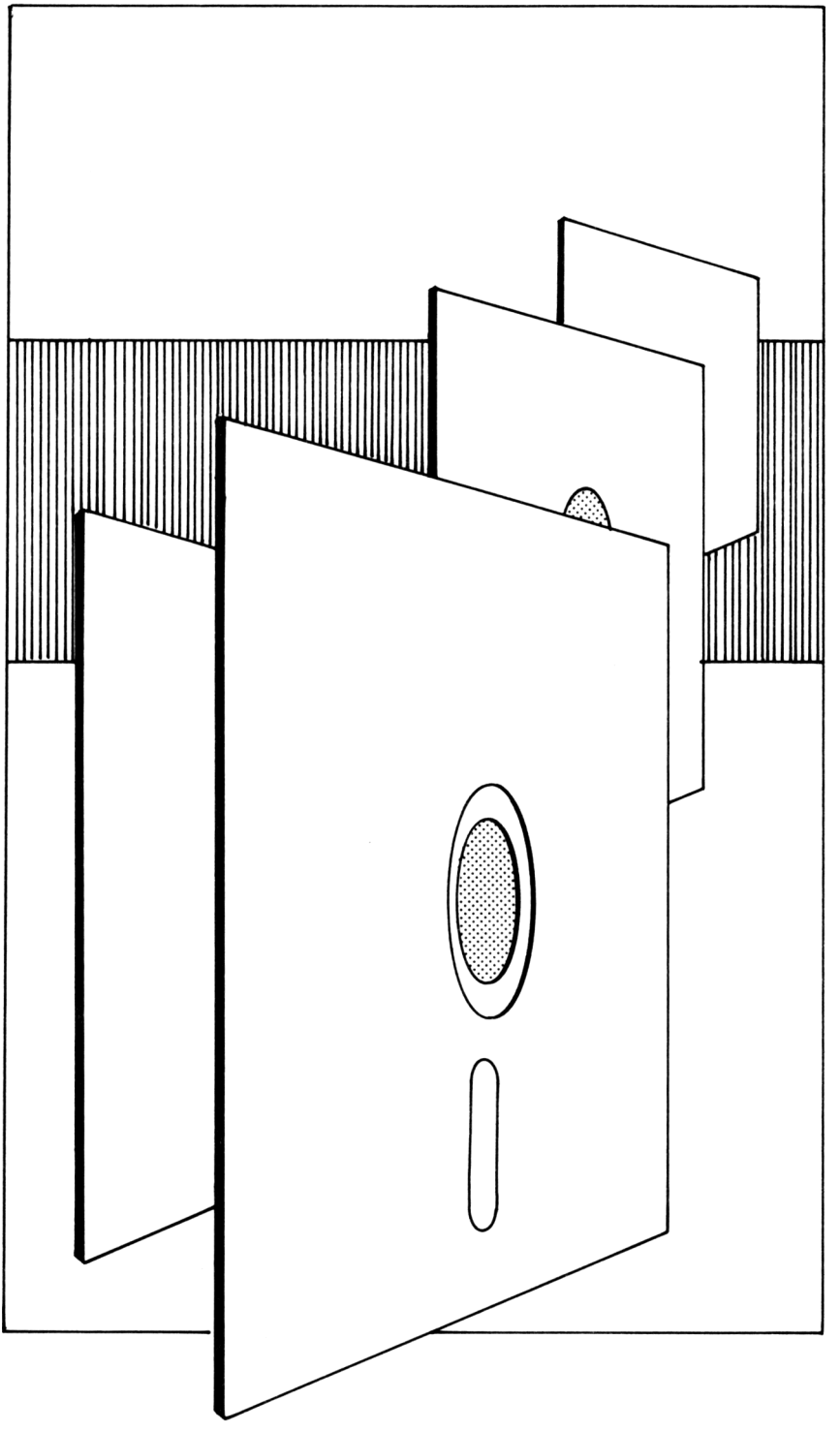
```
MDA00,DDFF,2700
```

This operation will block move a copy of BIOS or USER from the regular working position (DA00 – DDFF) down to the SYSGEN location (2700 – 2A00). We return to the CP/M system by typing control-C, and then we execute SYSGEN. Proceed as in the previous section to save the memory image of the system on the system tracks of a disk.

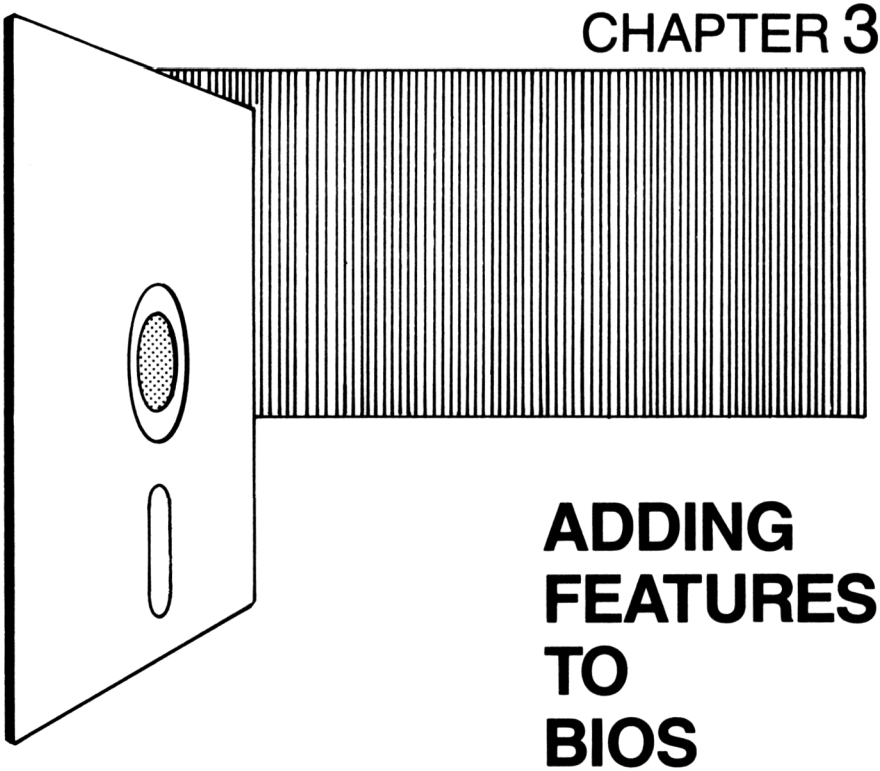
Now that you have a working copy of your BIOS or USER routines, we can begin to add some new features. *Be sure to keep a copy of the original.* Then if your current copy refuses to work, you can go back to the original and begin again.

SUMMARY

In this chapter we have seen how to duplicate a diskette. The steps included formatting a new diskette, copying the data regions onto it, and copying the system tracks onto it. We also learned how to alter the BIOS or USER area of CP/M and how to make the change permanent by writing the new version onto the system tracks. We will now be able to add the features discussed in the next chapter.



CHAPTER 3



ADDING FEATURES TO BIOS

INTRODUCTION

In Chapter 1 we learned that the CP/M basic input/output system (BIOS) contains the software needed to operate the peripherals, such as the console, the printer, and the disks. In Chapter 2 we learned how to access the BIOS or USER area so that it can be modified. In this chapter we will study the BIOS in more detail, and we will modify it to incorporate several useful features. These include the ability to direct the console output to the printer and to check that the printer is turned on.

Because only a small amount of memory is allocated for BIOS routines, it is necessary to write the programs in assembly language rather than in a higher-level language such as BASIC or Pascal. Let us therefore review the operation of assemblers.

ASSEMBLY LANGUAGE PROGRAMMING

Assembly language is a low-level computer language in which the instructions of a particular CPU are selected directly by a mnemonic operation code (opcode). The 8080 CPU has three general-purpose, 16-bit registers. They are given the names HL, DE, and BC. The complete instruction sets for both the 8080 and the Z80 CPUs are given in the Appendices.

Consider, for example, the following operation code:

```
JMP   D303
```

This instruction tells the CPU to branch to address D303 hex. The assembler generates the corresponding binary code. Thus there is a one-to-one correspondence between an assembly language instruction and the CPU operation it generates.

By contrast, a single instruction in a high-level language, such as Pascal or BASIC, usually generates more than one CPU instruction. Although each compiler operates differently from the next, the line

```
I = I + 5
```

might be converted into the following CPU instructions:

```
LXI   D, 5
LHLD  I%
DAD   D
SHLD  I%
```

This sequence of instructions loads the DE register *with* the value of 5 and the HL register *from* the location of I%. The values in DE and HL are added together and the result is placed in HL. The result is then stored in the memory location referenced by I%.

Assembly language programs are written and altered with one of the many CP/M editors, such as ED, WordStar, WordMaster, Magic Wand, PMATE, or Benchmark, among others. The resulting source program is assembled with an assembler program, then converted into executable binary code. The CP/M operating system provides an assembler called ASM. This assembler is not suitable for many of the programs in this book, however, because it does not incorporate a *macro processor.* (We will begin discussing macros and macro processors in Chapter 4.) There are two common CP/M assemblers that do contain a macro processor. These are the Microsoft MACRO-80 assembler and the Digital Research MAC assembler. Both of these macro assemblers accept the standard Intel 8080 mnemonics. The Microsoft assembler can also use the Zilog Z80

mnemonics directly. The Digital Research assembler can only generate the Z80 opcodes with macros.

A Simple Assembly Language Program

To ensure that you understand the operation of your assembler and the associated programs, we will assemble and execute a very simple program in 8080 assembly language. Use a CP/M editor to generate the source program shown in Figure 3.1, and give it a file name of BELL. The file type should be either ASM for the Digital Research assemblers or MAC for the Microsoft assembler. If you are using the Microsoft assembler, omit the ORG statement and the apostrophes enclosing the TITLE statement. This is one of the few programs you can assemble with the Digital Research ASM assembler. Use the file type of ASM, but omit the first line beginning with the word TITLE.

Notice that there are generally four different columns of information in the listing of the source program. It is common practice to use the ASCII tab character to align these four columns. The Digital Research and Microsoft assemblers do not require such an alignment, but it makes the source program easier to understand. For a regular operation code, the four columns are as follows:

```
LABEL:    MNEMONIC    OPERAND    ;comment
```

The label consists of alphanumeric characters and is terminated by a colon. (The colon is optional for the Digital Research assemblers but required for the Microsoft assembler.) Program control can be transferred to the label from any other part of the program. The mnemonic corresponds to the desired CPU instruction; its spelling may differ from one assembler to another. The operand is the parameter for the CPU instruction; it can refer to a CPU register, a constant, or a memory address. The comment, which is preceded by a semicolon, documents the instruction.

Not all lines in the source program contain opcodes. Some lines contain *assembler directives* or *pseudo operation codes* (pseudo ops) instead. These lines do not generate CPU instructions; rather, they are used to create constants, set aside memory locations, or give directions to the assembler. For example, the first line,

```
TITLE    'Ring the console bell'
```

directs the assembler to place the indicated title at the top of each page of the assembly listing. The directive

```
ORG    100H
```

```
TITLE    'Ring the console bell'
;(Put current date here)
ORG      100H                    ;Digital Research version
;
BEL      EQU      7              ;ASCII bell char.
BDOS     EQU      5              ;DOS entry point
TYPEF    EQU      2              ;type char. on console
;
START:
         LXI      SP,100H
         MVI      C,TYPEF
         MVI      E,BEL
         CALL     BDOS
         JMP      0
         END      START
```

Figure 3.1: Program BELL to Ring the Console Bell

sets the address of the assembled code to 100 hex. The next three lines are called *equates.* They define the values of the symbols BEL, BDOS, and TYPEF. For example,

```
BEL    EQU    7
```

defines the value of BEL to be 7. We omit the colon at the end of a definition label because it does not represent a memory location.

Five lines of the source program in Figure 3.1 actually generate computer instructions. The first instruction sets the stack pointer to 100 hex:

```
LXI    SP,100H
```

(The stack pointer is a CPU register that refers to a particular region of memory. In this example we are initializing the pointer to a value of 100 hex. However, its value is altered by instructions such as PUSH, POP, CALL, and RET.) The second instruction places the value of 2 (TYPEF) in the C register:

```
MVI    C,TYPEF
```

The third instruction loads the E register with the value of 7 (BEL):

```
MVI    E,BEL
```

The fourth instruction generates a subroutine call to address 5 (BDOS). The fifth instruction branches to address 0.

The final statement in the program declares the starting address to be the label START.

Program Assembly

After you have created your source program with the editor, obtain a listing and compare it to Figure 3.1. Correct any errors, then assemble the program. If you are using the Digital Research MAC or ASM assemblers, give the command

```
MAC BELL
```

or

```
ASM BELL
```

For the Microsoft assembler, type

```
M80 =BELL/L
```

An assembly listing file named BELL.PRN will be created at this step. Compare your assembly listing to Figure 3.2 for the Digital Research assemblers or Figure 3.3 for the Microsoft assembler.

The assembly listing gives the original source program along with the corresponding instructions and the addresses where the instructions will reside during execution. The instructions and addresses are given in hexadecimal notation. Instructions such as JMP and CALL, which refer to memory locations, are three bytes long. The second and third bytes contain the memory address. The low half of the address is stored in the second byte and the high half is stored in the third byte. That is, the two bytes of the address appear to be reversed. The Digital Research assembler gives the address in this reversed order. For example, a call to BDOS at address 0005 looks like this:

```
CD0500    CALL    BDOS    ;Digital Research version
```

However, it is more natural to think of a two-byte address as the high byte followed by the low byte. As a consequence, the Microsoft assembler gives the address with the high byte shown first. Thus a call to BDOS looks like this:

```
CD 0005   CALL    BDOS    ;Microsoft version
```

It must be remembered that the sequence of bytes in memory matches the Digital Research order rather than the Microsoft order.

```
                TITLE   'Ring the console bell'
                ;(Current date)
0100            ORG     100H            ;Digital Research version
                ;
0007 =          BEL     EQU     7       ;ASCII bell char.
0005 =          BDOS    EQU     5       ;DOS entry point
0002 =          TYPEF   EQU     2       ;type char. on console
                ;
                START:
0100 310001             LXI     SP,100H
0103 0E02               MVI     C,TYPEF
0105 1E07               MVI     E,BEL
0107 CD0500             CALL    BDOS
010A C30000             JMP     0
                ;
010D                    END     START
```

Figure 3.2: Digital Research Assembly Listing for Figure 3.1

The next step is to run the program. However, we cannot do this just yet, because the assembler has not created an executable file. The Digital Research assembler has generated an ASCII hexadecimal file called BELL.HEX. This HEX file can be converted into an executable file named BELL.COM by giving the command

```
LOAD BELL
```

(LOAD is a program that is included with the CP/M operating system.) Now give the command

```
BELL
```

to execute the program. The console bell should sound, and control will return to the CP/M operating system.

The Microsoft assembler, on the other hand, creates a REL file, which must be processed differently. It is possible to create a HEX file from a Microsoft REL file, but it is simpler to convert the REL file into an executable file with the linking loader L80. For example, the program BELL.REL can be executed with the command

```
L80 BELL/G
```

This command will generate a binary file, starting at memory location 100

```
                        TITLE   Ring the console bell
                        ;(Current date)
                        ;
0007                    BEL     EQU     7       ;ASCII bell char.
0005                    BDOS    EQU     5       ;DOS entry point
0002                    TYPEF   EQU     2       ;type char. on console
                        ;
0000'                   START:
0000'  31 0100                  LXI     SP,100H
0003'  0E 02                    MVI     C,TYPEF
0005'  1E 07                    MVI     E,BEL
0007'  CD 0005                  CALL    BDOS
000A'  C3 0000                  JMP     0
                        ;
                                END     START
```

Figure 3.3: Microsoft Assembly Listing for Figure 3.1

hex, and execute it. After the program has finished execution and the CP/M prompt is displayed, type

```
SAVE 1 BELL.COM
```

(We learned in Chapter 2 that L80 tells us the number of blocks to save.) This will save the executable memory image. The program can be run again by typing the name BELL. In Chapter 1 we created a program called CONTIN, which we can execute to rerun BELL since the memory image is still intact.

It is also possible to create a COM file with L80. For example, the command

```
L80 BELL/N, BELL/E
```

will generate a disk file named BELL.COM and then exit to CP/M. The program can be run by typing the name BELL.

We can now prepare to alter the BIOS.

BIOS ENTRY VECTORS

We learned in Chapter 2 that there is a series of vectors at the beginning of the BIOS that gives the addresses of the corresponding routines within

the BIOS. We also learned that some versions of CP/M incorporate an extension to the BIOS called USER. In those cases the BIOS contains the disk operation routines and the USER area contains the remaining routines. There is one set of vectors at the beginning of the BIOS and a second set of vectors at the beginning of the USER area. The vectors at the beginning of the BIOS that relate to disk operations will point into BIOS. The remaining vectors, which refer to console and printer operation, will generally refer to a matched set of vectors at the beginning of the USER area.

Vectors at the beginning of BIOS are shown in Figure 3.4. The first vector is called the cold-start entry. It is used during the initial startup of CP/M. The second vector is used at the completion of major tasks; it is called the warm-start vector. Vectors for the four *logical devices,* the console, reader, punch, and list, appear next. These four devices are referenced by the following symbols:

CON:	Console (input and output)
RDR:	Reader (input)
PUN:	Punch (output)
LST:	List or printer (output)

Notice that the symbolic names for logical devices end in a colon. This is consistent with the naming of disk drives as A:, B:, etc. By this means, a device name can be distinguished from a disk file name. For example, the name PUN refers to a disk file, whereas PUN: refers to the logical punch device.

Exploring the BIOS Vectors with the Debugger

In Chapter 2 we located the BIOS jump vectors by using the debugger. If you have not already performed this task, you should do so at this time.

Recall that we executed the debugger and gave the command

```
L0   (the letter L followed by a zero)
```

The expected response is something like this:

```
0000   JMP   D303
0003   NOP
0004   NOP
0005   JMP   AD00
  . . .
```

The branch at address 0 references the warm-start entry into BIOS. Thus, for this system BIOS begins at address D300 hex. The branch at

```
BIOS      JMP   COLD     ;initial cold start
BIOS+3    JMP   WARM     ;warm-start reset
BIOS+6    JMP   CSTAT    ;console status
BIOS+9    JMP   CONIN    ;console input
BIOS+12   JMP   CONOUT   ;console output
BIOS+15   JMP   LIST     ;printer output
BIOS+18   JMP   PUNCH    ;punch output
BIOS+21   JMP   READER   ;alternate input device
```

Figure 3.4: The First Eight CP/M BIOS Vectors

address 5 is usually the BDOS entry; we used this location to ring the console bell in the program BELL. Now, however, the address stored at location 5 has been altered by the debugger. That is, the normal BDOS address for this system is C506 hex, but in this example the debugger changed it to AD00 hex.

When DDT (or SID) is executed, CP/M copies the program into memory at the beginning of the TPA and branches to it. The debugger then relocates itself into high memory. This allows another program (the one to be debugged) to be loaded into the TPA and run under the control of the debugger. However, the debugger needs to intercept BDOS calls made by the program it is studying. Consequently, it changes the BDOS address stored at location 5.

After finding the location of BIOS, we can disassemble the vectors at the beginning of BIOS by giving (in this example) the debugger command LD300. The response might be as follows:

```
D300   JMP   D380   (initial cold start)
D303   JMP   D39F   (warm-start reset)
D306   JMP   DA06   (console status)
D309   JMP   DA09   (console input)
D30C   JMP   D4E6   (console output)
D30F   JMP   DA0F   (printer output)
D312   JMP   DA12   (punch output)
D315   JMP   DA15   (alternate input device)
D318   JMP   D4CA   (beginning of disk routines)
D31B   JMP   D499
D31E   JMP   D4CC
```

We saw in Chapter 2 that there might be a second set of jump vectors in

a separate region of memory known as the USER area. For the above example, the USER area starts at address DA00 hex. If we examine this area with the debugger command LDA00, the following response might appear:

```
DA00   JMP   DA1B   (initial cold start)
DA03   JMP   DA41   (warm-start reset)
DA06   JMP   DA7E   (console status)
DA09   JMP   DA96   (console input)
DA0C   JMP   DAB6   (console output)
DA0F   JMP   DACC   (printer output)
DA12   JMP   DB8B   (punch output)
DA15   JMP   DBD6   (alternate input device)
```

In the following sections we will be interested in the vectors at addresses DA0C and DA0F, which branch to the routines that operate the console and the printer.

ENGAGING THE PRINTER WITH THE DEBUGGER

Sometimes it is desirable to reproduce console output on the CP/M printer (list device). This can be accomplished by typing control-P during console input. However, an executing program cannot engage a printer in this way. Nevertheless, it is sometimes desirable for a program to be able to engage the printer. In the next section we will write a short program to accomplish this task; but first we will perform the feat more directly, using the debugger.

Notice that the vector pointing to the printer routine (at address DA0F in the above list) immediately follows the vector for console output (at address DA0C). By changing the *console output jump* instruction to a *call* instruction, we can activate console and printer output simultaneously. We will make this change with the debugger. This type of operation is sometimes called a *patch.* You must be very careful with this step, because you are actually changing the BIOS. You are only going to change one byte, but you must not use the wrong value or change the wrong byte. Otherwise, CP/M will not respond to your commands or it may do strange things.

Use the debugger command A (for assemble) to change the location of DA0C (in this example):

```
ADA0C              (you type this)
DA0C CALL DAB6     (you type CALL DAB6)
DA0F               (you type a carriage return)
```

In this example, we used the debugger to engage the printer by changing a jump instruction to a call instruction. Any executing program (except Microsoft BASIC) can also use this technique.

Alternatively, we can use the debugger command S (for set) to change the jump instruction (C3 hex) to a call instruction (CD hex). The commands are as follows:

```
SDA0C          (you type this)
DA0C C3 CD     (you type CD)
DA0D B6 .      (you type a period)
```

From now on, the printer should display the same information as the console screen. We can return the BIOS to its original condition by changing the call instruction back to a jump instruction. (If something has gone wrong and CP/M no longer works, just perform a cold boot. You may have to turn the computer off and on again to get it working.) Let us now automate this patching operation.

A PROGRAM TO ENGAGE AND DISENGAGE THE PRINTER

We can make the process of engaging and disengaging the printer under computer control easier by using two programs to do the patching. Because these programs are so short, we will create them with the debugger rather than with the assembler. Load the debugger and give the command A100 to assemble a program starting at 100 hex. Then type the following instructions:

```
LHLD   1
LXI    D,9
DAD    D
MVI    M,CD
RET
```

Type an extra carriage return to terminate this step.

After this short program has been written, disassemble it by giving the command L100. The result should look like this:

```
0100   LHLD   0001
0103   LXI    D,0009
0106   DAD    D
0107   MVI    M,CD
0109   RET
```

Return to CP/M by typing control-C. Save the program:

```
SAVE 1 LISTON.COM
```

Before you run this program, create the complementary program to restore the BIOS vector to its original state. Load LISTON with the debugger:

```
DDT LISTON.COM
```

Change the call instruction at location 108 hex, the second operand of the MVI instruction, to a jump instruction (C3 hex). (Use an S108 command to deposit the value of C3 or enter the instruction 'MVI M,C3' with an A107 command.) Return to CP/M and save the second program with the command

```
SAVE 1 LISTOFF.COM
```

When LISTON is executed, the first instruction loads the BIOS warm-start address into the HL register. (Recall that address 0 contains the jump instruction and addresses 1 and 2 contain the warm-start address.) The second instruction loads the DE register with the value of 9, the difference between the warm-start entry and the console-output entry. The third instruction adds the HL and DE registers, placing the sum in HL. The HL register now refers to the console-output vector. The fourth operation places a call instruction (CD hex) over the console-output jump instruction. The final instruction returns to the system level of CP/M.

Because hexadecimal is the default mode of the debugger, we can enter hex data without the suffix H. By contrast, decimal is usually the default mode for an assembler.

To test these programs, turn on the printer and give the CP/M command

```
LISTON
```

followed by

```
DIR
```

The directory listing should appear at both the console and the printer. Then give the commands

```
LISTOFF
DIR
```

The directory listing should appear only at the console.

These two short programs are more useful for the insight they give into the workings of CP/M than for their actual operation. In fact, they will not always work as expected. In particular, they will not operate with

Microsoft BASIC. We will now alter the BIOS so the printer can be engaged by changing the value of the IOBYTE. (Refer to Chapter 2 for a review of how to access and alter the BIOS or USER routines.)

ENGAGING THE PRINTER WITH THE CP/M IOBYTE

We learned that the BIOS provides vectors to the operation of the four logical peripherals: console, reader, punch, and printer. CP/M provides a mechanism for mapping these four logical devices to 16 physical devices. That is, each of the four logical devices can be assigned to as many as four different actual devices. At any time, the current assignments for the four logical devices are coded in a single byte located at address 3. The two low-order bits hold the console assignment, the next two refer to the reader, the two after that refer to the punch, and the two high-order bits hold the printer assignment.

While the IOBYTE feature can be used to map the four logical peripherals to 16 actual devices, it is not necessary to implement all these capabilities. Each part can be added as a single step, greatly simplifying the process. The IOBYTE feature can be useful even if the console and the printer are the only peripherals.

Let us begin with a simple implementation of the IOBYTE—engaging and disengaging the printer. We will designate the low-order bit of the IOBYTE at address 3 as a printer switch. If this bit has a value of 1, the printer will display console output. Otherwise, the printer will not respond to console output. Of course, the printer can still be engaged by typing control-P in the usual way. Furthermore, the video screen will always display the console output, whether or not the printer is engaged.

The first thing we have to do is ensure that the IOBYTE is properly initialized. Look at the BIOS assembly listing and locate the first jump vector. This will be the first executable statement near the beginning of the program. Now find the referenced address and follow the instructions until a return statement is encountered. Somewhere in this section there may be statements like the following:

```
COLD:
        MVI     A,0
        STA     3
```

or

```
COLD:
        MVI     A,INITAL
        STA     IOBYTE
```

In the second example the value of INITAL is defined as 0 and the IOBYTE is set to a value of 3. If you cannot find such a passage, insert the equivalent instructions with the system editor. Be sure to define the symbols INITAL and IOBYTE if you use the second form. For example:

```
INITAL   EQU  0
IOBYTE   EQU  3
```

The next step is to alter the console-output routine. Look at the BIOS assembly listing and locate the console-output vector (XX0C hex) and the list-output vector (XX0F hex). These are the fifth and sixth vectors in the list. Note the labels used as the operands. They might be something like this:

```
XX0C   JMP   CONOUT
XX0F   JMP   LIST
```

Go to the location of CONOUT and insert the following code at the very beginning:

```
CONOUT:                    ;console output
        LDA   IOBYTE       ;get the value
        ANI   1            ;mask for bit 0
        CNZ   LISTT        ;printer output
CONO2:                     ;regular console output
```

The first instruction loads the accumulator from memory address 3, the location of the IOBYTE:

```
LDA   IOBYTE
```

The second instruction performs a logical AND with the accumulator and the value of 1:

```
ANI   1
```

This masking AND operation resets (zeros) all but the low-order bit (bit 0) of the accumulator. The operation also alters the zero flag of the CPU accordingly. That is, the zero flag is set if the low-order bit is zero. It is reset otherwise.

The third instruction tells the CPU to call the printer subroutine at LISTT if the zero flag is reset (the low-order bit is not zero):

```
CNZ   LISTT
```

Be sure to include the label CONO2. We will need it for a later program in this chapter. Also notice that the branch to the printer routine is called

LISTT. This is necessary to distinguish the logical list from the physical list. Find the location of the label LIST. If an opcode also appears on this line, split the line in two so that the label is on a line by itself. Add the label LISTT: on the line immediately following the label LIST.

Put today's date as a comment statement near the beginning of the source program.

Changing the IOBYTE with the System Debugger

Assemble the new version and load it into memory with the debugger. Check the IOBYTE at address 3 to see that it has a value of 0. Give the debugger command S3. The response will be

```
0003 X
```

where X is the value of the IOBYTE. If this value is 0, enter a period to terminate this step. Otherwise, enter the value of 0. Turn on the printer, and with the S command of the debugger, change memory address 3 to a value of 1:

```
S3          (you type this)
0003 0 1    (type a 1)
0004 0 .    (type a period)
```

The last line above should be displayed on the printer as well as on the console screen, because the IOBYTE is now 1. Try some other commands, such as

```
D0
```

The printer should again follow the console screen. Change the IOBYTE back to 0 with the S command. The printer should no longer repeat the console output. You must now copy the new BIOS version to the system tracks if you want to make it permanent. Of course, it should still be possible to turn on the printer with a control-P command.

Changing the IOBYTE in BASIC

Now we will try out this method with Microsoft BASIC. Load BASIC and write a short program such as the one we used in Chapter 1:

```
10 FOR K = 1 TO 9
20   PRINT K, 1/K, K*K
30 NEXT K
40 END
```

Then run the program. The results will appear on the console. Turn on the printer and give the following commands:

```
POKE 3,1
RUN
```

The BASIC POKE command will change the IOBYTE to a value of 1. When the program is run, output will appear at the printer as well as the console.

We have already noted that Microsoft BASIC will not allow the printer to be engaged with control-P or with the LISTON program. Now we have a method of performing this task. The printer can be disengaged with the BASIC command

```
POKE 3,0
```

Return to CP/M with a SYSTEM command.

Changing the IOBYTE with STAT

We have learned how to change the IOBYTE at address 3 with the system debugger or in BASIC. It is also possible to change the IOBYTE with STAT.

We have seen that the four logical devices CON:, RDR:, PUN:, and LST: are each allocated two bits of the IOBYTE. Four separate physical devices can be assigned to each of these logical devices through changes in the IOBYTE. STAT has 16 names coded into it for this purpose. The 16 names are given in Table 3.1. You can get STAT to display this table by giving the command

```
STAT VAL:
```

The IOBYTE can be changed from 0 to 1 by typing the command

```
STAT CON:=CRT:
```

STAT will change the IOBYTE back to 0 with the command

```
STAT CON:=TTY:
```

Changing the STAT Device Names The names for the four logical devices were chosen years ago when teletypewriters (TTY) were common. It might be more meaningful now to change them to something else. For example, TTY: could be changed to CRT: and CRT: could be changed to LST:. This change is easily accomplished with the system debugger.

Table 3.1: STAT's Name for the Four Logical Devices

	Bits			
	00	01	10	11
CON:	TTY:	CRT:	BAT:	UC1:
RDR:	TTY:	PTR:	UR1:	UR2:
PUN:	TTY:	PTP:	UP1:	UP2:
LST:	TTY:	CRT:	LPT:	UL1:

Copy STAT into memory with the command

```
DDT STAT.COM
```

Look at the first part of STAT with the command

```
D100
```

The ASCII representation of the data on the right side of the screen shows the four logical device names, CON:, RDR:, PUN:, and LST:, starting at address 139 hex. The 16 physical device names are encoded starting at address 159 hex. You can change the names of the first two physical devices with the SID command:

```
S159                (you type this)
159 54 "CRT:LST     (you start typing with the quote)
160 3A .            (you type a period)
```

If you are using DDT, you will have to enter the hexadecimal equivalent of the ASCII characters with the S command. The ASCII characters and their corresponding hexadecimal values are as follows:

ASCII	C	R	T	:	L	S	T
Hex	43	52	54	3A	4C	53	54

You type the command S159 as with SID. Then you type the seven hex numbers in the following display:

```
159  54  43
15A  54  52
15B  59  54
15C  3A  3A
15D  43  4C
15E  52  53
15F  54  54
160  3A  .      (you finish with a period)
```

Return to CP/M and save the correct amount of memory with a new name such as STAT2.COM. Try changing the IOBYTE from 0 to 1 by giving the command

```
STAT2 CON:=LST:
```

Printer output should now duplicate the console. Disengage the printer with the complementary command

```
STAT2 CON:=CRT:
```

We could use this method to change some of the other device names in STAT.

We will now add some new features to the printer routine in BIOS.

ADDING A PRINTER-READY ROUTINE

Computers communicate with peripherals through input/output *registers* or *ports*. A common arrangement uses a bidirectional *data* register for transferring the information and a separate, bidirectional *status* register to indicate the state of readiness. With this technique, the status register is automatically reset to a not-ready condition each time the CPU places a byte in the data register.

Sometimes the CPU incorporates a special signal line for servicing peripherals. Using this line a peripheral can *interrupt* the CPU to request service. A more common method for communicating with the peripherals is called the *looping* method. With this technique, the computer checks the status register to see if the device is ready. The status register is repeatedly checked by looping through the necessary statements. When the status register indicates that the peripheral is ready, the computer performs the transfer and then goes on to something else.

Let us consider the looping method for a printer-output routine. The instructions in BIOS might look like this:

```
LIST:
LISTT:
        IN      5
        ANI     1
        JZ      LISTT
```

In this example, the status register has an address of 5 and the least significant bit is used as the ready flag. The 8080 instruction IN 5 reads the status port. The following instruction, ANI 1, performs a logical

operation on the accumulator. The result is zero if the peripheral is not ready. Consequently, the third instruction, JZ LISTT, causes a branch to the top of the three-instruction loop. Looping around the above three instructions continues until the peripheral is ready. When the ready bit indicates that the peripheral has finished its task, the instructions following JZ LISTT are executed. The computer sends another byte to the data port and then returns. The computer operates much faster than the peripheral, so much of its time is spent looping around the above three instructions.

The looping method works satisfactorily if the printer is actually turned on. Unfortunately, if the printer is turned off, the data-ready flag will usually tell the computer to send more data anyway. The computer then sends the data to a printer that is not doing anything. Therefore, we must consider two separate items—whether the printer is turned on and whether the last byte has been printed. We have been considering the latter; now we must consider the former.

There may be an easy solution to this problem. We have been looking at only one of the eight bits of the status register, the one that indicates whether the printer buffer is empty. Many computers use another bit of the status register to indicate whether the peripheral is turned on. This is called the data-terminal-ready (DTR) bit.

Locating the Bit for Data Terminal Ready

The assembly language program given in Figure 3.5 can be used to determine whether your printer status port has a DTR bit. For the standard RS-232 serial port, the DTR signal is usually assigned to pin 20. However, pin 11 is sometimes used for this purpose. Consequently, you may have to move one of the wires in the printer cable.

Create a source file with the program given in Figure 3.5. Check your BIOS or USER listing to find the address of your printer's status register, and change the value of PORT to the address of your printer's status port. Assemble the program and execute it. Remember to omit the ORG statement if you are using the Microsoft assembler. The program will read the status port and display the value on the console in binary notation. If your printer is off, turn it on; if it is on, turn it off. If any of the bits change, the new value will be printed on the screen. For some printers, it may take as long as one minute for the bit to change after the printer switch is turned off.

Continue in this way, alternately turning the printer on and off. If you find a bit changing, make a note of which bit changes and the sense of its

```
TITLE   'Display I/O port in binary'
;(Put current date here)
ORG     100H
;
PORT    EQU     5       ;status port
BDOS    EQU     5
TYPEF   EQU     2       ;console output
CSTATF  EQU     11      ;console status
CR      EQU     13      ;carriage return
LF      EQU     10      ;line feed
;
START:
        LXI     SP,STACK
        IN      PORT    ;read
        MOV     H,A     ;save
        CALL    BITS    ;show binary
NEXT:
        IN      PORT    ;next sampling
        MOV     L,A     ;save
        CMP     H       ;different?
        JNZ     SHOW    ;yes
        PUSH    H
        MVI     C,CSTATF ;console status
        CALL    BDOS
        POP     H
        RRC             ;check bit 0
        JC      0       ;quit
        JMP     NEXT
;
SHOW:
        CALL    BITS    ;show binary
        MOV     H,L     ;switch
        JMP     NEXT
;
BITS:                   ;convert binary to ASCII
        MOV     C,A
        MVI     B,8     ;8 bits
BIT2:
        MOV     A,C
```

Figure 3.5: Program to Locate the Bit for Data Terminal Ready

```
        ADD     A       ;shift left
        MOV     C,A
        MVI     A,0     ;zero
        ACI     '0'     ;carry + ASCII 0
        CALL    OUTT
        DCR     B       ;count
        JNZ     BIT2    ;8 times
CRLF:                   ;carriage return, line feed
        MVI     A,CR
        CALL    OUTT
        MVI     A,LF
OUTT:                   ;console output
        PUSH    H
        PUSH    B
        PUSH    PSW
        MVI     C,TYPEF ;console print
        MOV     E,A
        CALL    BDOS
        POP     PSW
        POP     B
        POP     H
        RET
        DS      12      ;stack space
STACK:
        END     START
```

Figure 3.5 (continued)

logic (0 or 1) when the printer is off. For example, suppose that the result is as follows:

```
10110111  (printer on)
00110111  (printer off)
```

In this example, bit 7 (the high-order bit) indicates that the printer is ready (DTR) when it is set to 1. The bit is reset to 0 when the printer is off. For this port, bit 0 indicates whether the printer buffer is empty. If the printer is turned on but busy, the bit pattern is

```
10110110  (printer on)
```

When the printer is ready to receive another byte, the pattern is

```
10110111  (printer on)
```

You can terminate the program by pressing any console key.

Let us see how this program works. We begin with the usual TITLE, ORG, and EQU directives. The status register in this example has a value of 5.

The actual instructions begin with the label START. The stack is placed at the end of the program, rather than at 100 hex as in the program shown in Figure 3.1. The status register is read into the accumulator and then moved into the H register. The value is displayed on the console by calling subroutine BITS.

The port is then read again, but this time the value is placed into the L register. The two values are compared. If they are different, the new value is displayed by calling subroutine BITS again. Then the new value is moved into the H register. If the values are the same, nothing is displayed. However, the console status is checked to see if the program is to be terminated. If not, the program loops repeatedly.

Subroutine BITS converts a binary number in the accumulator to a string of eight ASCII zeros and ones and then displays the result on the console. The routine moves the byte into the C register and initializes register B to a value of 8, the number of characters to be displayed.

The loop beginning at BIT2 is then executed eight times. On each pass through the loop, the current value of the byte is added to itself with the ADD A instruction. This action performs a logical shift left. The bits of the accumulator are each moved one position. The original high-order bit moves into the carry flag. The low-order bit is zeroed. The new value is saved in the C register for the next step.

At this point, the carry flag is set to 1 if the original high-order bit had a value of 1. It is reset to 0 if the value was 0. We will display the value of 1 if the carry flag is set; we will display a 0 otherwise. This is accomplished by zeroing the accumulator. We then add an ASCII zero and the carry flag. The instructions are as follows:

```
MVI   A,0   ;zero accumulator
ACI   '0'   ;carry + ASCII zero
```

Let us go through the first two loops of the algorithm with an example. Consider the binary number 10101010 (AA hex). When this number is added to itself, the result is 01010100 and the carry flag is set to 1. Our algorithm will display a 1. The next addition will produce the binary number 10101000 and reset the carry flag to 0. The routine displays a 0 this time.

This algorithm can be used with both an 8080 and a Z80 CPU, but it can be implemented more effectively on a Z80 computer by performing the logical shift directly in the C register. All of the common algorithms for

base conversion can be found elsewhere.*

If you found a printer-ready bit, the next section will show you how to incorporate a test for DTR into your BIOS.

Checking for Printer Ready

As noted above, not all computers incorporate a DTR bit. However, if you have discovered a printer-ready bit, you can include a test in your BIOS that will notify you when there is printer output but the printer is turned off. This test checks the printer-ready flag. If it indicates that the printer is off, the console bell will sound and an appropriate message will be displayed. When the printer is turned on, the instruction following this portion (the usual test that the printer buffer is empty) will be executed.

Suppose that the printer status port is given the name LSTATP, the data-terminal-ready mask is given the name DTRMSK, and the regular port-ready mask is called LMSK. The physical console-output routine is referenced as CONO2, because we want to distinguish physical console output from logical console output. The original list routine might look like that in Figure 3.6, while the new version will look like that in Figure 3.7.

The first three lines of the new version define the symbols CR (carriage return), LF (line feed), and BEL (console bell). Then the executable code begins. The printer status port (LSTATP) is read. All of the bits, except for the DTR bit, are zeroed with the ANI DTRMSK instruction. If this bit is set, the zero flag will be reset. The instruction

```
JNZ   LIST2
```

branches to LIST2, the original printer-output routine.

But if the DTR bit is reset to 0, it indicates that the printer is turned off. In this case the console bell sounds and the message

```
TURN PRINTER ON
```

is displayed on the console. The status port is monitored again starting with the label LIST3. The program continually loops around the next three instructions until the printer is turned on. At that time, the program continues with the printer-output routine.

Incorporate the new passage into your BIOS. Assemble it and load it with the debugger. Engage the printer with control-P and give the DIR command. While the printer is working, turn it off. The console bell will

*A. R. Miller, *8080/Z80 Assembly Language: Techniques for Improved Programming,* New York: Wiley, 1981.

```
LIST:                              ;logical list output
LISTT:
        IN      LSTATP      ;check status
        ANI     LMSK        ;mask for output
        JZ      LISTT       ;loop until ready
        MOV     A,C
        OUT     LDATAP      ;send
        RET
```

Figure 3.6: Original Version of a Typical Printer Routine

sound and the message

TURN PRINTER ON

will appear on the console. When the printer is turned on again, the output should take up where it left off. This routine will work correctly even within programs such as WordStar and BASIC (except, of course, that different commands are used to engage the printer).

DIRECTING LIST OUTPUT WITH THE IOBYTE

Earlier in this chapter we incorporated the IOBYTE into the console-output routine. That feature used the two low-order bits of the IOBYTE. We will now add several new features to the logical list output using the two high-order bits of the IOBYTE.

One of the features we will add is relatively easy to install. Sometimes called a "bit bucket," this routine is useful when a program with a long output must be tested, but the output itself is not wanted. In addition to this, we will be able to direct the list output to the printer, as is usually the case, to the console, or to a separate memory area.

We reserve an IOBYTE value of 0 for the usual output to the printer. The value of 40 hex sends list output to the console, and the value of 80 hex discards the data—that is, the data disappear. An IOBYTE value of C0 hex will be allocated at this time for storing list output in a separate memory area called a cache. However, we will not actually add the routine until later. The list assignments follow; they should be coded into the

```
CR      EQU     13              ;carriage return
LF      EQU     10              ;line feed
BEL     EQU     7               ;ASCII bell
;
LIST:
LISTT:
        IN      LSTATP          ;check status
        ANI     DTRMSK          ;printer on?
        JNZ     LIST2           ;yes
        PUSH    H               ;printer off
        PUSH    B
        LXI     H,MESG          ;location
        MVI     B,AROUND-MESG   ;length
LLOOP:
        MOV     C,M
        CALL    CONO2           ;send to console
        INX     H               ;pointer
        DCR     B               ;count
        JNZ     LLOOP           ;keep going
        POP     B
        POP     H
        JMP     AROUND          ;the message
MESG:
        DB      BEL,CR,LF
        DB      ' TURN PRINTER ON ',CR,LF
AROUND:
LIST3:
        IN      LSTATP
        ANI     DTRMSK          ;printer on?
        JZ      LIST3           ;no
LIST2:
        IN      LSTATP          ;check status
        ANI     LMSK            ;mask for output
        JZ      LISTT           ;loop until ready
        MOV     A,C
        OUT     LDATAP          ;send
        RET
```

Figure 3.7: Revised Version of a Typical Printer Routine

BIOS source program as comments.

IOBYTE	*Action*
00	Printer output
40	Console output
80	Bit bucket
C0	Memory cache

The list output routine begins with the following statements:

```
LIST:                       ;logical
LISTT:                      ;physical
        IN      LSTATP      ;check status
        . . .   . . .
```

LIST refers to the logical output and LISTT refers to the physical printer. We will now insert instructions between these two labels.

We must include a test of the IOBYTE at the beginning of the list-output routine, just as we did for the console-output routine. The new instructions will be placed between the labels LIST and LISTT. First we read the IOBYTE. Then, because we are only interested in the two high-order bits, we perform a masking AND with the value of C0 hex. This operation zeros the six low-order bits. If the result is 0, output is sent to the printer. If the result is 40 hex, output is sent to the console. If the result is 80 hex, the subroutine simply returns to the calling program—that is, no action is performed. The last possibility, C0 hex, indicates that list output is to be stored in a memory cache. We will not incorporate this feature now, so we will simply return to the calling program. The source program for this feature is shown in Figure 3.8.

Notice that when the value of the IOBYTE is 40 hex, the list output is sent to the label CONO2 rather than to the logical console-output label of CONOUT. This ensures that list output destined for the console will not be diverted back to the printer if the low-order bit of the IOBYTE is set.

Assemble these instructions into your BIOS or USER area. Load the new version into memory with the debugger and try it out. Change the IOBYTE with the debugger, setting it to a value of 40 hex. Engage the list output with control-P. Each character should now be displayed twice on the console, because both the logical console and logical list are directed to the physical console. Disengage the list with another control-P. If you are satisfied with the new version, use SAVEUSER or SYSGEN to save a copy on the system tracks of a diskette.

We will now add a routine to store the list output in a memory cache.

```
LIST:                              ;logical list output
        LDA     IOBYTE
        ANI     0C0H               ;mask for bits 6,7
        JZ      LISTT              ;printer output
        CPI     40H
        JZ      CONO2              ;console output
        CPI     80H
        RZ                         ;bit bucket
;
;add memory cache routine here
;
        RET                        ;(for now)
;
LISTT:                             ;physical list output
        . . .   . . .
```

Figure 3.8: Incorporating the IOBYTE into Printer Output

STORING LIST OUTPUT IN A MEMORY CACHE

There are times when it is desirable to store the list output in a memory buffer or *cache* rather than send it to the printer. The result can then be saved as a disk file for editing or for incorporation into a report. In fact, all of the computer outputs in this book were obtained in this manner.

The operation of the memory cache is managed with two pointers. The first pointer indicates where the next byte is to be placed. This pointer is initially set to the beginning of the buffer and is incremented each time a byte is added to the buffer. At the conclusion of the task a 1A hex, end-of-file mark is placed at the end of the text, the second pointer is set to the end of the file, and the first pointer is reset to the beginning of the buffer. The two pointers are stored immediately in front of the buffer; they are each two-byte values.

We must choose a region for the buffer area that will never be used by the CP/M operating system. Otherwise, the cache may be accidentally overwritten. There are several ways to accomplish this. For example, a North Star Horizon computer uses the region from E800 to EBFF hex for the disk-controller memory. Because CP/M requires a contiguous block of memory, the maximum CP/M address for this machine is E7FF hex. Therefore, the memory region from F000 to FFFF hex is free. Another

possibility is to create a smaller CP/M system with MOVCPM. The region of memory above CP/M can then be used for the memory cache.

In the previous section we allocated the IOBYTE value of C0 hex to indicate that list output will be stored in a memory buffer. We will now write the routines necessary for this feature. We select the region F000 to FFFF (the top 4K bytes) as the memory block. The two pointers are stored at F000 and F002 hex. The memory buffer itself begins at F004 hex.

There is another complication we should consider. The buffer will overflow if too many bytes are entered into it. The pointer will attempt to go beyond the end of the buffer, address FFFF hex in this case. When FFFF hex, the largest 16-bit number, is incremented, the result is 0. Thus, the pointer now refers to the beginning of memory rather than the end. (This phenomenon is known as wrap around.) As we saw in Chapter 1, CP/M maintains several important items at the beginning of memory. Consequently, we must ensure against wrap around and the consequent alteration of important CP/M information.

We will reset the pointer to the beginning of the buffer and ring the console bell if wrap around is imminent. This action protects the CP/M system. Of course, the information initially placed into the cache is then lost, but this is not likely to be a problem. You will find that a 4K-byte buffer will be sufficiently large for most purposes.

At the end of the task, we can use the system debugger to move the information from the memory cache down to the TPA at 100 hex. We then return to CP/M with control-C and save the information in a disk file. In Chapter 7 we will write a program that can automatically write a disk file directly from the memory cache. This program uses the buffer pointers to determine the file size.

We need two separate sets of instructions to implement the memory cache. One portion initializes the pointers and sets the end-of-file marker. These instructions are placed in the warm-start and cold-start areas of BIOS or USER. Instructions for the second part place each byte into the memory cache and increment the main pointer. This portion is located with the list-output routines. We begin with the routines that initialize the pointers.

Initializing the Memory Cache Pointers

In this section we alter the warm-start and cold-start areas of BIOS or USER to insert the instructions for initializing the cache pointers and adding the end-of-file marker. We first define four symbols—the names of the two pointers, the name of the buffer, and the end-of-file reference.

Place the following four lines near the top of the source program:

```
MPOINT  EQU  0F000H      ;pointer to beginning
MMAX    EQU  MPOINT+2    ;pointer to end
MBUFF   EQU  MMAX+2      ;buffer start
EOF     EQU  1AH         ;end-of-file mark
```

Locate the warm start vector of your BIOS or USER. Remember, this can be found from the second jump vector. Follow the warm-start instructions until the final return statement is encountered. Place the instructions shown in Figure 3.9 just before this return statement.

Let us see how this segment works. We begin by checking whether the logical list output is being directed to the memory cache option. This information is coded into the two high-order bits of the IOBYTE. The first new instruction copies the value of the IOBYTE into the accumulator:

```
LDA   IOBYTE
```

All but the two high-order bits are zeroed with the instruction

```
ANI   0C0H
```

If the result is not C0 hex, then we complete the warm start with a return instruction.

On the other hand, if the result is C0 hex, the cache option has been selected. The HL and DE registers are then saved with PUSH instructions. Then we determine if the pointer is already reset to the beginning of the buffer. If so, the task is complete. The HL and DE registers are restored by POP instructions and a return is executed.

If the pointer has not been reset, it points to the buffer end. An end-of-file marker (1A hex) is placed at this point. The address of the buffer end is saved in the second pointer (MMAX) and the main pointer (MPOINT) is reset to the beginning of the buffer. The registers are restored and a return is executed.

It will also be necessary to initialize the buffer pointer during the cold start, so we must locate the cold-start entry. It is referenced by the first vector at the beginning of the BIOS or USER. In an earlier section of this chapter, we added two instructions to initialize the IOBYTE during a cold start. Place the following two instructions immediately after these.

```
COLD:
        . . .
        LXI     H,MBUFF
        SHLD    MPOINT
```

Now we will incorporate the remainder of the cache instructions.

```
WARM:
        . . .
        LDA     IOBYTE
        ANI     0C0H        ;mask for list
        CPI     0C0H        ;memory?
        RNZ                 ;no, leave alone
        PUSH    H           ;save registers
        PUSH    D
        LXI     D,MBUFF     ;buffer start
        LHLD    MPOINT      ;pointer
        MOV     A,L         ;check low
        CMP     E           ;pointers reset?
        JNZ     MEM3        ;no
        MOV     A,H         ;check high
        CMP     D           ;reset?
        JZ      MEM4        ;yes
MEM3:                       ;reset pointers
        MVI     M,EOF       ;end of file mark
        SHLD    MMAX        ;save last address
        LXI     H,MBUFF     ;buffer start
        SHLD    MPOINT      ;save pointer
MEM4:
        POP     D           ;restore
        POP     H
        RET                 ;original
```

Figure 3.9: Setting up the Memory Pointers

Instructions for Storing List Output in Memory

Now that we have added the instructions for initializing the memory pointers, we can incorporate the code for actually storing the data in memory. The new instructions, shown in Figure 3.10, are placed between the RZ and RET instructions in the list-output region shown in Figure 3.8.

This section has two parts. The first part stores each byte in memory and advances the memory pointer. The second part checks for wrap around. We begin by saving the contents of the HL register with a PUSH instruction. The main pointer is retrieved and used to deposit the byte in memory. The pointer is incremented and then checked to ensure that it is not wrapping around zero.

If wraparound did not occur, the pointer is updated and a return is executed. On the other hand, if the pointer has a value of 0, it is reset to the beginning of the buffer and the console bell sounds.

```
;
;send list output to a memory cache
;
            PUSH    H               ;save
            LHLD    MPOINT          ;pointer
            MOV     M,C             ;put byte in memory
            INX     H               ;increment pointer
            MOV     A,H             ;see if
            ORA     L               ;passing zero
            JNZ     MEM2            ;ok to continue
;
;buffer is wrapping around zero; reset it
;and sound console bell as a warning
;
            PUSH    D
            MVI     C,BEL
            CALL    CONO2           ;ring bell
            POP     D
            LXI     H,MBUFF         ;start
MEM2:                               ;update pointer
            SHLD    MPOINTER        ;save it
            POP     H
            RET
```

Figure 3.10: Storing List Output in Memory

Incorporate the remainder of the instructions for the memory cache into the BIOS. Assemble the new version and test it. Load the program into place with the debugger.

It is extremely important that the main pointer is correctly set before you use the cache. Otherwise, CP/M will deposit bytes in the wrong place with unpredictable results. The two instructions we added to the cold-start section will initialize the main pointer each time you start up CP/M. However, we want to test the routines before they are written to the system tracks of the disk. Therefore, for this one time, we will have to initialize the main pointer.

Use the debugger S command to set the main pointer to F004 hex. The instructions are as follows:

```
SF000           (you type this line)
F000 XX 4       (you type 4)
F001 XX F0      (you type F0)
F002 XX .       (you type a period)
```

Set the IOBYTE to a value of C0 hex, again using the S command:

```
S3              (you type this line)
0003 X C0       (you type C0)
0004 X .        (you type a period)
```

Perform a warm start by typing control-C. You are now at the CP/M system level. Engage the list output by typing control-P, then give the command DIR. No output should appear at the printer, because we are diverting list output to the memory cache. Perform another warm start by typing control-C. This disengages the list output and resets the pointers.

Load the debugger and inspect the beginning of the buffer with the D command:

```
DF000,F03F
```

The ASCII representation of the previous DIR output should appear on the right side of the screen. Look at the second pointer stored at F002 and F003. This pointer references the end of the text. The corresponding memory location should contain a 1A hex end-of-file mark.

You can now use the debugger M command to move the information in the cache down to 100 hex. Perform a warm start and save the information on a disk file. You should now use SAVEUSER or SYSGEN to write the current version of BIOS or USER to the system tracks of a floppy diskette. Turn the computer off and on again; perform a cold boot with the new version. Use the debugger to check the main cache pointer, to be sure that it is properly initialized.

An assembly listing of a set of USER routines is shown in Figure 3.11. This listing incorporates all the features described in this chapter. It operates on a Lifeboat version 2.2 CP/M running on a 56K-byte North Star system. Several key features will have to be changed if it is to be used on other systems.

```
            TITLE   'Sample BIOS/USER program'
            ;
            ;(Current date)
            ;
DA00        ORG     0DA00H
            ;
0003 =      IOBYTE  EQU     3
```

Figure 3.11: USER Routines for a 56K-Byte Lifeboat Version 2.2 CP/M for North Star

```
0003 =      CSTATP  EQU     3               ;console status
0002 =      CDATAP  EQU     CSTATP-1        ;console data
0001 =      COMSK   EQU     1               ;console-output mask
0002 =      CIMSK   EQU     2               ;console-input mask
0005 =      LSTATP  EQU     5               ;list status
0004 =      LDATAP  EQU     LSTATP-1        ;list data
0001 =      LMSK    EQU     1               ;list-output mask
0080 =      DTRMSK  EQU     80H             ;list-ready mask
F000 =      MPOINT  EQU     0F000H          ;pointer to beginning
F002 =      MMAX    EQU     MPOINT+2        ;pointer to end
F004 =      MBUFF   EQU     MMAX+2          ;buffer start
            ;
000D =      CR      EQU     13              ;carriage return
000A =      LF      EQU     10              ;line feed
0007 =      BEL     EQU     7               ;ASCII bell
001A =      EOF     EQU     1AH             ;end-of-file mark
            ;
            START:
DA00 C399DA         JMP     COLD            ;initial cold start
DA03 C3A5DA         JMP     WARM            ;warm-start reset
DA06 C3CDDA         JMP     CSTAT           ;console status
DA09 C3D5DA         JMP     CONIN           ;console input
DA0C C318DA         JMP     CONOUT          ;console output
DA0F C32BDA         JMP     LIST            ;printer output
DA12 C3E0DA         JMP     PUNCH           ;punch output
DA15 C3E1DA         JMP     READER          ;alternate input device
            ;
            CONOUT:                         ;console output
DA18 3A0300         LDA     IOBYTE          ;get the value
DA1B E601           ANI     1               ;mask for bit 0
DA1D C455DA         CNZ     LISTT           ;printer output
            CONO2:                          ;regular console output
DA20 DB03           IN      CSTATP          ;read status
DA22 E601           ANI     COMSK           ;mask for output
DA24 CA20DA         JZ      CONO2           ;loop until ready
DA27 79             MOV     A,C             ;get byte
DA28 D302           OUT     CDATAP          ;send
DA2A C9             RET
            ;
```

Figure 3.11 (continued)

```
        LIST:                                   ;logical list output
DA2B 3A0300     LDA     IOBYTE
DA2E E6C0       ANI     0C0H            ;mask for bits 6,7
DA30 CA55DA     JZ      LISTT           ;printer output
DA33 FE40       CPI     40H
DA35 CA20DA     JZ      CONO2           ;console output
DA38 FE80       CPI     80H
DA3A C8         RZ                      ;bit bucket
        ;
        ;send list output to a memory cache
        ;
DA3B E5         PUSH    H               ;save
DA3C 2A00F0     LHLD    MPOINT          ;pointer
DA3F 71         MOV     M,C             ;put byte in memory
DA40 23         INX     H               ;increment pointer
DA41 7C         MOV     A,H             ;see if
DA42 B5         ORA     L               ;passing zero
DA43 C250DA     JNZ     MEM2            ;ok to continue
        ;
        ;buffer is wrapping around zero; reset it
        ;and sound console bell as a warning
        ;
DA46 D5         PUSH    D
DA47 0E07       MVI     C,BEL
DA49 CD20DA     CALL    CONO2           ;ring bell
DA4C D1         POP     D
DA4D 2104F0     LXI     H,MBUFF         ;start
        MEM2:                           ;update pointer
DA50 2200F0     SHLD    MPOINT          ;save it
DA53 E1         POP     H
DA54 C9         RET
        ;
        LISTT:                          ;physical printer output
DA55 DB05       IN      LSTATP          ;check status
DA57 E680       ANI     DTRMSK          ;printer on?
DA59 C28EDA     JNZ     LIST2           ;yes
DA5C E5         PUSH    H               ;printer off
DA5D C5         PUSH    B
DA5E 2171DA     LXI     H,MESG          ;location
```

Figure 3.11 (continued)

```
DA61 0616          MVI   B,AROUND-MESG  ;length
        LLOOP:
DA63 4E            MOV   C,M
DA64 CD20DA        CALL  CONO2           ;send to console
DA67 23            INX   H               ;pointer
DA68 05            DCR   B               ;count
DA69 C263DA        JNZ   LLOOP           ;keep going
DA6C C1            POP   B
DA6D E1            POP   H
DA6E C387DA        JMP   AROUND          ;the message
        MESG:
DA71 070D0A        DB    BEL,CR,LF
DA74 205455        DB    ' TURN PRINTER ON ',CR,LF
        AROUND:
        LIST3:
DA87 DB05          IN    LSTATP
DA89 E680          ANI   DTRMSK          ;printer on?
DA8B CA87DA        JZ    LIST3           ;no
        LIST2:
DA8E DB05          IN    LSTATP          ;check status
DA90 E601          ANI   LMSK            ;mask for output
DA92 CA55DA        JZ    LISTT           ;loop until ready
DA95 79            MOV   A,C
DA96 D304          OUT   LDATAP          ;send
DA98 C9            RET
        ;
        COLD:                            ;cold-start entry
DA99 3E00          MVI   A,0
DA9B 320300        STA   IOBYTE          ;reset
DA9E 2104F0        LXI   H,MBUFF
DAA1 2200F0        SHLD  MPOINT          ;reset
DAA4 C9            RET
        ;
        WARM:                            ;warm-start entry
DAA5 3A0300        LDA   IOBYTE
DAA8 E6C0          ANI   0C0H            ;mask for list
DAAA FEC0          CPI   0C0H            ;memory?
DAAC C0            RNZ                   ;no, leave alone
DAAD E5            PUSH  H               ;save registers
```

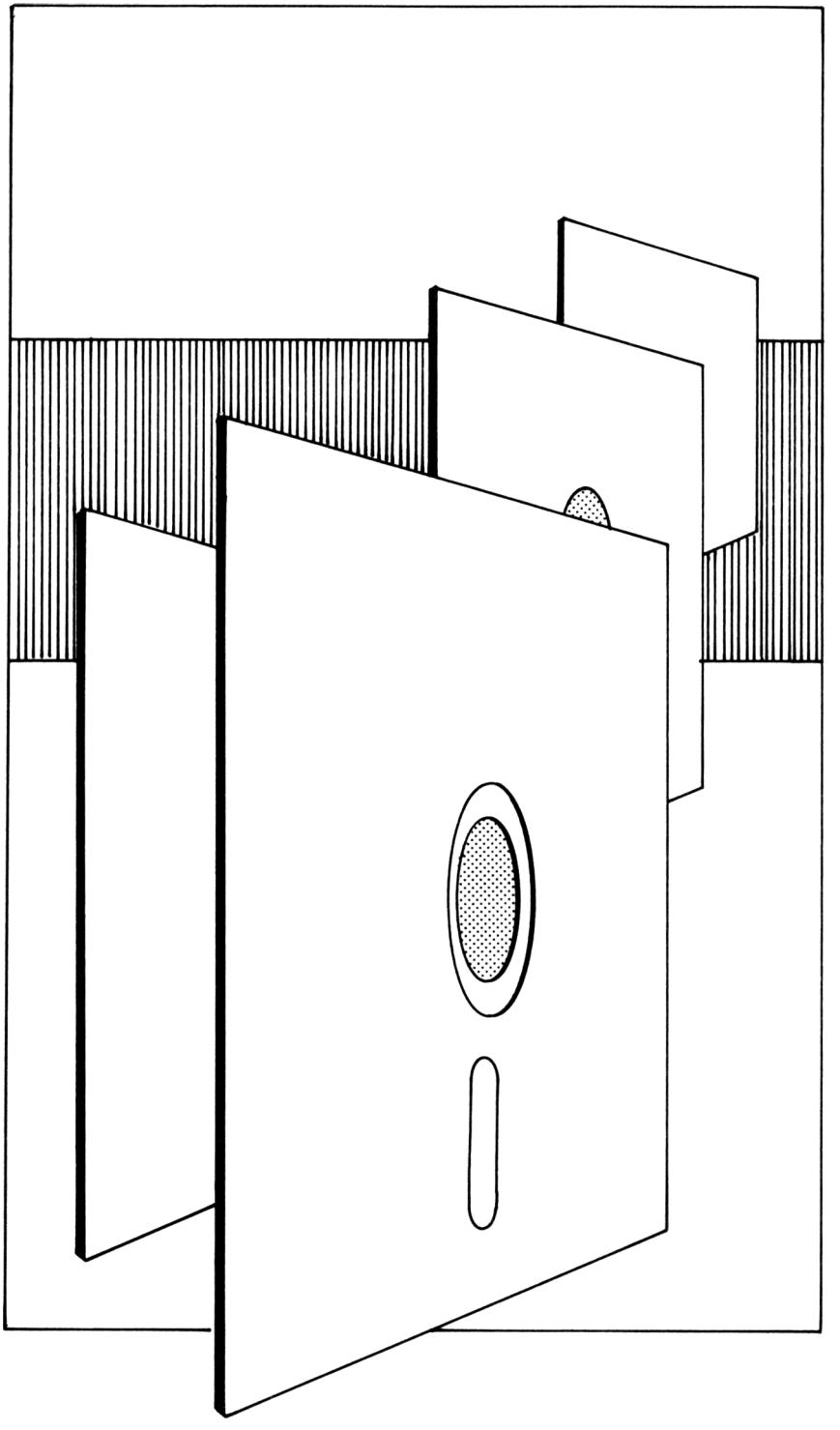
Figure 3.11 (continued)

```
DAAE D5          PUSH   D
DAAF 1104F0      LXI    D,MBUFF          ;buffer start
DAB2 2A00F0      LHLD   MPOINT           ;pointer
DAB5 7D          MOV    A,L              ;check low
DAB6 BB          CMP    E                ;pointers reset?
DAB7 C2BFDA      JNZ    MEM3             ;no
DABA 7C          MOV    A,H              ;check high
DABB BA          CMP    D                ;reset?
DABC CACADA      JZ     MEM4             ;yes
          MEM3:                          ;reset pointers
DABF 361A        MVI    M,EOF            ;mark end of buffer
DAC1 2202F0      SHLD   MMAX             ;save last address
DAC4 2104F0      LXI    H,MBUFF          ;buffer start
DAC7 2200F0      SHLD   MPOINT           ;save pointer
          MEM4:
DACA D1          POP    D                ;restore
DACB E1          POP    H
DACC C9          RET                     ;original ret.
          ;
          ;necessary routines not discussed in text
          ;
          CSTAT:                         ;console input status
DACD DB03        IN     CSTATP           ;read status
DACF E602        ANI    CIMSK            ;mask for input
DAD1 C8          RZ                      ;not ready
DAD2 3EFF        MVI    A,0FFH
DAD4 C9          RET                     ;ready
          CONIN:
DAD5 CDCDDA      CALL   CSTAT
DAD8 CAD5DA      JZ     CONIN            ;not ready
DADB DB02        IN     CDATAP           ;get byte
DADD E67F        ANI    7FH              ;mask parity
DADF C9          RET
          PUNCH:
DAE0 C9          RET
          READER:
DAE1 C9          RET
          ;
DAE2             END
```

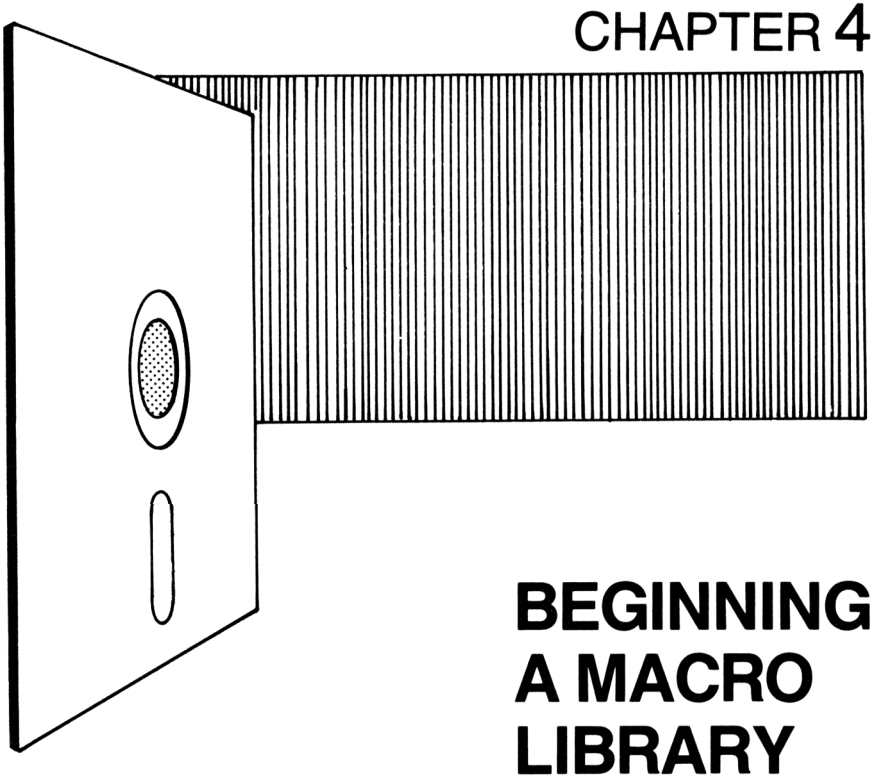
Figure 3.11 (continued)

SUMMARY

In this chapter, we have explored the CP/M BIOS and USER routines in greater detail. We have developed and implemented several useful features to increase the power of our CP/M operating system, including routines to engage and disengage the printer, a printer-ready routine, and a routine to direct the list output to a memory cache. In addition to these, you may consider incorporating other features such as sending logical punch output to a telephone modem or taking console input from the printer keyboard. These will be left as exercises.



CHAPTER 4



BEGINNING A MACRO LIBRARY

INTRODUCTION

In this chapter we will introduce the concept of macro instructions, also called macros. We will develop several powerful macros that will be used in the remainder of this book. We begin with housekeeping macros that incorporate the version number and save and restore the stack pointer. We will then write macros that move information, fill memory with a constant, compare information, convert lowercase letters to uppercase, perform 16-bit subtraction, and convert an ambiguous file name to an unambiguous name.

MACROS

A macro instruction, or macro, is an assembler directive that defines a collection of other commands, instructions, or macros. A macro actually consists of two parts—the definition and one or more implementations or

expansions. The name of the macro is associated with the set of instructions it defines. Whenever the macro name appears in a computer program, the macro assembler substitutes the corresponding instructions. This is called the macro expansion. For example, the following sequence of instructions can be defined by a macro named SAVE:

```
PUSH    H
PUSH    D
PUSH    B
PUSH    PSW
```

Then, whenever the name SAVE appears in the computer program, the corresponding four instructions will be substituted. A complementary macro named UNSAVE can perform the inverse operations:

```
POP     PSW
POP     B
POP     D
POP     H
```

The macro definition is placed near the top of the program or in a separate disk file called a macro library. The first line of the macro defines the macro name. The middle portion, which contains the instructions, is usually called the macro body. The last line terminates the macro with the statement ENDM. You must always remember to include the ENDM statement at the conclusion of the macro definition. If ENDM is omitted, the remainder of the program is incorrectly interpreted as part of a very large macro. Most macro assemblers are confused by this omission and issue cryptic error statements.

Macro definitions for the above examples would look like this:

```
SAVE    MACRO
        PUSH    H
        PUSH    D
        PUSH    B
        PUSH    PSW
        ENDM

UNSAVE  MACRO
        POP     PSW
        POP     B
        POP     D
        POP     H
        ENDM
```

Macro Parameters

Macros become more versatile with the addition of parameters. For example, suppose we want to interchange the contents of the H and L registers using the accumulator as a working register. A macro to perform this task might appear as follows:

```
INTER   MACRO
        PUSH    PSW
        MOV     A,H
        MOV     H,L
        MOV     L,A
        POP     PSW
        ENDM
```

Now whenever the macro name INTER appears in the program, the assembler substitutes the corresponding five instructions:

```
PUSH    PSW
MOV     A,H
MOV     H,L
MOV     L,A
POP     PSW
```

Notice that this macro will always generate instructions to interchange the H and L registers. However, if we change the macro slightly by adding two parameters, the macro becomes more versatile. For example, the following macro is similar to INTER except that the dummy parameters* REG1? and REG2? are given on the first line. (The question marks in the parameters are considered to be regular characters.)

```
INTER2  MACRO   REG1?,REG2?
        PUSH    PSW
        MOV     A,REG1?
        MOV     REG1?,REG2?
        MOV     REG2?,A
        POP     PSW
        ENDM
```

The assembler substitutes the actual parameters for the dummy

*Dummy parameters are sometimes called formal parameters. However, there appears to be some confusion in this usage, as the actual parameters are also sometimes referred to as formal parameters.

parameters. For example, the statement

```
INTER2   H,L
```

is assembled into the same five statements we got with the previous macro. However, the expression

```
INTER2   D,E
```

will generate the following instructions:

```
PUSH     PSW
MOV      A,D
MOV      D,E
MOV      E,A
POP      PSW
```

Macros and Conditional Assembly

Conditional assembly statements further increase the power of macros. For example, the following pair of statements can be used to test for the presence of an optional parameter corresponding to the dummy parameter PARAM?:

```
IF       NUL PARAM?
. . .
ENDIF
```

The expression NUL PARAM? is true if a parameter is not provided; it is false otherwise. Of course, the complementary expression

```
IF   NOT NUL PARAM?
```

can be used to reverse the sense of the expression; that is, the expression is true if a parameter *is* provided.

The Microsoft assembler also accepts the alternate forms

```
IFDEF    PARAM?
. . .
ENDIF
```

and

```
IFNDEF   PARAM?
. . .
ENDIF
```

for IF NOT NUL and IF NUL. The expressions IFDEF and IFNDEF respectively mean "if defined" and "if not defined."

For some programs we will want to execute a return statement when we are finished. On other occasions, however, we will branch to a specific address. For example, consider the following fragments of macro EXIT, which we will develop shortly:

```
EXIT    MACRO   WHERE?
        . . .
        . . .
        IF      NUL WHERE?
        RET
        ELSE
        JMP     WHERE?
        ENDIF
        . . .
        ENDM
```

Parameter WHERE? is optional in this example. Suppose that macro EXIT is used without this parameter:

```
EXIT
```

A simple return statement will be created in this case, because the expression IF NUL WHERE? is true. However, if a parameter is included, then a branch to the parameter is generated. Thus the macro reference

```
EXIT    BOOT
```

will generate the instruction

```
JMP     BOOT
```

Before we begin our macro library, let us first consider the generation of Z80 intructions by using macros and an 8080 macro assembler.

GENERATING Z80 INSTRUCTIONS WITH AN 8080 ASSEMBLER

The Z80 CPU can execute all of the 8080 instructions; consequently, an 8080 assembler is commonly used for generating assembly language programs to run on a Z80 computer. The Digital Research macro assembler,

called MAC, uses the Intel 8080 mnemonic instructions. The Microsoft macro assembler, MACRO-80, can assemble either the Intel 8080 or the Zilog Z80 mnemonics. Throughout this book we will use primarily the 8080 mnemonics. Consequently, either of these macro assemblers will be suitable.

However, there are several powerful Z80 instructions that are sometimes useful when writing assembly language programs. An 8080 assembler can generate these instructions with macros. In fact, the Digital Research macro assembler is supplied with a set of macros for this purpose.

For example, suppose that we must subtract one number from another. This operation can be performed by taking the two's complement of the first number and then adding the result to the second number. There is a Z80 instruction that can perform this operation; the mnemonic is NEG.

The 8080 instruction set does not explicitly incorporate this operation, but it can be performed by combining two 8080 instructions. The two's complement can be obtained by incrementing the one's complement. Because there is an 8080 mnemonic for performing a one's complement and another for incrementing the result, we can combine these two operations into a macro. The macro definition is as follows:

```
NEG   MACRO       ;two's complement
      CMA         ;;one's complement
      INR     A
      ENDM
```

Whenever the two's complement is needed, the macro

```
NEG
```

is placed into the source program. The 8080 assembler will substitute the corresponding instructions:

```
CMA
INR   A
```

Notice that the comment in the first statement begins with two semicolons rather than the usual one:

```
CMA               ;;one's complement
```

This has a special meaning in macro definitions. When a comment beginning with a single semicolon appears in a macro definition, the comment is reproduced at each expansion of the macro. However, if a comment begins with a double semicolon, it is not written at each expansion. Because the first and last lines of the macro are not reproduced at each

expansion, the comments on these lines can be written with one semicolon.

Again, notice how the above macro becomes more versatile with the addition of a parameter. Suppose that we change the definition of the previous macro to look like this:

```
NEG     MACRO   REG?            ;two's complement
;;
        IF      NOT NUL REG?
        PUSH    PSW             ;;save A
        MOV     A,REG?          ;;get register
        ENDIF
        CMA                     ;;one's complement
        INR     A
        IF      NOT NUL REG?
        MOV     REG?,A          ;;return value
        POP     PSW             ;;restore A
        ENDIF
        ENDM
```

The macro reference NEG will generate the same two instructions as the previous version did, because no parameter was included in the macro reference. However, if a parameter is included in the expression, the result is different. For example, the expression

```
NEG     C
```

contains the parameter C. This time the resulting assembly code will be as follows:

```
PUSH    PSW
MOV     A,C
CMA
INR     A
MOV     C,A
POP     PSW
```

That is, the single macro statement NEG C produces six lines of instructions rather than two. During the macro expansion, the dummy parameter REG? is replaced with the parameter C. The conditional passage

```
IF      NOT NUL REG?
. . .
ENDIF
```

will generate instructions only if a parameter is included in the calling statement. Otherwise, the section between IF and ENDIF will be omitted.

THE 8080/Z80 SWITCH

Even though the Z80 computer is very popular, there are many 8080 and 8085 computers in use. There is also a combination CPU card that contains both an 8085 and an 8088 CPU. (The 8085 CPU can execute all of the 8080 instructions but none of the Z80 instructions that are not common to the 8080.) Consequently, it may be necessary to use 8080 code on one occasion, while the more efficient Z80 code can be used at other times. This is easily accomplished with macros and conditional statements.

A Z80 flag can be defined at the beginning of the program. For example, the statement

```
Z80M    EQU    TRUE    ;Z80 mode flag
```

is used to indicate that Z80 code is desired. Otherwise, the statement

```
Z80M    EQU    FALSE   ;Z80 mode flag
```

is used. (Of course, the symbols TRUE and FALSE must be defined separately.) The macro will generate either Z80 or 8080 code, depending on the definition of the Z80M flag.

As an example, let us consider the unconditional relative jump. Sometimes we need to transfer control (branch) to a different portion of a program. In this case we use an *unconditional* jump instruction. With the Z80 we have a choice of either a *relative* unconditional branch to a location a certain distance away from the present position or an *absolute* unconditional branch to a fixed address. The relative jump is usually preferred because the instruction is shorter than the absolute jump. However, the 8080 CPU cannot perform the relative jump. Thus we might wish to use the relative jump with a Z80 but an absolute jump with an 8080.

We can write a dual macro using conditional assembly statements so that we can generate the Z80-compatible instruction for one application and the 8080-compatible instruction on other occasions. For example, we can define a relative jump macro as follows:

```
JR      MACRO   ADDR?
        IF      Z80M
        DB      18H, ADDR? -$ -1
        ELSE
        JMP     ADDR?
        ENDIF
        ENDM
```

If the Z80 flag is true, then the macro reference

```
JR      DONE
```

will generate the two bytes corresponding to the desired Z80 code:

```
DB    18H, DONE -$ -1
```

Otherwise, the three-byte 8080 instruction

```
JMP   DONE
```

will be generated.

As another example, consider the Z80 mnemonic DJNZ. This instruction decrements the B register, then jumps relative to the operand if the zero flag is reset.* The dual macro might look like this:

```
DJNZ  MACRO   ADDR?
      IF      Z80M
      DB      10H, ADDR? -$ -1
      ELSE
      DCR     B
      JNZ     ADDR?
      ENDIF
      ENDM
```

The Z80 version of the macro reference

```
DJNZ  LOOP
```

will assemble into

```
DB    10H, LOOP -$ -1
```

for the corresponding Z80 instruction. On the other hand, the 8080 mode produces the lines

```
DCR   B
JNZ   LOOP
```

The resulting assembled code is fixed. It will perform the same way each time it is executed. This is a very different concept from a Pascal or BASIC expression such as

```
IF A = B THEN . . .
```

With this BASIC statement, one set of instructions might be executed if the statement is true. However, another set could be executed if the statement is false.

Before beginning the macro library, let us briefly summarize the concept

*Remember that a flag is *reset* or *false* when zero and *set* or *true* otherwise.

of macros. A macro assembler will analyze the source program by reading it several times. Each reading is called a pass. On one pass, the part of the assembler that processes the macros converts the macro references into the desired instructions. For example, we saw that the macro NEG generates the following two instructions:

```
CMA
INR     A
```

On the next pass, the assembler analyzes the instructions created by the macro processor as though the instructions had been included in the original source program. The resulting binary code will be the same whether or not macros were used.

STARTING THE MACRO LIBRARY

In this chapter and those that follow we are going to create a disk file of useful macros. This macro "library" will be used in many of the programs we will develop. If we place a copy of each macro in each program, there will be much duplication. Therefore, we will find it more convenient to place all the macros in a separate macro library. We can then simply refer to them from each program.

Another advantage of the macro library is that it can greatly simplify program revision. Suppose you have to change a macro that is used in many different programs. If the macro were coded into each program, you would have to change each occurrence. However, if the macro appears only once in the macro library, only that one copy has to be changed.

Let us begin our macro library with a heading and some useful symbols.

Commonly Used Constants

There are several values we will need in almost all our programs. These include the characters such as carriage return, line feed, and blank. It will be more convenient to refer to these values symbolically rather than through the corresponding decimal or hexadecimal value. We could give a set of symbolic constants at the beginning of each program, but it will be more convenient to place the definitions in the macro library. If you are using the Digital Research assembler, use your system editor to create a disk file with the name

```
CPMMAC.LIB
```

If you are using the Microsoft assembler, name the file

```
CPMMAC.MAC
```

We will add each new macro to this disk file. Each assembly language program that references this file will contain the following statement near the beginning:

```
MACLIB    CPMMAC
```

The MACLIB statement instructs the macro assembler to search the disk file named CPMMAC for the required macro definitions.

Notice that the Digital Research assembler requires an extension of ASM for the assembly language program and an extension of LIB for the macro library. On the other hand, the Microsoft assembler expects an extension of MAC for both.

Enter the information given in Figure 4.1 into the disk file CPMMAC.LIB (or CPMMAC.MAC if you use the Microsoft assembler). Notice that the library begins with a brief description on the first line, and the current date is placed on the second line. *Change this date whenever alterations are made to the file.* The third item in the library is a directory listing of the macros defined in the file. Of course, there are no macros at this time. However, this library will contain about 40 macros by the time we have completed this book, so we should document the contents carefully. The symbolic constants are added next.

We will now place our first macro in the library. This macro will code the version number into each program we write.

A Macro to Code the Version Number

In Figure 3.11 we placed a creation date near the beginning of the source program so we could distinguish the new version from previous versions. However, this date is not actually coded into the binary form of the program. After a program is assembled into binary code, it is difficult to determine exactly when it was created. If there are two programs with similar names, it may not be possible to choose the more recent version. For this reason we will code a version number into each program we write from now on. We will write the information in ASCII so that it will be easy to decipher. To make matters simple, we will code the date and the program name. Then we can easily identify the name of the program and the most recent date. To accomplish this we will use an inline macro called VERSN.

The lines of a computer program are normally executed in sequence, one after the other. Therefore, one programming technique is to place the

main part of the program at the beginning and the subroutines at the end. For example:

```
MAIN:
        . . .
        CALL    SUB1
        CALL    SUB2
        . . .
SUB1:
        . . .
        RET
SUB2:
        . . .
        RET
```

With this method, the main program with its subroutine calls can be written first. The subroutines then follow the main program.

An alternate technique is to place the subroutines directly in the path of the main program. In this case we must use a branch to get around the obstruction. For example:

```
MAIN:
        . . .
        CALL    SUB1
        JMP     AROUND
SUB1:
        . . .
        RET
AROUND:
        . . .
        CALL    SUB2
        JMP     OVER
SUB2:
        . . .
        RET
OVER:
```

While this approach appears to be less organized than the previous method, it has an important advantage—the subroutine is written into the main program (inline) where it is needed. Furthermore, this method can be implemented easily with macros. Within this book we shall refer to a macro of this type as an *inline* macro.

Our first macro, shown in Figure 4.2, is called VERSN (for version number). This inline macro is placed near the beginning of the program.

```
;;Macro library for CP/M system routines
;;(Put current date here)
;;
;;Macros in this library:
;;(List each macro name at this point)
;
;
EOF     EQU   1AH     ;end of file
ESC     EQU   1BH     ;escape
CR      EQU   13      ;carriage return
LF      EQU   10      ;line feed
TAB     EQU   9       ;control-I
BLANK   EQU   32      ;space
PERIOD  EQU   46      ;decimal point
COMMA   EQU   44
;;(place macros here)
```

Figure 4.1: The Beginning of a Macro Library: Frequently Used Symbols

```
VERSN       MACRO   NUM
;;(Put current date here)
;;Inline macro to embed version number.
;;NUM is enclosed in quotes.
;;
;;Usage:     VERSN   'XX.XX.XX.NAME'
;;
            LOCAL   AROUND
            JMP     AROUND
            DB      'Ver ',NUM
AROUND:                             ;;VERSN
            ENDM
```

Figure 4.2: Macro VERSN to Code the Version Number

The macro reference

```
VERSN   '9.23.82.FIRST'
```

will generate three statements:

```
        JMP     ??0001
        DB      'Ver ','9.23.82.FIRST'
??0001:
```

This macro can be used to embed information, such as the date and program name, directly into the binary code. The data statement "Ver 9.23.82.FIRST" is embedded in the program and a jump instruction is used to get around the expression. The label AROUND is declared to be a local variable in the macro definition. This means that it has meaning only within the macro definition. The Digital Research assembler assigns the symbol ??0001 to the first use of a local variable (AROUND in this case). Other macro assemblers may use a different symbol.

If this macro is used more than once in the same program, a different label will be generated each time. Thus the word AROUND does not actually appear in the assembly listing. The label AROUND can be used elsewhere in the program, or as a variable in another macro, without producing a duplicate-name error. Notice that the symbol NUM is a dummy parameter. It too can be used outside the macro without producing a conflict.

The second statement generated by macro VERSN defines the data to be embedded in the program. (The assembler directive DB stands for "define byte.") The operand in this example consists of a string of alphanumeric characters enclosed in apostrophes. However, byte-sized symbols can also be used. The third statement generated by macro VERSN is the label ??0001, which is the target of the jump instruction.

Now create another file called TESTVER.ASM. We will use this program to test our first macro. Type in the information shown in Figure 4.3. Notice that this program references our macro library. Put today's date at the beginning of the program and also in the parameter to macro VERSN.

If you are using the Microsoft assembler, you have to make a few changes. First, remove the apostrophes enclosing the title on the first line. Second, be sure that the MACLIB statement is written in uppercase letters.* Third, remove the ORG statement. Fourth, place a .XLIST statement just before the MACLIB statement and a .LIST afterward. This

*Uppercase letters are not necessary in the Digital Research version, but they are shown here to clearly differentiate program lines from comment lines. To highlight the macro references, we set them in boldface type in this book.

```
TITLE    'TESTVER to test macro VERSN'
;
;Mar. 3, 82
;
BOOT     EQU      0              ;warm boot
TPA      EQU      100H           ;where programs go
;
         MACLIB   CPMMAC
ORG      TPA                     ;omit for Microsoft version
;
START:
         VERSN    '3.3.82.FIRST'
         JMP      BOOT
;
         END      START
```

Figure 4.3: Program to Test Macro VERSN

tells the Microsoft assembler not to print out the macro library.

Assemble the program and compare your assembly listing to the one given in Figure 4.4. (Assemblers consider lowercase and uppercase letters to be equivalent. However, if you use lowercase letters, the Digital Research assembler converts them to uppercase.) The first instruction shown in the assembly listing follows the label START. It is a jump around the coding of the program name and date. Notice that the first two lines of code contain plus symbols between the address and the corresponding code. This is the method Digital Research uses to indicate lines that are generated by macros.

Load the assembled file into memory and examine it with the debugger. For the Digital Research version, this is done with the command

```
SID    TESTVER.HEX
```

Display the first part of memory with the D command:

```
D100,11F
```

The result will be as follows:

```
0100: C3 13 01 56 65 72 20 33 2E 33 2E 38 32 2E 46 49 ...Ver 3.3.82.FI
0110: 52 53 54 C3 00 00 00 00 00 00 00 00 00 00 00 00 RST.............
```

```
                TITLE   'TESTVER to test macro VERSN'
                ;
                ;Mar. 3, 82
                ;
0000 =          BOOT    EQU     0           ;warm boot
0100 =          TPA     EQU     100H        ;where programs go
                ;
                        MACLIB  CPMMAC
0100            ORG     TPA                 ;omit for  Microsoft version
                ;
                START:
                        VERSN   '3.3.82.FIRST'
0100+C31301             JMP     ??0001
0103+566572             DB      'Ver ','3.3.82.FIRST'
0113 C30000             JMP     BOOT
                ;
0116                    END     START
```

Figure 4.4: Assembly Listing for Figure 4.3

There are three parts to this display. The first number on each line is the address (100 hex for the first line in this example.) The second part of the line gives the contents of 16 bytes of memory expressed in hexadecimal. The third part shows the ASCII representation of the same 16 bytes. If the bytes are not printable ASCII characters, they are shown as decimal points. You can branch to this program with the command G100. However, this simple program does not actually do anything; we only wrote it test the assembler operation.

Macros to Save and Restore the Incoming Stack

When a program is executed from the CP/M operating system, it is loaded from disk into the transient program area (TPA) starting at address 100 hex. CP/M then branches to 100 hex. At the conclusion of the program, it is possible to return to CP/M by one of two different methods. The simplest approach is to perform a warm start with a jump to address 0, as we did in Figure 4.3.

Another method of returning to CP/M is to save the incoming stack pointer and set up a new stack for the program to use. At the conclusion of the program, the original stack pointer is restored and a return instruction

is executed. This method of termination is faster and therefore preferable to the previous method, because it does not reload the CCP and BDOS from disk. We will use this approach for most of the programs in this book.

Sometimes, however, a program is so large that it destroys the CCP. In this case the program *must* terminate with a warm start. A new copy of the CCP and BDOS is then loaded from the system disk.

Saving and restoring the stack pointer is easily accomplished with a Z80 CPU. The Z80 mnemonics are as follows:

```
START:
        LD      (OLDSTK),SP
        LD      SP,STACK
        . . .
DONE:
        LD      SP,(OLDSTK)
        RET
```

The first two instructions are placed at the beginning of the program. The stack pointer is saved in a memory location called OLDSTK. The new stack is placed at the location defined by STACK. Two other instructions are placed at the end of the program. The first instruction restores the original stack pointer from the memory location OLDSTK. The final instruction returns to CP/M.

The 8080 CPU does not have instructions for directly saving and restoring the stack pointer. Consequently, the 8080 version is more complicated. The usual method is to copy the incoming stack pointer into the HL register pair and save this directly in memory. The instructions are as follows:

```
START:
        LXI     H,0             ;clear
        DAD     SP              ;add pointer
        SHLD    OLDSTK          ;save
        LXI     SP,STACK        ;new one
```

At the conclusion of the program, the original stack pointer is loaded from memory into the HL register and then transferred into the stack pointer register. A return is then executed. The instructions look like this:

```
DONE:
        LHLD    OLDSTK      ;orig stack
        SPHL
        RET
```

With either the Z80 or 8080 version, we also have to allocate the storage place for the original stack pointer and the new stack area. Thus we include the following lines:

```
OLDSTK:   DS    2     ;incoming stack
          DS    34
STACK:
```

For most of the programs in this book we will want to save the incoming stack pointer and restore it at the end. Consequently, it will be convenient to perform these operations with two macros. The macro at the beginning of the program will be called ENTER and the one at the end will be called EXIT. (We must be careful that the symbols we choose are not reserved by the assembler. For example, we cannot select the symbol END.)

Add the macros shown in Figure 4.5 to the macro library (CPMMAC). If you place them before macro VERSN, the three macros will be in alphabetical order. Be sure to place the names ENTER and EXIT in the directory near the beginning of the macro library.

The ENTER macro generally will be placed immediately after the label START; the EXIT macro is placed at the end of the program. Notice that macro ENTER has no parameters, but macro EXIT has two dummy parameters. There are also two conditional assembly blocks within macro EXIT. With this arrangement, it is possible to generate many different sets of instructions from the same macro definition.

If no parameters are included in the reference to macro EXIT, the two dummy parameters WHERE? and SPACE? will be defined as NUL. The conditional expressions

```
IF    NUL WHERE?
```

and

```
IF    NUL SPACE?
```

will be true and the first part of the conditional block, down to the ELSE statement, will be assembled. The instruction between the ELSE and the ENDIF statements will not be assembled. The resulting code will include a return instruction after the incoming stack pointer is restored, and 34 bytes of stack space will be provided.

Notice that the stack is placed at the end of the program. It might seem more logical to place the stack at the beginning. However, the resulting program will then be much larger, because the stack space must be included with the program. The stack need not be saved when it is placed at the end.

Make a copy of the test program given in Figure 4.3 and alter it to look

```
ENTER     MACRO
;;(Put current date here)
;;inline macro to save the incoming stack
;;
          LXI       H,0             ;clear
          DAD       SP              ;add pointer
          SHLD      OLDSTK          ;save
          LXI       SP,STACK
                                    ;;ENTER
          ENDM
;
EXIT      MACRO     WHERE?,SPACE?
;;
;;Inline macro to restore the incoming stack
;;and branch to location WHERE?
;;If WHERE? is omitted, execute a return instruction.
;;SPACE? sets stack space; default is 34.
;;
          LHLD      OLDSTK
          SPHL
          IF        NUL WHERE?
          RET
          ELSE
          JMP       WHERE?
          ENDIF
;
OLDSTK:   DS        2               ;incoming stack
          IF        NUL SPACE?
          DS        34
          ELSE
          DS        SPACE?
          ENDIF
STACK:    EQU       $               ;omit EQU $ for Microsoft
                                    ;;EXIT
          ENDM
```

Figure 4.5: Macros ENTER and EXIT to Save and Restore the Incoming Stack Pointer

```
TITLE   'TESENT to test macros ENTER and EXIT'
;
;Mar. 3, 82
;
BOOT    EQU     0       ;warm boot
TPA     EQU     100H    ;where programs go
;
;Digital Research version
        MACLIB  CPMMAC
;
ORG     TPA
;
START:
        ENTER
        VERSN   '3.3.82.SECOND'
        EXIT
;
        END     START
```

Figure 4.6: Program to Test Macros ENTER and EXIT

like the program shown in Figure 4.6. If you are using the Microsoft assembler, make the same changes you made in Figure 4.3. In addition, you must remove the expression EQU $ following the label STACK in macro EXIT. For the Microsoft version, the end of macro EXIT looks like this:

```
STACK:
                    ;;EXIT
        ENDM
```

Notice that the reference to macro EXIT has no parameters. Assemble the program and compare the assembly listing to Figure 4.7. This program can be executed, but it will not do anything.

Using Parameters in Macro EXIT

In the previous program, the reference to macro EXIT did not contain parameters. But consider the program fragment shown in Figure 4.8. In

```
                TITLE   'TESENT to test macros ENTER and EXIT'
                ;
                ;Mar. 3, 82
                ;
0000 =          BOOT    EQU     0           ;warm boot
0100 =          TPA     EQU     100H        ;where programs go
                ;
                ;Digital Research version
                        MACLIB  CPMMAC
                ;
0100            ORG     TPA
                ;
                START:
                        ENTER
0100+210000             LXI     H,0         ;clear
0103+39                 DAD     SP          ;add pointer
0104+222301             SHLD    OLDSTK      ;save
0107+314701             LXI     SP,STACK
                        VERSN   '3.3.82.SECOND'
010A+C31E01             JMP     ??0001
010D+566572             DB      'Ver ','3.3.82.SECOND'
                        EXIT
011E+2A2301             LHLD    OLDSTK
0121+F9                 SPHL
0122+C9                 RET
0123+           OLDSTK: DS      2           ;incoming stack
0125+                   DS      34
0147+ =         STACK:  EQU     $           ;omit EQU $ for Microsoft
                ;
0147                    END     START
```

Figure 4.7: Assembly Listing for Figure 4.6

this example, the reference to macro EXIT contains two parameters:

```
EXIT    BOOT,20
```

During assembly, the dummy parameter WHERE? is defined as the label BOOT and the dummy parameter SPACE? takes on the value of 20.

```
                        EXIT    BOOT, 20
011E+2A2501             LHLD    OLDSTK
0121+F9                 SPHL
0122+C30000             JMP     BOOT
0125+           OLDSTK: DS      2           ;incoming stack
0127+                   DS      20
013B+=          STACK:  EQU     $
```

Figure 4.8: Using Parameters in Macro EXIT

Thus the expressions

```
IF    NUL WHERE?
```

and

```
IF    NUL SPACE?
```

are false. The assembled code includes a jump to BOOT and provides 20 bytes of stack space.

Of course, other combinations of parameters are possible. For example, the statement

```
EXIT    ,20
```

contains only the second parameter. The comma in front of the 20 indicates that the first parameter is omitted and is therefore defined as NUL. This statement will generate a return instruction and provide 20 bytes of stack space.

A MACRO TO MOVE INFORMATION

From time to time we will find it necessary to move information from one part of the computer's memory to another. This is called a block move. We will now write a macro to perform this task. Both the 8080 and the Z80 CPUs incorporate 16-bit registers that can be used as pointers during the move. The Z80 also contains instructions for directly performing block moves. The block move can be greatly simplified, therefore, if a program is designed to run on a Z80 CPU. However, we will only consider the 8080 version at this time.

Add the MOVE macro given in Figure 4.9 to your macro library. Place

```
MOVE        MACRO   FROM, TO, BYTES
;;(Put current date here)
;;inline macro to move text
;;
            LOCAL   AROUND
            PUSH    H
            PUSH    D
            PUSH    B
            LXI     H,FROM
            LXI     D,TO
            LXI     B,BYTES
            CALL    MOVE2?
            POP     B
            POP     D
            POP     H
            JMP     AROUND
;
MOVE2?:
            MOV     A,M         ;get it
            STAX    D           ;put it
            INX     H           ;from
            INX     D           ;to
            DCX     B           ;byte count
            MOV     A,C
            ORA     B
            JNZ     MOVE2?      ;not done
            RET
AROUND:                         ;;MOVE
            ENDM
```

Figure 4.9: Macro MOVE, Version 1

it between macros EXIT and VERSN to maintain alphabetic order. Be sure to add the name MOVE to the directory at the beginning of the macro library. The MACLIB directory should now list the following macros:

```
ENTER
EXIT
MOVE
VERSN
```

The organization of macro MOVE is typical of many of the macros we will write in this book. There will be an initialization section, a subroutine call, a jump around the subroutine, and the subroutine itself.

Let us examine the details of macro MOVE. There are three dummy parameters: FROM, TO, and BYTES. As the names imply, FROM refers to the address of the source block, TO refers to the destination block, and BYTES gives the number of bytes to be moved. The macro begins by saving the CPU registers with PUSH instructions. Then the HL register is loaded with the source address, the DE register is loaded with the destination address, and the BC register is loaded with the number of bytes to be moved. (Remember that the X in the mnemonic refers to the extended or double register. Thus, the operand H means HL, and so forth.)

The main part of the macro calls subroutine MOVE2? to perform the actual move. A byte is moved from the original memory location to the accumulator with a MOV A,M instruction. The byte is then moved to the destination with a STAX D instruction. The HL and DE pointers are incremented and the byte count in register BC is decremented. The subroutine continues in this way until the byte count in register BC reaches zero.

Testing a double register for zero is more complicated than testing a single register, because the CPU flags are not affected by double-register increment or decrement instructions. Thus, the instructions

```
DCX   B
JZ    MOVE2?
```

will not work. The macro performs the test for zero by moving one half of the register to the accumulator and executing a logical OR with the other half. At the conclusion of the block move, control returns to the main part of the macro.

For the first expansion of macro MOVE, subroutine MOVE2? is coded inline, immediately after the main part of the macro. Consequently, there is a jump instruction to skip over this subroutine. The local label AROUND is used for this purpose. Notice that the name of subroutine MOVE2? has not been declared as a local variable; rather, it is a global variable. It can therefore be called from other parts of the main program.

Create a disk file named MOVE1.ASM and enter the program shown in Figure 4.10. We will use this program to test the operation of macro MOVE.

Our test program begins with macro VERSN and continues with macro MOVE. The instructions terminate with a jump to BOOT followed by an arrow that points to this jump. The source string begins at the label TEXT

```
TITLE   'TESTMOVE to test macro MOVE'
;
;Dec. 16, 81
;
FALSE   EQU     0
TRUE    EQU     NOT FALSE
;
BOOT    EQU     0           ;system reboot
BDOS    EQU     5           ;BDOS entry point
TPA     EQU     100H        ;transient program area
;
        MACLIB  CPMMAC
;
ORG     TPA
;
START:
        VERSN   '12.16.81.TESTMOVE.1'
        MOVE    TEXT, NEWTEX, TEXEND-TEXT
        JMP     BOOT
;
        DB      '<====='
TEXT:
        DB      'A test of macro MOVE'
TEXEND:
;
ORG     400H
;
NEWTEX: DS      1
;
        END     START
```

Figure 4.10: Program to Test Version 1 of Macro MOVE

and continues to the label TEXEND. The destination address is NEWTEX.

Assemble the program and compare the assembly listing to Figure 4.11. Take note of the final jump instruction at address 13A hex. (The address in your program may be different, depending on how you coded the date.)

```
                TITLE   'TESTMOVE to test macro MOVE'
                ;
                ;Dec. 16, 81
                ;
0000 =          FALSE   EQU     0
FFFF =          TRUE    EQU     NOT FALSE
                ;
0000 =          BOOT    EQU     0           ;system reboot
0005 =          BDOS    EQU     5           ;BDOS entry point
0100 =          TPA     EQU     100H        ;transient program area
                ;
                        MACLIB  CPMMAC
                ;
0100            ORG     TPA
                ;
                START:
                        VERSN   '12.16.81.TESTMOVE.1'
0100+C31A01             JMP     ??0001
0103+566572             DB      'Ver ','12.16.81.TESTMOVE.1'
                        MOVE    TEXT, NEXTEX, TEXEND-TEXT
011A+E5                 PUSH    H
011B+D5                 PUSH    D
011C+C5                 PUSH    B
011D+214201             LXI     H,TEXT
0120+110004             LXI     D,NEWTEX
0123+011400             LXI     B,TEXEND-TEXT
0126+CD2F01             CALL    MOVE2?
0129+C1                 POP     B
012A+D1                 POP     D
012B+E1                 POP     H
012C+C33A01             JMP     ??0002
012F+7E                 MOV     A,M         ;get it
0130+12                 STAX    D           ;put it
0131+23                 INX     H           ;from
0132+13                 INX     D           ;to
0133+0B                 DCX     B           ;byte count
0134+79                 MOV     A,C
0135+B0                 ORA     B
```

Figure 4.11: Assembly Listing for Figure 4.10

```
0136+C22F01             JNZ     MOVE2?    ;not done
0139+C9                 RET
013A C30000             JMP     BOOT
                ;
013D 3C3D3D             DB      '<===='
                TEXT:
0142 412074             DB      'A test of macro MOVE'
                TEXEND:
                ;
0400            ORG     400H
                ;
0400            NEWTEX: DS      1
                ;
0401                    END     START
```

Figure 4.11 (continued)

We will need this location in our next step. Load the hex file into memory with the debugger command

```
SID    MOVE1.HEX
```

Display the first part of memory with the command D100,15F. The result will be as follows:

```
0100: C3 1A 01 56 65 72 20 31 32 2E 31 36 2E 38 31 2E ...Ver 12.16.81.
0110: 54 45 53 54 4D 4F 56 45 2E 31 E5 D5 C5 21 42 01 TESTMOVE.1...!B.
0120: 11 00 04 01 14 00 CD 2F 01 C1 D1 E1 C3 3A 01 7E ......./.....:.~
0130: 12 23 13 0B 79 B0 C2 2F 01 C9 C3 00 00 3C 3D 3D .#..y../.....<==
0140: 3D 3D 41 20 74 65 73 74 20 6F 66 20 6D 61 63 72 ==A test of macr
0150: 6F 20 4D 4F 56 45 00 00 00 00 00 00 00 00 00 00 o MOVE..........
```

The text that was coded with macro VERSN (near the beginning of the program) is plainly visible in the ASCII representation. On the fourth line, the left-pointing arrow indicates the location of the final jump instruction at 13A hex. Run the program by giving the command

```
G100,13A
```

This command begins execution of the program at address 100 and terminates it with a return to the debugger at address 13A hex. The debugger sets a breakpoint (an automatic return to itself) at address 13A. It does this

by changing the jump instruction at 13A hex to restart 7. The debugger will respond with the statement

```
*013A
```

indicating that it stopped execution at address 13A.

Give the debugger command D400,41F to display the destination block. The result will be as follows:

```
0400: 41 20 74 65 73 74 20 6F 66 20 6D 61 63 72 6F 20 A test of macro
0410: 4D 4F 56 45 00 00 00 00 00 00 00 00 00 00 00 00 MOVE............
```

The program has moved the text "A test of macro MOVE" from the source block to the destination block.

If you want to repeat this test, zero the destination memory with the debugger fill command:

```
F400,41F,0
```

Then repeat the original command G100,13A.

Macro MOVE, Version 2

For our second version of macro MOVE, we will introduce a technique that is applicable to many of the macros we will be writing in this book. We saw previously that our inline macros contain four parts—an initialization section, a subroutine call, a jump around the subroutine, and the subroutine itself. This arrangement is used for the first expansion of the macro. However, on subsequent macro expansions, only the first two parts of the macro are needed. The subroutine generated during the first expansion of the macro is referenced by the other expansions.

We will use a special symbol to indicate whether the macro has been referenced more than once in a program. There are some important reasons for this feature. On the first reference to macro MOVE, a copy of subroutine MOVE2? will be generated. The second reference to macro MOVE will generate another copy of the MOVE2? subroutine. That is, a separate copy of subroutine MOVE2? will be generated for each call to macro MOVE. This is an unnecessary duplication of code. Furthermore, the label MOVE2? is a global variable. When it appears more than once, your assembler will report a phase error, meaning that a symbol has been assigned two different values.

We need a method for generating a copy of the MOVE2? subroutine the first time macro MOVE is referenced in a program, but not on

subsequent references. There are several ways to do this, but we will choose the one that can be used by all assemblers.

We will define the symbol MVFLAG to indicate whether a copy of subroutine MOVE2? has been generated. The symbol will have one of two values: true or false. This kind of symbol is called a *flag*. This flag is initially defined as FALSE by the statement

```
MVFLAG    SET    FALSE
```

The flag must be defined with a SET statement rather than the usual EQU statement so that it can be changed during assembly. (EQU expressions cannot be changed.) The ideal location for this flag is at the beginning of the macro library. However, the Digital Research assembler does not allow this construction. Consequently, we will place the flag at the beginning of each program that references the macro.

Alter macro MOVE to look like the version shown in Figure 4.12. Notice that just before the JMP AROUND statement there is a conditional expression for testing the state of MVFLAG. On the first reference to macro MOVE, the flag will be false and the expression NOT MVFLAG will be true. Consequently, the next instructions down to the ENDIF statement will be assembled. These instructions generate a copy of subroutine MOVE2?. There is also a very important statement just prior to the ENDIF statement. This is the expression that changes the state of the flag:

```
MVFLAG    SET    TRUE
```

The next time macro MOVE is referenced within the same program, the flag will be true and the expression NOT MVFLAG will be false. Therefore, the assembler will not create another copy of the MOVE2? subroutine. The jump around the subroutine will not be necessary, either.

Make a copy of the source program given in Figure 4.10 and alter it to look like Figure 4.13. Give the new version the name MOVE2.ASM.

Assemble the new test program and compare the last portion of the listing to the one shown in Figure 4.14. Notice that the first call to macro MOVE, at address 11A hex, generates a copy of subroutine MOVE2? at address 12F hex. The JMP AROUND becomes JMP ??0002 (when the Digital Research assembler is used) because it is a local variable. The second reference to macro MOVE, at address 13A hex, does not generate another copy of subroutine MOVE2?, but calls the copy generated by the first reference.

Load the program into memory with the debugger command

```
SID    MOVE2.HEX
```

Display the program with the command D100,17F to give the following:

```
0100: C3 1A 01 56 65 72 20 31 32 2E 31 36 2E 38 31 2E ...Ver 12.16.81.
0110: 54 45 53 54 4D 4F 56 45 2E 32 E5 D5 C5 21 54 01 TESTMOVE.2...!T.
0120: 11 00 04 01 14 00 CD 2F 01 C1 D1 E1 C3 3A 01 7E ......./.....:.~
0130: 12 23 13 0B 79 B0 C2 2F 01 C9 E5 D5 C5 21 68 01 .#..y../.....!h.
0140: 11 14 04 01 10 00 CD 2F 01 C1 D1 E1 C3 00 00 3C ......./.......<
0150: 3D 3D 3D 3D 41 20 74 65 73 74 20 6F 66 20 6D 61 ====A test of ma
0160: 63 72 6F 20 4D 4F 56 45 2E 20 41 20 73 65 63 6F cro MOVE. A seco
0170: 6E 64 20 4D 4F 56 45 2E 00 00 00 00 00 00 00 00 nd MOVE.........
```

As with the previous version, we can see the ASCII characters at the beginning of the program. The left-pointing arrow is also visible, although now it is pointing to the jump instruction at address 14C hex. Zero the destination block with the command

```
F400,42F,0
```

and execute the program with the statement

```
G100,14C
```

This sets a breakpoint at location 14C hex, the new location of the final instruction. The debugger responds with

```
*014C
```

Display the destination area with the debugger command D400,42F. Verify that the two separate calls to macro MOVE generated the following composite string:

```
0400: 41 20 74 65 73 74 20 6F 66 20 6D 61 63 72 6F 20 A test of macro
0410: 4D 4F 56 45 2E 20 41 20 73 65 63 6F 6E 64 20 4D MOVE. A second M
0420: 4F 56 45 2E 00 00 00 00 00 00 00 00 00 00 00 00 OVE.............
```

```
MOVE        MACRO   FROM, TO, BYTES
;;(Put current date here)
;;inline macro to move text
;;
            LOCAL   AROUND
            PUSH    H
            PUSH    D
            PUSH    B
            LXI     H,FROM
```

Figure 4.12: Macro MOVE, Version 2

```
                LXI     D,TO
                LXI     B,BYTES
                CALL    MOVE2?
                POP     B
                POP     D
                POP     H
                IF      NOT MVFLAG
                JMP     AROUND
;
MOVE2?:
                MOV     A,M             ;get byte
                STAX    D               ;new place
                INX     H               ;from
                INX     D               ;to
                DCX     B               ;byte count
                MOV     A,C
                ORA     B
                JNZ     MOVE2?          ;not done
                RET
MVFLAG          SET     TRUE            ;;one copy
                ENDIF                   ;;not MVFLAG
AROUND:                                 ;;MOVE
                ENDM
```

Figure 4.12 (continued)

```
TITLE   'TESTMOVE to test macro MOVE'
;
;Dec.16, 81
;
FALSE   EQU     0
TRUE    EQU     NOT FALSE
;
BOOT    EQU     0           ;system reboot
BDOS    EQU     5           ;BDOS entry point
TPA     EQU     100H        ;transient program area
;
```

Figure 4.13: Program to Test Version 2 of Macro MOVE

```
MVFLAG  SET     FALSE       ;block move
;
        MACLIB  CPMMAC
;
ORG     TPA
;
START:
        VERSN   '12.16.81.TESTMOVE.2'
        MOVE    TEXT, NEWTEX, TEXT2-TEXT
        MOVE    TEXT2, NEWTEX+TEXT2-TEXT, TEXEND-TEXT2
        JMP     BOOT
;
        DB      '<====='
TEXT:
        DB      'A test of macro MOVE'
TEXT2:
        DB      '. A second MOVE.'
TEXEND:
;
ORG     400H
;
NEWTEX: DS      1
;
        END     START
```

Figure 4.13 (continued)

```
                MOVE    TEXT, NEWTEX, TEXT2-TEXT
011A+E5         PUSH    H
011B+D5         PUSH    D
011C+C5         PUSH    B
011D+215401     LXI     H,TEXT
0120+110004     LXI     D,NEWTEX
0123+011400     LXI     B,TEXT2-TEXT
0126+CD2F01     CALL    MOVE2?
```

Figure 4.14: Partial Assembly Listing of Figure 4.13

```
0129+C1          POP     B
012A+D1          POP     D
012B+E1          POP     H
012C+C33A01      JMP     ??0002
012F+7E          MOV     A,M         ;get byte
0130+12          STAX    D           ;new place
0131+23          INX     H           ;from
0132+13          INX     D           ;to
0133+0B          DCX     B           ;byte count
0134+79          MOV     A,C
0135+B0          ORA     B
0136+C22F01      JNZ     MOVE2?      ;not done
0139+C9          RET
                 MOVE    TEXT2, NEWTEX+TEXT2-TEXT, TEXEND-TEXT2
013A+E5          PUSH    H
013B+D5          PUSH    D
013C+C5          PUSH    B
013D+216801      LXI     H,TEXT2
0140+111404      LXI     D,NEWTEX+TEXT2-TEXT
0143+011000      LXI     B,TEXEND-TEXT2
0146+CD2F01      CALL    MOVE2?
0149+C1          POP     B
014A+D1          POP     D
014B+E1          POP     H
014C C30000      JMP     BOOT
                 ;
014F 3C3D3D      DB      '<===='
                 TEXT:
0154 412074      DB      'A test of macro MOVE'
                 TEXT2:
0168 2E2041      DB      '. A second MOVE.'
                 TEXEND:
                 ;
0400             ORG     400H
                 ;
0400             NEWTEX: DS      1
                 ;
0401                     END     START
```

Figure 4.14 (continued)

Macro MOVE, Version 3

Sometimes we will find it necessary to move a particular string of characters into a memory location. Because the string will not exist prior to the move, the two previous versions of the MOVE macro will not be suitable. Therefore, for our third version we will add a new feature. This version will accept a string, rather than the usual memory pointer, as the first parameter to the macro reference. Thus we can write

```
MOVE    "THIRD", FCB2+1
```

if we want to write the string "THIRD" into the memory location that is one byte beyond the beginning of FCB2. Notice that the third parameter (the number of bytes) and the second comma are omitted in this example. The assembler will automatically calculate the length we need. We will use this method to signal to the macro that the first parameter is a literal variable rather than an address pointer.

The literal parameter is not limited to a quoted string of characters. Variables and constants can also be included if the entire parameter is enclosed in angle brackets. For example, the expression

```
MOVE    <2,"FIFTH">, FCB1
```

will place six bytes in memory starting at the location FCB1 (5C hex). The first byte is the binary number 2; the ASCII string "FIFTH" is placed immediately following it. Of course, symbols such as EOF (end of file), CR (carriage return), and LF (line feed) can be included as well. Notice that there is a comma separating the constant 2 from the string "FIFTH".

Alter macro MOVE so that it looks like Figure 4.15. This third version of macro MOVE begins as before by saving the registers. We then encounter a new feature. When the assembler finds the expression

```
IF      NOT NUL TO
LXI     D,TO
ENDIF
```

it checks to see if the second parameter, the destination address, is actually supplied in the macro reference. If this parameter is omitted, it is assumed that the program has loaded the DE register with the destination address prior to the macro reference. The expression IF NOT NUL TO is false. On the other hand, if the second parameter is provided, the expression IF NOT NUL TO is true. The instruction LXI D,TO is then included.

With the previous versions, the destination address always had to be included in the macro reference as a parameter. But sometimes the destination address is not known at assembly time. This new version of macro

MOVE allows us to obtain the destination address from a memory location or from the result of a calculation performed during execution of the program. Suppose, for example, that the destination address is stored at location DEST. The following instructions will move 20 bytes starting at address FROM into the memory area whose address is stored at location DEST:

```
PUSH    H           ;save
LHLD    DEST        ;get it
XCHG                ;into DE
POP     H           ;restore
MOVE    FROM,,20
```

The next portion of macro MOVE checks to see whether the third parameter, the number of bytes to move, is present. If this parameter is omitted, a literal move is indicated. The instructions between IF NUL BYTES and the ELSE statement are then included. With this version, the assembler generates code to copy the literal first parameter into memory at the location referenced by the symbol MESG. This label is located near the end of the macro. Note that MESG is defined as a local variable. Thus there can be one copy in each expansion of the macro.

The alternate passage between ELSE and ENDIF is assembled when the third parameter is supplied in the macro reference. The first parameter can be omitted in this case as well. Thus the command

```
MOVE    , ,20
```

will move 20 bytes from the address referenced by HL to the address referenced by DE.

The macro continues with the usual call to subroutine MOVE2? and then restores the registers. The JMP AROUND instruction is embedded in a conditional block that checks for two things: the state of MVFLAG and whether the third parameter, BYTES, is present.

```
IF      NOT MVFLAG OR NUL BYTES
JMP     AROUND
ENDIF
```

If NOT MVFLAG is true, subroutine MOVE2? will be needed and so will the jump instruction. Also, whenever a string move is indicated by a missing third parameter, we need a jump around the string. Otherwise, subroutine MOVE2? and the jump instruction are omitted.

It is important to notice that the two expressions on either side of the logical OR operation, NOT MVFLAG and NUL BYTES, must appear in the

```
MOVE        MACRO   FROM, TO, BYTES
;;(Put current date here)
;;inline macro to move text
;;
            LOCAL   AROUND, MESG
            PUSH    H
            PUSH    D
            PUSH    B
            IF      NOT NUL TO
            LXI     D,TO
            ENDIF
            IF      NUL BYTES           ;;string move
            LXI     H,MESG              ;;test
            LXI     B,AROUND-MESG
            ELSE                        ;;not string move
            IF      NOT NUL FROM
            LXI     H, FROM
            ENDIF
            LXI     B,BYTES
            ENDIF                       ;;string/not string
            CALL    MOVE2?
            POP     B
            POP     D
            POP     H
            IF      NOT MVFLAG OR NUL BYTES
            JMP     AROUND
            ENDIF
;
            IF      NOT MVFLAG
MOVE2?:
            MOV     A,M                 ;get byte
            STAX    D                   ;new place
            INX     H                   ;from
            INX     D                   ;to
            DCX     B                   ;byte count
            MOV     A,C
            ORA     B
            JNZ     MOVE2?              ;not done
            RET
```

Figure 4.15: Macro MOVE, Version 3

```
;
MVFLAG      SET       TRUE              ;;one copy
            ENDIF                       ;;not MVFLAG
            IF        NUL BYTES
MESG:
            DB        FROM              ;;text
            ENDIF
AROUND:                                 ;;MOVE
            ENDM
```

Figure 4.15 (continued)

order shown. They cannot be interchanged or the assembler will interpret the combination differently. This is due to the order of evaluation of the NUL and OR operators. The expression

NUL BYTES OR NOT MVFLAG

is interpreted as

NUL (BYTES OR NOT MVFLAG)

This is not the same as

(NUL BYTES) OR NOT MVFLAG

which is the desired result.

To test this third version of macro MOVE, create a new file named MOVE3.ASM and copy file MOVE2.ASM into it. Alter MOVE3.ASM to look like Figure 4.16. Assemble the program and load it into memory with the debugger. Display the first part of the program with the command D100,1AF. Notice that the final jump is located at address 17C hex:

```
0100: C3 1A 01 56 65 72 20 31 32 2E 31 36 2E 38 31 2E ...Ver 12.16.82.
0110: 54 45 53 54 4D 4F 56 45 2E 33 E5 D5 C5 11 75 00 TESTMOVE.3....u.
0120: 21 3A 01 01 03 00 CD 2F 01 C1 D1 E1 C3 3D 01 7E !:.....=.~
0130: 12 23 13 0B 79 B0 C2 2F 01 C9 24 24 24 E5 D5 C5 .#..y../..$$$...
0140: 11 5C 00 21 52 01 01 06 00 CD 2F 01 C1 D1 E1 C3 .\.!R...../.....
0150: 58 01 02 46 49 46 54 48 E5 D5 C5 11 00 04 21 84 X..FIFTH......!.
0160: 01 01 14 00 CD 2F 01 C1 D1 E1 E5 D5 C5 11 14 04 ...../..........
0170: 21 98 01 01 10 00 CD 2F 01 C1 D1 E1 C3 00 00 3C !....../.......<
0180: 3D 3D 3D 3D 41 20 74 65 73 74 20 6F 66 20 6D 61 ====A test of ma
0190: 63 72 6F 20 4D 4F 56 45 2E 20 41 20 73 65 63 6F cro MOVE. A seco
01A0: 6E 64 20 4D 4F 56 45 2E 00 00 00 00 00 00 00 00 nd MOVE.........
```

```
TITLE   'TESTMOVE to test macro MOVE'
;
;Dec. 16, 81
;
FALSE   EQU     0
TRUE    EQU     NOT FALSE
;
BOOT    EQU     0               ;system reboot
BDOS    EQU     5               ;BDOS entry point
TPA     EQU     100H            ;transient program area
FCB1    EQU     5CH             ;input FCB
FCB2    EQU     6CH             ;2nd parameter
;
MVFLAG  SET     FALSE           ;block move
;
        MACLIB  CPMMAC
;
ORG     TPA
;
START:
        VERSN   '12.16.81.TESTMOVE.3'
        MOVE    '$$$', FCB2+9
        MOVE    <2,'FIFTH'>,FCB1
        MOVE    TEXT, NEWTEX, TEXT2-TEXT
        MOVE    TEXT2, NEWTEX+TEXT2-TEXT, TEXEND-TEXT2
        JMP     BOOT
;
        DB      '<====='
TEXT:
        DB      'A test of macro MOVE'
TEXT2:
        DB      '. A second MOVE.'
TEXEND:
;
ORG     400H
;
NEWTEX: DS      1
 ;
        END     START
```

Figure 4.16: Program to Test Version 3 of Macro MOVE

Execute the third version with the command G100,17C. Display the region from 50 to 7F hex with the command D50,7F. The result shows that the three dollar signs were moved to address 75 hex, a binary 2 was placed at 5C hex, and the string "FIFTH" was deposited immediately afterward:

```
0050: 00 00 00 00 00 00 00 00 00 00 00 00 02 46 49 46 .............FIF
0060: 54 48 00 00 00 00 00 00 00 00 00 00 00 00 00 00 TH..............
0070: 00 00 00 00 00 24 24 24 00 00 00 00 00 00 00 00 .....$$$........
```

A check can also be made of the region starting at 400 hex to see that the other two parts worked properly. All three versions of the MOVE macro are coded inline; that is, the macro statement is placed wherever it is needed. The macro includes the JMP AROUND statement to skip over subroutine MOVE2? at the first reference.

A MACRO TO FILL MEMORY WITH A CONSTANT

The MOVE macro we just developed can be used to deposit a string of characters in memory. As an example, we placed three dollar signs in the second file control block with the macro statement

```
MOVE    '$$$', FCB2+9
```

However, the MOVE macro is not convenient if we want to fill a large number of locations with a particular value. So we will now develop a companion macro named FILL. With this macro we can fill any portion of memory with a particular constant. This macro is coded directly inline, just as the MOVE macro was.

Incorporate macro FILL, shown in Figure 4.17, into your macro library. Also add the name FILL to the directory at the beginning of the macro library. This is the second macro in the library to use a flag. Many of the macros we will add to the library will use flags, so we will add a new column to the directory listing to identify the associated flag. The directory should now look like this:

```
;;Macros in this library                          Flags
;;ENTER   MACRO                                   (none)
;;EXIT    MACRO   SPACE?                          (none)
;;FILL    MACRO   ADDR, BYTES, CHAR               FLFLAG
;;MOVE    MACRO   FROM, TO, BYTES                 MVFLAG
;;VERSN   MACRO   NUM                             (none)
```

```
FILL        MACRO   ADDR, BYTES, CHAR
;;(Put current date here)
;;Inline macro to fill byte memory
;;locations with CHAR starting at ADDR
;;Usage:    FILL    FCB+1, BLANK, 8
;;          FILL    FCB+9, '?', 3
;;
            LOCAL   AROUND
            PUSH    H
            PUSH    B
            IF      NOT NUL ADDR
            LXI     H,ADDR
            ENDIF
            MVI     C,BYTES
            MVI     A,CHAR
            CALL    FILL2?
            POP     B
            POP     H
            IF      NOT FLFLAG
            JMP     AROUND
FILL2?:
            MOV     M,A             ;put into memory
            INX     H               ;pointer
            DCR     C               ;count
            JNZ     FILL2?          ;keep going
            RET
FLFLAG      SET     TRUE
            ENDIF
AROUND:                             ;;FILL
            ENDM
```

Figure 4.17: Macro FILL to Fill a Block of Memory with a Byte

Notice that the address of the area to be filled is the first parameter to macro FILL. Because of the conditional expression

```
IF      NOT NUL ADDR
LXI     H,ADDR
ENDIF
```

the first parameter in the macro reference may be omitted. The second

parameter, the number of bytes in the block, is loaded into the C register. Because this is an 8-bit register, the block size is limited to 256 bytes. (A value of 0 fills a block of 256 bytes.) If a larger block is needed, the macro can be referenced more than once. Alternatively, the macro could be rewritten to use the BC double register rather than the C register. We will do this in Chapter 8.

Make a copy of the test program in Figure 4.16 and give it the name TESTFILL.ASM. Alter the program so it looks like the version shown in Figure 4.18. Notice that FLFLAG is set to FALSE near the beginning of the source program. This flag serves the same purpose as MVFLAG did in the previous macro. The flag is initially set to FALSE so that a copy of subroutine FILL2? is generated when the macro is first referenced. The flag is then set to TRUE in the macro so that no additional copies of FILL2? are made on subsequent references.

Assemble the program and load it into memory with the debugger. Display the program with the command

```
D100,16F
```

The resulting output contains the familiar arrow pointing to an important jump instruction at 156 hex. Notice that macros ENTER and EXIT are included in this version. The FILL macro is used three times in this program. On the first reference, macro FILL deposits dollar signs in the second file control block. This performs the same task as the first reference to macro MOVE in the previous program. The next reference to macro FILL sets 40 hex bytes to blanks and the final reference sets the next 40 hex bytes to binary zeros.

```
0100: 21 00 00 39 22 63 01 31 87 01 C3 22 01 56 65 72 !..9"c.1...".Ver
0110: 20 31 32 2E 32 34 2E 38 31 2E 54 45 53 54 46 49  12.24.81.TESTFI
0120: 4C 4C E5 C5 21 75 00 0E 03 3E 24 CD 33 01 C1 E1 LL..!u...>$.3...
0130: C3 3A 01 77 23 0D C2 33 01 C9 E5 C5 21 00 08 0E .:.w#..3....!...
0140: 40 3E 20 CD 33 01 C1 E1 E5 C5 21 40 08 0E 40 3E @> .3.....!@..@>
0150: 00 CD 33 01 C1 E1 C3 5E 01 3C 3D 3D 3D 3D 2A 63 ..3....^.<====*c
0160: 01 F9 C9 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
```

Fill the 800 hex block with the constant A5 using the debugger command F800,8FF,A5. Execute the new program with the debugger command G100,156. Then display the file control blocks with the command D50,7F and verify that the three dollar signs are present:

```
0050: 00 00 00 00 00 00 00 00 00 00 00 28 00 20 20 20 ...........(.
0060: 20 20 20 20 20 20 20 20 00 00 00 00 16 00 00 00          ........
0070: 00 00 00 00 00 24 24 24 00 00 00 00 00 00 00 00 .....$$$........
```

```
TITLE    'TESTFILL to test macro FILL'
;
;Dec. 24, 81
;
FALSE    EQU      0
TRUE     EQU      NOT FALSE
;
BOOT     EQU      0              ;system reboot
BDOS     EQU      5              ;BDOS entry point
TPA      EQU      100H           ;transient program area
FCB1     EQU      5CH            ;input FCB
FCB2     EQU      6CH            ;2nd parameter
;
FLFLAG   SET      FALSE          ;FILL flag
;
         MACLIB   CPMMAC
;
ORG      TPA
;
START:
         ENTER
         VERSN    '12.24.81.TESTFILL'
         FILL     FCB2+9, 3, '$'
         FILL     800H, 40H, BLANK
         FILL     800H+40H, 40H, 0
         JMP      DONE
         DB       '<====='
DONE:
         EXIT
;
         END      START
```

Figure 4.18: Program to Test Macro FILL

A final display of the 800 hex block will show the results of the second and third macro references. Give the command D800,88F. The results should look like this:

```
0800: 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
0810: 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
0820: 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
0830: 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
0840: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
```

```
0850: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
0860: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
0870: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
0880: A5 A5 A5 A5 A5 A5 A5 A5 A5 A5 A5 A5 A5 A5 A5 A5 ................
```

Return to CP/M by typing control-C. Now we will develop a pair of macros for comparing one region of memory to another.

A MACRO TO COMPARE TWO BLOCKS OF INFORMATION

We will often need to determine whether a particular memory area matches another memory area or string of characters. For example, in Chapter 6 we will write a program to display an ASCII disk file on the video screen. A binary COM file cannot be displayed in this way, so we will want to compare the file type the user has entered to the string COM. The program can be terminated when a COM file is given.

As a second example, suppose a program needs a file name, but the user enters an ambiguous file name such as

```
SORT.*
```

The CCP converts the asterisk to three question marks. The program is looking for a single file name but the CCP gives

```
SORT.???
```

In this case the program may have to deal with many different files rather than a single file. To be prepared for this possibility, we must compare the input file name to a string of question marks.

The inline macro COMPAR, shown in Figure 4.19, can be used to make general comparisons of blocks up to 256 bytes in length. If you want to compare two memory regions, give the addresses of each block as the first and second parameters. The number of bytes in each block is given as the third parameter. The maximum block size is 256 bytes, because the C register counts the block size. Copy this macro into your macro library, placing it in alphabetic order.

The conditional blocks

```
IF    NOT NUL FIRST
```

and

```
IF    NOT NUL SECOND
```

allow either or both of the first two parameters to be omitted. If the

```
COMPAR  MACRO   FIRST, SECOND, BYTES
;;(Put current date here)
;;Inline macro to compare 2 memory areas.
;;Zero flag is set if both are the same,
;;first and second may be addresses,
;;third parameter is number of bytes.
;;First parameter may be a quoted string,
;;in which case there is no third parameter.
;;Any of the parameters may be omitted.
;;Register A is altered.
;;
;;Usage:  COMPAR  FCB1, FCB2, 12
;;        COMPAR  '???', FCB1+9
;;        COMPAR  , , 5
;;
        LOCAL   MESG, AROUND
        PUSH    H
        PUSH    D
        PUSH    B
        IF      NUL BYTES
        LXI     H,MESG              ;quoted text
        MVI     C,AROUND-MESG       ;length
        ELSE
        IF      NOT NUL FIRST
        LXI     H,FIRST
        ENDIF
        IF      NOT NUL BYTES
        MVI     C,BYTES
        ENDIF
        ENDIF                       ;nul bytes
        IF      NOT NUL SECOND
        LXI     D,SECOND
        ENDIF
        CALL    COMP2?
        POP     B
        POP     D
        POP     H
        IF      NOT CMFLAG OR NUL BYTES
        JMP     AROUND
```

Figure 4.19: Macro COMPAR to Perform a Binary Comparison

```
        ENDIF
        IF      NOT CMFLAG      ;one copy
COMP2?:                         ;compare routine
        LDAX    D               ;get char
        CMP     M               ;same?
        RNZ                     ;no
        INX     H
        INX     D               ;pointers
        DCR     C               ;and count
        JNZ     COMP2?          ;keep going
        RET
CMFLAG  SET     TRUE            ;only one
        ENDIF
        IF      NUL BYTES
MESG:   DB      FIRST           ;;text
        ENDIF
AROUND:                         ;;COMPAR
        ENDM
```

Figure 4.19 (continued)

parameters are missing, the registers must be loaded prior to referencing the macro. The macro call might look like this:

```
COMPAR  , , 8
```

If you want to determine whether the first and second parameters of a CP/M command line are identical, use the following macro reference:

```
COMPAR  FCB1, FCB2, 12
```

This will compare the 12 bytes starting at the first file control block to the 12 bytes in the second. The macro will set the zero flag if the two blocks are identical. The zero flag will be reset otherwise.

If you want to compare a memory block to a particular string of text, you can omit the third parameter. The first parameter then contains the text itself. The assembler finds the length of the block from the length of the first parameter. For example, the macro reference

```
COMPAR  '???', FCB1+9
```

will set the zero flag if the three characters starting at FCB1+9 are all question marks. (FCB1+9 contains the file type of the first parameter of a CP/M command line.)

An ASCII Comparison

Although each byte contains eight bits, the ASCII character set uses only the lower seven bits (0–6). Since the high-order bit, bit 7, is not needed in this case, it can be used to convey other information. Thus one 8-bit byte can be divided into a 1-bit flag followed by a 7-bit ASCII character. The byte does double duty. The CP/M system uses this method to denote file protection and thus reduce the likelihood of accidentally erasing important files. For example, CP/M file names consist of a primary name and an extension of up to three characters. The extension often suggests the kind of information contained in the file (FOR for FORTRAN, BAS for BASIC, and so on). If the high-order bit of the first character of the extension is set, CP/M considers the file to be write protected. On the other hand, if this bit is reset, the file can be deleted or altered. The remaining seven bits contain the ASCII character.

Suppose we want to ensure that a given file has the extension COM. It appears that we could use the macro reference

```
COMPAR    'COM', FCB1+9
```

for this purpose. However, this approach will fail whenever the given file is write protected. For example, the ASCII representation of the letter C is

```
100 0011
```

which we can write as

```
0100 0011
```

when the high-order bit is zeroed. However, if the file is write protected, the high-order bit is set. The pattern is as follows:

```
1100 0011
```

We therefore need a different version of the comparison macro, so that we can compare only the lower seven bits of each byte. The macro given in Figure 4.20 can be used for this purpose. Enter this macro into your library.

```
COMPRA    MACRO   FIRST, SECOND, BYTES
;;(Put current date here)
;;ASCII version (high bit is zeroed).
;;Inline macro to compare two memory areas.
;;Zero flag is set if both are the same,
```

Figure 4.20: Macro COMPRA to Perform an ASCII Comparison

```
;;first and second may be addresses,
;;third parameter is number of bytes.
;;First parameter may be a quoted string,
;;in which case there is no third parameter.
;;All three parameters may be omitted.
;;Register A is altered.
;;
;;Usage:      COMPRA  FCB1, FCB2, 11
;;          COMPRA  'COM', FCB1+9
;;          COMPRA  , FCB1+1, 11
;;
            LOCAL   MESG, AROUND
            PUSH    H
            PUSH    D
            PUSH    B
            IF      NUL BYTES
            LXI     H, MESG             ;quoted text
            MVI     C,AROUND-MESG       ;length
            ELSE
            IF      NOT NUL FIRST
            LXI     H,FIRST
            ENDIF
            IF      NOT NUL C
            MVI     C,BYTES
            ENDIF
            ENDIF                       ;nul bytes
            IF      NOT NUL SECOND
            LXI     D,SECOND
            ENDIF
            CALL    COMP2?
            POP     B
            POP     D
            POP     H
            IF      NOT CMFLAG OR NUL BYTES
            JMP     AROUND
            ENDIF
            IF      NOT CMFLAG          ;one copy
COMP2?:                                 ;compare routine
            LDAX    D                   ;get char
```

Figure 4.20 (continued)

```
        ANI     7FH         ;mask bit 7
        PUSH    B
        MOV     C,A
        MOV     A,M
        ANI     7FH
        CMP     C           ;same?
        POP     B
        RNZ                 ;no
        INX     H
        INX     D           ;pointers
        DCR     C           ;and count
        JNZ     COMP2?      ;keep going
        RET
CMFLAG  SET     TRUE        ;only one
        ENDIF
        IF      NUL BYTES
MESG:   DB      FIRST       ;;text
        ENDIF
AROUND:                     ;;COMPRA
        ENDM
```

Figure 4.20 (continued)

A MACRO TO RAISE LOWERCASE LETTERS TO UPPERCASE

Any lowercase letters given on a CP/M command line are automatically raised to uppercase. However, if the user inputs information while a program is executing, uppercase and lowercase letters remain distinctly different. For example, suppose that a program displays the statement

```
DELETE ALL FILES?
```

It is not sufficient to test the user response with the statement

```
CPI   'Y'
```

because the input might be either uppercase or lowercase. Of course, it is possible to consider both possibilities with additional instructions. For example:

```
CPI   'Y'
JZ    . . .
CPI   'y'
JZ    . . .
```

A more efficient approach, however, is to use the macro given in Figure 4.21 to raise a lowercase letter to uppercase. The macro is referenced just before the comparison is made:

```
UCASE
CPI     'Y'
JZ      . . .
```

We can understand the operation of this macro by considering the ASCII coding of alphabetic characters. For example, the uppercase letter Y and the lowercase letter y differ by only one bit. The lower seven bits of each are as follows:

```
Y    101 1001    (uppercase)
y    111 1001    (lowercase)
```

```
UCASE   MACRO   REG
;;(Put current date here)
;;Inline macro to convert a character in any
;;register to uppercase.
;;Omit parameter for register A.
;;
;;Usage:  UCASE
;;       UCASE   C
;;
        LOCAL   NOTUP?
        IF      NOT NUL REG
        PUSH    PSW             ;save
        MOV     A,REG           ;get value
        ENDIF
        CPI     'Z'+7           ;uppercase?
        JC      NOTUP?          ;no
        ANI     5FH             ;make uppercase
NOTUP?:
        IF      NOT NUL REG
        MOV     REG,A           ;put back
        POP     PSW             ;restore
        ENDIF
                                ;;UCASE
        ENDM
```

Figure 4.21: Macro UCASE to Convert Lowercase Letters to Uppercase

The patterns for the other alphabetic characters are similar. The example shows that we can convert a lowercase letter to uppercase by resetting bit 5. The operation we want is a logical AND with the value of 5F hex.

```
        y    111 1001    (lowercase)
AND     5F   101 1111
        Y    101 1001    (uppercase)
```

This approach works properly for lowercase letters. It also gives the desired answer when applied to uppercase letters:

```
        Y    101 1001    (uppercase)
AND     5F   101 1111
        Y    101 1001    (uppercase)
```

That is, we can use the same operation on either uppercase or lowercase letters and we will get uppercase letters. Remember that this technique is designed to work only for letters.

Consider, for example, what would happen if we performed a logical AND with the value 5F hex and the ASCII number 8. The bit patterns are as follows:

```
        8    011 1000    (number 8)
AND     5F   101 1111
             001 1000    (control-X)
```

We have converted the number 8 into the character control-X. We must therefore be careful to apply the conversion routine only to letters. (There are several special characters, such as the braces, that are located with the lowercase letters. However, this is not likely to be a problem.)

The macro contains the following instructions:

```
CPI   'Z'+7
JC    . . .
```

The CPI instruction determines whether the character is lowercase. The value of the lowercase letter 'a' is seven greater than the value of an uppercase letter Z. So if the character has a value less than a lowercase letter 'a', the JC instruction causes a branch around the logical AND operation. (If we consider it important enough, we could add a second test to the program for characters that have values greater than z. This would ensure that the program would only try to convert characters from 'a' to 'z'. However, this is a minor point, because there are only a few characters in the ASCII range beyond z.)

A second feature of UCASE is the optional parameter. If the parameter

is omitted, the character is expected to be in the accumulator. However, if a register is given as a parameter, the assembler will insert additional instructions to operate on the character in the given register. For example, the macro reference

```
UCASE   C
```

will generate the additional instructions

```
PUSH    PSW
MOV     A,C
```

at the beginning of the macro expansion and the instructions

```
MOV     C,A
POP     PSW
```

at the end.

We will usually include macro UCASE in programs that require input from the operator.

A MACRO TO CONVERT AN AMBIGUOUS FILE NAME TO AN UNAMBIGUOUS FILE NAME

In Chapter 7 we will write a program for renaming disk files. The program will allow ambiguous file names, and the original file name will be given before the new file name.

If we give the command

```
RENAME SORT.PAS *.BAK
```

we want the result to be the same as if we had given the command

```
RENAME SORT.PAS SORT.BAK
```

That is, the file name *.BAK must be changed into SORT.BAK. This conversion occurs in two steps.

The CP/M system will convert the first parameter to a slightly different form and place it in the file control block at 5C hex. This location is given the symbolic name FCB1 (or sometimes simply FCB) in this book. CP/M removes the decimal point separating the primary name from the extension. It then fills out the four characters of the primary name to eight characters by using blanks, and it places them in memory starting at 5D hex. The extension name is placed immediately after the primary name.

The second parameter is placed into memory starting at 6C hex. The symbolic name FCB2 refers to this location. CP/M converts the asterisk into eight question marks and puts them into memory starting at 6D hex.

The extension name is placed after the primary name. The second parameter becomes ????????.BAK.

At some point, the question marks in the second file name will have to be converted by our program into the four letters 'SORT' and four blanks corresponding to the first file name.

Macro AMBIG, given in Figure 4.22, can be used to convert an ambiguous

```
AMBIG   MACRO   OLD, NEW
;;(Put current date here)
;;Inline macro to change ambiguous file name
;;at FCB NEW to match FCB OLD.
;;
;;Usage:  AMBIG   FCB1, FCB2
;;
        PUSH    H
        PUSH    D
        PUSH    B
        LXI     H,NEW+1
        LXI     D,OLD+1
        MVI     C,11        ;number of char
AMB2?:
        MVI     A,'?'
        CMP     M           ;question mark?
        JNZ     AMB3?       ;no
;
;copy one char from original to new
;
        LDAX    D           ;get old char
        MOV     M,A         ;put into new
AMB3?:
        INX     H           ;new
        INX     D           ;orig
        DCR     C           ;count
        JNZ     AMB2?
        POP     B
        POP     D
        POP     H
                            ;;AMBIG
        ENDM
```

Figure 4.22: Macro AMBIG to Convert an Ambiguous File Name to an Unambiguous File Name

file name located at one address to an unambiguous file name located at another address. In this example, the address of the unambiguous file name is the first parameter (OLD) and the address of the ambiguous file name is the second parameter (NEW). Each character is examined, one at a time. Whenever a question mark is found in the ambiguous file name, it is replaced by the corresponding character of the unambiguous file name. For example, the first question mark is replaced by S, the second by O, and so forth. Copy this macro into your macro library.

Macro AMBIG begins by saving the original contents of the HL, DE, and BC registers. Then HL and DE are given the addresses corresponding to the parameters NEW and OLD. Register C is loaded with the value of 11, the file name length (8 + 3).

The accumulator is loaded with a question mark. Then each character in the new name is compared to the question mark in the accumulator. The instruction is

```
CMP     M
```

If a question mark is discovered, the corresponding character is copied from the old name. The instructions are as follows:

```
LDAX    D
MOV     M,A
```

After each comparison, the count in register C is decremented. When the value reaches zero, the routine is finished. The original contents of the registers are restored by POP statements.

A MACRO TO MOVE THE UPPER FOUR BITS TO THE LOWER POSITION

The three methods of representing numbers in a computer are ASCII, binary, and binary-coded decimal (BCD). ASCII numbers require seven bits, so each byte can store a maximum of one ASCII character (digit). With binary representation, we can code values from 0 to 255 decimal (one less than 2^8) in a single byte. With BCD mode, each digit is coded with four bits. Thus, a byte can represent BCD numbers from 0 to 99.

The BCD method is nothing more than a hexadecimal coding, except that the hex digits A – F are not used. Therefore, a routine that converts a binary number to hexadecimal can also be used to decode a BCD number. In the next chapter we will write a macro for converting a binary number to two hexadecimal characters.

There will be occasions, however, when we are only interested in the left

character (or nibble) of a BCD or hexadecimal number (for example, in macro OUTHEX in Chapter 5). Therefore, we will now write a macro for obtaining this upper half of the byte. Macro UPPER, shown in Figure 4.23, first rotates the upper four bits down to the lower four bits (by performing the RAR instruction four times), and then zeros the new upper four bits by performing a logical AND with the value 0F hex. If the optional parameter is provided, the operation is performed on the register name (including memory) given as the parameter. Incorporate this macro into your library and enter the name in the directory.

```
UPPER     MACRO   REG
;;(Put current date here)
;;Macro to move the upper 4 bits of the
;;accumulator to the lower 4 bits. The
;;new upper 4 bits are zeroed.
;;Use this macro to isolate the left
;;character of packed BCD numbers.
;;
;;Usage:  UPPER                   ;rotate down
;;        OUTHEX                  ;print
;;
        IF      NOT NUL REG
        PUSH    PSW             ;save A
        MOV     A,REG           ;move to A
        ENDIF
        RAR                     ;move to
        RAR                     ;low half
        RAR
        RAR
        ANI     0FH             ;mask upper
        IF      NOT NUL REG
        MOV     REG,A           ;put back
        POP     PSW             ;restore A
        ENDIF
                                ;;UPPER
        ENDM
```

Figure 4.23: Macro UPPER to Move the Upper Four Bits of a Byte to the Lower Four Bits

A MACRO TO PERFORM 16-BIT SUBTRACTION

Both the 8080 and Z80 CPUs can perform 8-bit addition and 8-bit subtraction with and without considering the carry flag. In addition, the Z80 can perform 16-bit subtraction with carry. It is important to note, however, that the Z80 double-register subtraction always includes the carry in the subtraction. Therefore, we must reset the carry flag before we do the subtraction. Of course, the carry flag reflects the result of the subtraction.

The final macro in this chapter is given in Figure 4.24. It can be used to perform 16-bit subtraction without considering the carry flag. We will need to use macro SBC in several programs to calculate the distance from one memory location to another.

This 8080 version of a double-register subtraction calculates the difference between the value in HL and the value in DE. The result is placed in HL. The state of the carry flag at the beginning of the calculation is not used, but the carry flag at the end of the process correctly reflects the result. That is, if the original value in DE is larger than that in HL, the carry flag will be set at the conclusion of the calculation. This macro is

```
SBC         MACRO
;;(Put current date here)
;;Inline macro to subtract DE from HL.
;;The result is in HL. This is almost
;;the Z80 SBC HL,DE opcode.
;;
;; Usage:  SBC
            SBC         HL,DE
;;
            MOV         A,L
            SUB         E
            MOV         L,A
            MOV         A,H
            SBB         D
            MOV         H,A
                                    ;;SBC
            ENDM
```

Figure 4.24: Macro SBC to Perform 16-Bit Subtraction without Carry

equivalent to the two Z80 instructions

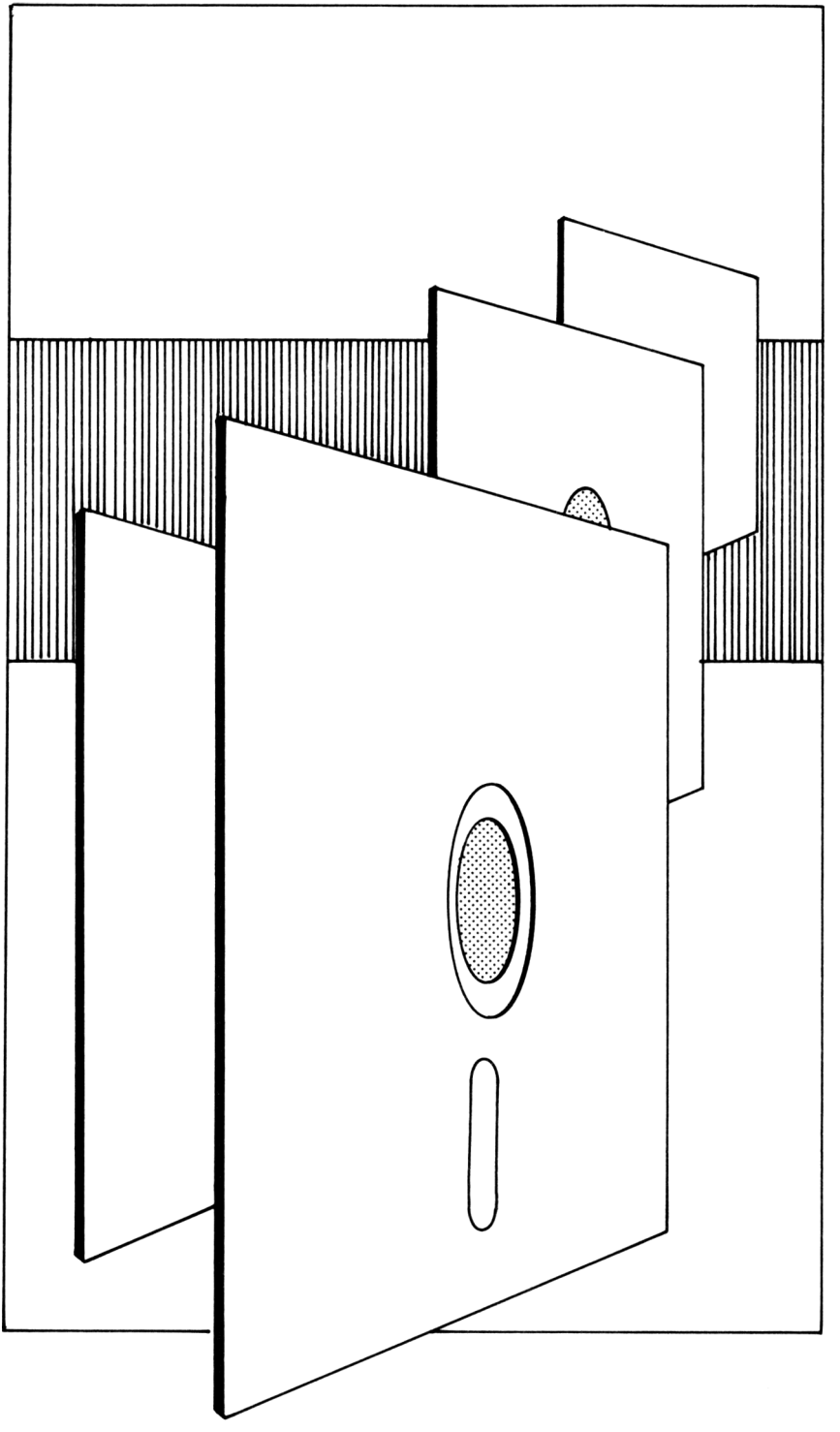
```
OR    A
SBC   HL,DE
```

SUMMARY

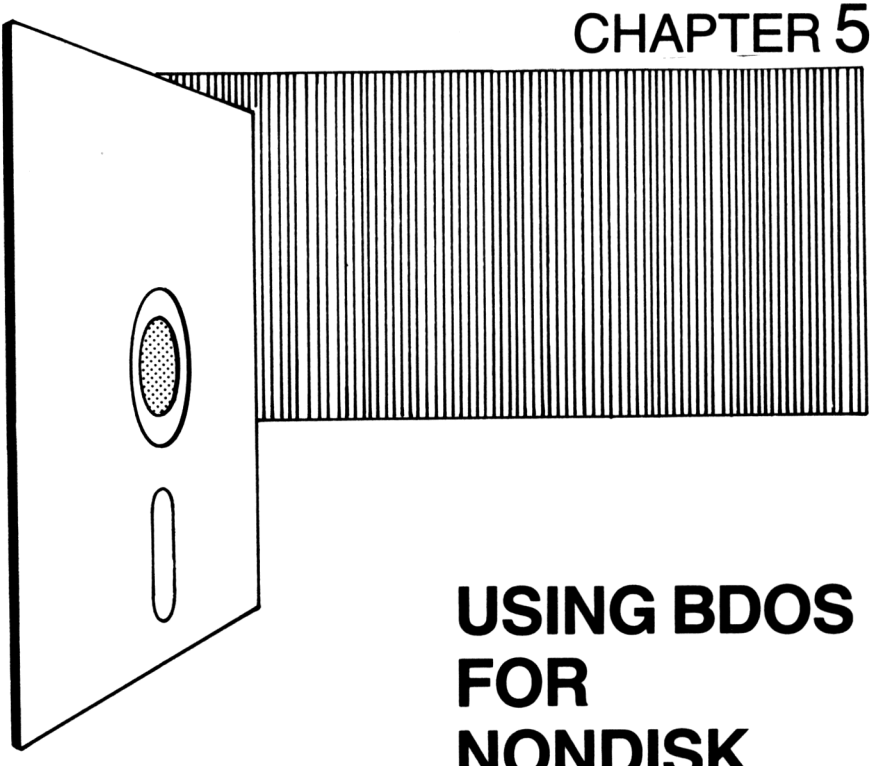
In this chapter, we have explored the importance of macro processing and we have developed several elementary macros. We will incorporate these macros in the programs we write in later chapters. It should be noted that these macros all have a common feature—they do not perform BDOS calls. In the next three chapters we will consider macros that use BDOS calls, and we will write programs that incorporate these macros.

The directory of your macro library should now look like this:

```
;;Macros in this library                          Flags
;;AMBIG      MACRO    OLD, NEW                   (none)
;;COMPAR     MACRO    FIRST, SECOND, BYTES       CMFLAG
;;COMPRA     MACRO    FIRST, SECOND, BYTES       CMFLAG
;;ENTER      MACRO                               (none)
;;EXIT       MACRO    SPACE?                     (none)
;;FILL       MACRO    ADDR, BYTES, CHAR          FLFLAG
;;MOVE       MACRO    FROM, TO, BYTES            MVFLAG
;;SBC        MACRO                               (none)
;;UCASE      MACRO    REG                        (none)
;;UPPER      MACRO    REG                        (none)
;;VERSN      MACRO    NUM                        (none)
```



CHAPTER 5



USING BDOS FOR NONDISK OPERATIONS

INTRODUCTION

In this chapter we will learn how to perform console input, console output, and list output by using the CP/M basic disk-operating system (BDOS). We will develop a number of useful macros to make these tasks easier. Along the way, we will write macros that convert binary numbers to decimal and hexadecimal characters, and hexadecimal characters to binary numbers. Finally, we will incorporate these macros into four executable programs that show us more about CP/M's organization. The program CPU determines whether an 8080 or a Z80 CPU is being used; IOBYTE displays and alters the CP/M IOBYTE feature we designed in Chapter 3; GO branches to an absolute address in memory; PAGE ejects one or more pages on the printer.

BDOS CALLS

As we saw in Chapter 1, the CP/M operating system divides the computer memory into several distinct regions. The upper portion of memory is called the full disk-operating system (FDOS) and is further divided into two regions. The basic input-output system (BIOS) occupies the upper part of FDOS, and the basic disk-operating system (BDOS) occupies the lower part of FDOS. In Chapter 3 we studied the organization of the BIOS and added several new features. We will now consider the BDOS.

The BIOS contains the primitive routines for operating the console, the printer, and the disks. These routines must be specifically programmed for the actual physical devices that are attached to the computer. Different computers will have different versions of BIOS. It is possible for CP/M executable programs to perform input and output operations by communicating directly with the BIOS. However, it is easier to use the BDOS as an intermediate to BIOS. All console, printer, and disk operations can be performed through the BDOS by using a special location in memory. Because BDOS is device independent, programs that operate on one CP/M computer will also operate on any other CP/M computer, even though the hardware and BIOS routines may be different.

Using BDOS to perform peripheral operations is not only more versatile, it is also more convenient. Recall that the first three bytes in memory, starting at address 0, contain a jump instruction to the warm-start vector of the BIOS. The next byte, at address 3, contains the IOBYTE. The following byte, address 4, indicates two things: the current disk drive and the current user number. The next three bytes, starting at address 5, contain a jump into the BDOS. This is the location that can always be called when console, printer, and disk operations are needed. Contrast this single jump address into BDOS to the multiple jump vectors at the beginning of the BIOS. The BIOS uses a separate entry point for each different operation.

We will now consider some simple BDOS operations.

Nondisk BDOS Function Numbers

When an executable program interacts with the peripherals through the BDOS, it calls the BDOS entry point at address 5. At this time, the C register of the 8080 or Z80 CPU contains a function number indicating the desired operation. The information sent by the program is placed in the E register if the value is byte size, or in the DE register pair if it is two bytes. Information is usually sent back to the calling program in the accumulator if byte size or in HL if it is two bytes.

Table 5.1: The Nondisk BDOS Functions

Function number (in C)	Operation	Value sent	Value returned
1	Read console		character in A
2	Write console	character in E	
3	Read reader		character in A
4	Write punch	character in E	
5	Write list	character in E	
6	Direct console I/O	FF (input) character (output)	0 = not ready or character in A
7	Determine IOBYTE		byte in A
8	Set IOBYTE	in E	
9	Print buffer	address in DE	
10	Read buffer	address in DE	
11	Return console status		byte in A
12	Return CP/M version		byte in A and L

We can perform many different operations with BDOS calls. We can divide the functions into two groups. One group deals with the console, reader, punch, and list devices. The other group performs disk operations, which will be considered in the next chapter. Here we will look at the nondisk functions. Table 5.1 summarizes the first 12 BDOS functions. These functions deal with the four logical devices—the console, the printer, the list, and the punch—as well as operations involving the IOBYTE and the CP/M version number. We will be explaining these operations as the chapter proceeds. Let us now consider a macro for performing general BDOS calls.

A MACRO TO PERFORM BDOS CALLS

The BDOS functions all work in the same way. Address 5 is called with the function number in register C. Information is sent in the E or DE register and returned in the accumulator or HL. Because the contents of the CPU registers change during the BDOS operations, it is usually necessary to save the registers on the stack before calling the BDOS. The registers are then restored after the return from BDOS. One note of caution, however: if we save the accumulator and flag register with a PUSH PSW instruction and then restore them with POP PSW, we will lose any

information that was returned from BDOS in the accumulator. Therefore, the accumulator should not be saved during input operations.

The macro shown in Figure 5.1 will be referenced by several other macros we will write. Add it to your macro library in alphabetic order. Be sure to enter the name into the directory at the beginning of the library.

Our macros are usually designed for direct, inline use. If a subroutine or a line of text must be included, there is a branch to get around the obstruction. Macro SYSF, however, is always referenced as a subroutine. We do not have to include a branch around the routine, because macro SYSF will generally be called by another macro that already includes the branch. However, if you use macro SYSF directly in the line of instructions, you must provide a branch around the routine.

Macro SYSF has two dummy parameters. The first parameter is the function number, which is loaded into register C. The second parameter is optional. It will be used only when we must transfer a byte from the accumulator to register E prior to calling BDOS for output (see macro PCHAR in Figure 5.3).

The macro begins by saving the HL, DE, and BC registers on the stack and loading the function number (the first parameter) into register C. If the optional second parameter is provided, the value in the accumulator is moved into register E and the accumulator is saved with PUSH PSW. The BDOS address is called to perform the desired function. After returning from BDOS, the accumulator is restored with POP PSW if it was previously saved. The other registers are restored and control returns to the calling program.

A MACRO TO READ A SINGLE CONSOLE CHARACTER

The first two BDOS functions are very important. Function 1 is used to read a single character from the console, and function 2 is used to write a single character on the console. Actually, these two functions are not complementary. When a console character is read with function 1, it is also displayed on the terminal at that time. Function 3 is similar to function 1 except that the character is obtained from the logical reader rather than from the console.

We can obtain a single character from the console by placing the function number 1 in register C and calling address 5. The 8080 instructions are as follows:

```
MVI     C,1
CALL    5
```

```
SYSF    MACRO   FUNC, AE
;;(Put current date here)
;;Macro to generate BDOS calls.
;;FUNC is BDOS function number for C.
;;THIS IS NOT AN INLINE MACRO.
;;Move A to E if there is a second parameter.
;;
;;Usage: OPEN:   SYSF   15
;;      PCHAR:  SYSF   2,AE
;;
        PUSH    H
        PUSH    D
        PUSH    B
        MVI     C,FUNC
        IF      NOT NUL AE
        MOV     E,A             ;console and list
        PUSH    PSW             ;save A
        CALL    BDOS
        POP     PSW
        ELSE
        CALL    BDOS
        ENDIF
        POP     B
        POP     D
        POP     H
        RET
                                ;;SYSF
        ENDM
```

Figure 5.1: Macro SYSF to Generate a BDOS Call

We can generate these instructions by using macro SYSF with the appropriate parameter. When the instructions are executed, BDOS calls the BIOS vector that performs console input. The next character entered from the console is read by the BIOS. Control then returns to the calling program through BDOS. The character is available in the accumulator.

Occasionally we will need to check the console status to determine whether the user has pressed a console key. We will then place function number 11 in register C and call the BDOS address. On return from BDOS, the accumulator contains a value of FF hex if a console character

has been typed. The accumulator contains a value of 0 otherwise.

When console input is performed with the BDOS function 1, the system waits until the console is ready. Because this function automatically performs a status check, it is not necessary to determine the console status by first making a call to BDOS function 11. On the other hand, if no character is typed, program execution ceases until a character is typed.

When BDOS function 1 is used, printable ASCII characters, such as the letters and digits, are displayed on the video screen as they are entered. Control characters such as the carriage return, line feed, tab (control-I), backspace (control-H), and control-C can also be read in this way, but they are not displayed on the screen.

Macro READCH is given in Figure 5.2. Add it to your macro library. If no parameter is given, this macro generates instructions to read one character from the console and then return the character in the accumulator. However, if a parameter is provided, the character is returned in the register given by the parameter.

```
READCH      MACRO     REG
;;(Put current date here)
;;Inline macro to read one character from
;;the console; character is returned in register
;;A unless a second parameter is given.
;;Macro needed: SYSF
;;
;;Usage:     READCH
;;          READCH    C
;;
            LOCAL     AROUND
            CALL      RDCH?
            IF        NOT NUL REG
            MOV       REG, A
            ENDIF
            IF        NOT CIFLAG
            JMP       AROUND
RDCH?:      SYSF      1
CIFLAG      SET       TRUE          ;only one copy
            ENDIF
AROUND:                             ;;READCH
            ENDM
```

Figure 5.2: Macro READCH to Read One Console Character

A MACRO TO WRITE A SINGLE CONSOLE CHARACTER

A program can perform console output by putting the character into register E, the value of 2 in register C, and calling the BDOS entry at address 5. Functions 4 and 5 are similar to function 2, the only difference being where the output is sent. If the function number in register C is 4, the byte in register E is sent to the punch device. If the function number is 5, the value is sent to the list device.

Macro PCHAR, shown in Figure 5.3, performs the BDOS function 2. We will use it frequently to send individual characters to the console, referencing it from other macros we write. Incorporate this macro into your library. Notice that macro SYSF is required.

Macro PCHAR can be used to display the byte that is present in the accumulator. The macro name is placed in the source program as though it were an operation code. This macro can also be used to display a particular constant that is known at assembly time. The constant is given as a

```
PCHAR       MACRO  PAR
;;(Put current date here)
;;Inline macro to print one console char.
;;Parameter, if present, is loaded into A.
;;Macro needed: SYSF
;;
;;Usage:     PCHAR
;;          PCHAR  '*'
;;
            LOCAL  AROUND
            IF     NOT NUL PAR
            MVI    A,PAR
            ENDIF
            CALL   PCH2?
            IF     NOT COFLAG
            JMP    AROUND
PCH2?:      SYSF   2,AE
COFLAG      SET    TRUE            ;only one copy
            ENDIF
AROUND:                            ;;PCHAR
            ENDM
```

Figure 5.3: Macro PCHAR to Display Single Characters on the Console

parameter to the macro reference. For example, if we want to print an asterisk, we can use the expression

```
PCHAR   '*'
```

When the assembler encounters the parameter, it generates an additional instruction to move the parameter into the accumulator. It thus generates the same instructions as the two lines

```
MVI     A,'*'
PCHAR
```

If two identical characters are needed, it is not necessary to give the parameter the second time:

```
PCHAR   '$'     ;print dollar sign
PCHAR           ;second dollar sign
```

These instructions will display two dollar signs. There is a potential problem, however, because the original value in the accumulator is lost. For example, suppose you want to print a particular character, then display the original value in the accumulator. You will first need to save the value that was originally in the accumulator. The program might look like this:

```
PUSH    PSW
PCHAR   '*'     ;print asterisk
POP     PSW
PCHAR           ;original character
```

A MACRO TO DISPLAY A CARRIAGE RETURN AND LINE FEED

PCHAR can be used to display single characters, but frequently we will find it necessary to display a carriage return followed by a line feed. Because this combination requires two references to PCHAR, we will write a very short macro called CRLF to make the task easier. Copy the macro shown in Figure 5.4 into your macro library.

Macro CRLF uses no parameters. It is referenced in a program wherever a carriage return and line feed are needed. The beginning of the macro calls the global subroutine CRLF2? to perform the desired operation. The subroutine first saves the accumulator on the stack, then references macro PCHAR twice. The accumulator is restored and control is returned to the beginning of the macro. A jump instruction allows the subroutine to be coded inline.

Two flags are needed with macro CRLF—COFLAG for macro PCHAR

```
CRLF        MACRO
;;(Put current date here)
;;Inline macro to send a
;;carriage return, line feed to console.
;;All registers saved including A.
;;Macro needed: PCHAR
;;
            LOCAL   AROUND
            CALL    CRLF2?
            IF      NOT CRFLAG      ;just one
            JMP     AROUND
CRLF2?:
            PUSH    PSW
            PCHAR   CR
            PCHAR   LF
            POP     PSW
            RET
CRFLAG      SET     TRUE            ;only one copy
            ENDIF
AROUND:                             ;;CRLF
            ENDM
```

Figure 5.4: Macro CRLF to Generate a Carriage Return and Line Feed

and CRFLAG for this macro. The latter flag ensures that there will only be one copy of subroutine CRLF2? and the corresponding jump instruction. Each additional reference to macro CRLF will only generate a call to subroutine CRLF2?.

A PROGRAM TO TEST MACROS SYSF, READCH, PCHAR, AND CRLF

The program shown in Figure 5.5 can be used to test macros SYSF, READCH, PCHAR, and CRLF. Type in the program, assemble it, and run it. The program begins with the usual macros ENTER and VERSN. Then macro CRLF is used to begin a new line. Macro PCHAR prints a colon for a prompt symbol and macro READCH waits for user input.

As soon as a single console character is typed, the program continues. An ASCII zero is subtracted from the user input. This operation converts the ASCII digits 0–9 to the corresponding binary digits. Of course, all

```
TITLE   'TEST PCHAR'
;
;(Put current date here)
;
FALSE   EQU     0
TRUE    EQU     NOT FALSE
;
BOOT    EQU     0           ;system reboot
BDOS    EQU     5           ;BDOS entry point
TPA     EQU     100H        ;program start
;
CIFLAG  SET     FALSE       ;for READCH
CRFLAG  SET     FALSE       ;for CRLF
COFLAG  SET     FALSE       ;for PCHAR
;
        MACLIB  CPMMAC
;
ORG     TPA
;
START:
        ENTER
        VERSN   '(current date)'
NEXT:
        CRLF
        PCHAR   ':'         ;prompt
        READCH              ;number of char
        SUI     '0'         ;make binary
        JZ      DONE        ;quit on zero
        MOV     C,A
        PCHAR   BLANK
LOOP:
        PCHAR   '*'
        DCR     C
        JNZ     LOOP
        JMP     NEXT
DONE:
        EXIT
;
        END     START
```

Figure 5.5: Program to Test Macros SYSF, READCH, PCHAR, and CRLF

other input characters are altered also. If the user inputs a value of 0, the program is terminated; otherwise, the value is saved in the C register. A blank is printed and the number of asterisks corresponding to the user input is displayed. The program then starts again.

An input in the range of 1—9 will produce as many asterisks. The uppercase letter A will give 17 asterisks and the lowercase letter 'a' will give 49 asterisks. Characters such as the dollar sign and percent symbol have ASCII values less than the digits, but because they are altered by the subtraction of an ASCII zero, they will produce several lines of asterisks.

PRINTING A STRING OF CHARACTERS

In the previous section, we used BDOS function 2 to display individual characters on the console, one at a time. However, frequently we will need to display a string of characters such as the expression

```
?FILE NOT FOUND
```

This is easily accomplished with BDOS function 9. The string is placed into memory and terminated with a dollar sign. The address of the beginning of the string is loaded into the DE register and the value of 9 is placed into register C. When BDOS is called, the string is displayed on the console. The dollar sign, of course, is not included in the display.

The program shown in Figure 5.6 demonstrates the use of BDOS function 9 to print a string of characters on the console. The program uses macro SYSF. Type in the program, assemble it, and execute it. The resulting console output should be as follows:

```
A test of BDOS function 9
```

In this program, the desired string begins with a carriage return and line feed. These two characters are embedded in the console buffer in this example. Previously we used macro CRLF for this purpose.

The remaining text, including the terminal dollar sign, is enclosed by apostrophes. The assembler places the text in memory immediately after macro SYSF, which is implemented as a subroutine. The JMP DONE statement provides a branch around both the subroutine and the string of characters.

A Macro to Print a String of Characters

Using function 9 to display a string of characters is more efficient than displaying individual characters with function 2. Nevertheless, we still

```
TITLE   'Print console buffer'
;
;(Put current date here)
;
BOOT    EQU     0               ;system reboot
BDOS    EQU     5               ;BDOS entry point
TPA     EQU     100H            ;transient program area
;
        MACLIB  CPMMAC
;
ORG     TPA
;
START:
        ENTER
        VERSN   '(current date).CONSOLE BUFFER'
        LXI     D,TEXT
        CALL    SEND
        JMP     DONE
SEND:
        SYSF    9
TEXT:
        DB      CR,LF,'A test of BDOS'
        DB      ' function 9$'
DONE:
        EXIT    BOOT            ;warm start
;
        END     START
```

Figure 5.6: Printing the Console Buffer

have to provide a branch around the string and a call to subroutine SEND. In this section we will write a new macro to further simplify the printing of strings. Our goal will be a macro called PRINT. Its use will be as simple as the following instruction:

```
PRINT   'A test of BDOS function 9'
```

That is, the parameter to the macro will be the string enclosed in apostrophes.

Make a copy of the program shown in Figure 5.6 and alter it to look like Figure 5.8. We will use this program to test macro PRINT shown in

```
PRINT       MACRO  TEXT
;;(Put current date here)
;;Inline macro to print a literal string.
;;Macro needed: SYSF
;;
;;Usage:     PRINT  'message'
;;          PRINT  <CR,LF, 'message'>
;;
            LOCAL  MESG, AROUND
            PUSH   D
            LXI    D,MESG
            CALL   PBUF?              ;print message
            POP    D
            JMP    AROUND
            IF     NOT PRFLAG         ;need subroutine
;
;print message on console up to $
;
PBUF?:
            SYSF   9
PRFLAG      SET    TRUE               ;no more copies
            ENDIF
;
MESG:       DB     TEXT,'$'
AROUND:
                                      ;;PRINT
            ENDM
```

Figure 5.7: Macro PRINT, Version 1

Figure 5.7. You can incorporate this macro into your macro library now, but we will be writing a more general version in the next section. Consequently, you may want to temporarily insert this version into Figure 5.8, the program to test the macro, rather than into your macro library. In that case, place it directly after the MACLIB CPMMAC statement.

When the assembler encounters the PRINT macro, it places the desired string into memory starting at the location MESG. A dollar sign is automatically placed at the end of the string so that CP/M will know where the buffer terminates. The original value in the DE register is saved on the stack with a push statement, then the DE register is loaded with the

```
TITLE   'Print console buffer'
;
;(Put current date here)
;
FALSE   EQU     0
TRUE    EQU     NOT FALSE
;
BOOT    EQU     0               ;system reboot
BDOS    EQU     5               ;BDOS entry point
TPA     EQU     100H            ;transient program area
;
PRFLAG  SET     FALSE           ;print console buffer
;
        MACLIB  CPMMAC
;
ORG     TPA
;
START:
        ENTER
        VERSN   '(current date)'
        PRINT   <CR,LF,'A test of BDOS function 9'>
DONE:
        EXIT    BOOT            ;warm start
;
        END     START
```

Figure 5.8: Program to Test Macro PRINT

address of MESG, the start of the string. Subroutine PBUF? is called to print the string. The DE register is restored with a POP command; a branch around both the subroutine and the string concludes the PRINT macro.

Several features should be noticed in this example. The symbol PRFLAG is initially set to FALSE so that only one copy of subroutine PBUF? is generated. PBUF? is a global variable, while the labels MESG and AROUND are local variables. They will appear in each expansion of the macro, but they will be different symbols. Finally, in the main program we have surrounded the parameter to macro PRINT with angle brackets:

```
<CR,LF,'A test of BDOS function 9'>
```

This step tells the assembler that the carriage return and line feed are to be included in the text.

Assemble the new program and execute it. The result should be the same as before.

Macro Print, Version 2

The macro we wrote in the previous section can be used to print strings of characters embedded in the source program, but we cannot print a dollar sign in this way. There will also be cases where we want to print a string stored at a particular memory location. We might not even know the location until execution time. We could adapt the previous macro for this purpose if we place a dollar sign at the end of the string, but this may not always be convenient. We will now rewrite macro PRINT so it can display a string located anywhere in memory or given as the macro parameter.

We will abandon the previous reference to BDOS function 9, which prints a string of characters, and we will use function 2 instead. We will print the characters one at a time using macro PCHAR. Macro PRINT calculates the string length and then determines the number of times to call the subroutine created by PCHAR. This may seem to be a step backward, but it is not really. This version has the ability to print strings from any memory location, and dollar signs can be embedded in the strings as well.

Incorporate the second version of macro PRINT, shown in Figure 5.9, into your macro library. Alter the test program in Figure 5.8 to look like Figure 5.10, using the file name PRN2. Notice that two flags, COFLAG and PRFLAG, are required. Also notice that no regular 8080 operation codes are shown in this example. There are only macro references. Assemble this program and execute it. Give the following CP/M command line:

```
PRN2 TEST OF PRINT
```

The program will respond with the following two lines:

```
The first 12 characters of the command line tail are:
TEST OF PRIN
```

This program contains three references to macro PRINT. The first two are similar to the previous uses. The desired string is printed on the console:

```
The first 12 characters of the command line tail are:
```

The third reference, however, is different. The presence of the second parameter in the macro reference is a signal to the assembler that the first

parameter contains the address of the string rather than the string itself. The first parameter references the beginning of the string at DBUFF+2. When any program is executed, CP/M places the command line tail in memory starting at 82 hex. The HL register is therefore loaded with the address of 82 hex (DBUFF+2).

Let us see how macro PRINT works by writing an executable program.

```
PRINT       MACRO   TEXT, BYTES
;;(Put current date here)
;;Inline macro to print string on console.
;;TEXT is address of string, BYTES is length.
;;TEXT may be in quotes if BYTES is omitted.
;;Macro needed: PCHAR
;;
;;Usage:     PRINT   FCB1+1, 11
;;          PRINT   'end of file'
;;          PRINT   <CR,LF, 'message'>
;;          PRINT   , 12
;;
            LOCAL   AROUND, MESG
            PUSH    H
            PUSH    B
            IF      NUL BYTES
            LXI     H,MESG
            MVI     B,AROUND-MESG
            ELSE
            IF      NOT NUL TEXT
            LXI     H,TEXT
            ENDIF
            MVI     B,BYTES
            ENDIF
            CALL    PBUF?
            POP     B
            POP     H
            IF      NOT PRFLAG OR NUL BYTES
            JMP     AROUND
            ENDIF
            IF      NOT PRFLAG
```

Figure 5.9: Macro PRINT, Version 2

```
PBUF?:      MOV     A,M
            PCHAR
            INX     H
            DCR     B
            JNZ     PBUF?
            RET
PRFLAG      SET     TRUE
            ENDIF
            IF      NUL BYTES
MESG:       DB      TEXT
            ENDIF                       ;;PRINT
AROUND:
            ENDM
```

Figure 5.9 (continued)

```
TITLE   'Print console buffer'
;
;(Put current date here)
;
FALSE       EQU     0
TRUE        EQU     NOT FALSE
;
BOOT        EQU     0           ;system reboot
BDOS        EQU     5           ;BDOS entry point
TPA         EQU     100H        ;transient program area
DBUFF       EQU     80H         ;default buffer
;
COFLAG      SET     FALSE       ;console output
PRFLAG      SET     FALSE       ;print console buffer
;
            MACLIB  CPMMAC
;
ORG         TPA
;
START:
            ENTER
```

Figure 5.10: Program to Test Version 2 of Macro PRINT

```
        VERSN   '(current date)'
        PRINT   <CR,LF,'The first 12 characters of '>
        PRINT   <'the command line tail are: ',CR,LF>
        PRINT   DBUFF+2, 12
DONE:
        EXIT    BOOT            ;warm start
;
        END     START
```

Figure 5.10 (continued)

A PROGRAM TO DISCOVER WHICH CPU IS BEING USED

The 8080 (and 8085) instruction set is incorporated into the much larger set of instructions used by the Z80 CPU. Consequently, 8080 executable programs can usually be run on a Z80 computer. However, computer programs that use the special features of the Z80, such as block moves and relative jumps, will not run on an 8080 computer.

Because of this difference, it may be necessary for a computer program to determine which CPU is being used. For example, if a program requires the special Z80 instructions, it could terminate execution when it is run on an 8080. Alternatively, two different sets of algorithms could be provided. The more efficient version could be used when the program is run on a Z80. Otherwise, the 8080 version could be selected.

Because the 8080 and Z80 CPUs respond differently to arithmetic operations, they can be distinguished easily. The difference lies in the behavior of the parity flag. The flag correctly reflects the result of logical operations for both the 8080 and the Z80 CPUs. However, for arithmetic operations the results are different. For the 8080, the flag reflects the parity of the result, just as for logical operations. The Z80, however, sets the parity flag only if there is overflow (from bit 6 to 7) during an arithmetic operation. For this reason, the parity flag on the Z80 is called a parity/overflow flag.

We can distinguish the 8080 and Z80 CPUs by using the following three instructions:

```
XRA     A
DCR     A
JPE     NOTZ80
```

The first of these instructions performs an exclusive OR on the accumulator with itself. This logical operation zeros the accumulator. It also sets the parity flag (meaning parity is even) on both the 8080 and the Z80 CPUs, because there is an even number of ones (zero) in the result.

The next instruction decrements the accumulator, giving a value of FF hex. This arithmetic operation will leave the 8080 parity flag set, because there is an even number of ones (eight). However, the Z80 parity/overflow flag is reset by the decrement operation because there is no overflow. The 8080 CPU will branch at the JPE instruction because the 8080 parity flag is set. The Z80 CPU, however, will not branch because the parity/overflow flag is reset.

The above three lines could be incorporated into a Z80-only program to detect when it was run on an 8080 CPU. Let us see how this works by writing a short assembly language program.

The program given in Figure 5.11 will print the expression

```
CPU is Z80
```

when run on a Z80 computer. Otherwise the expression

```
CPU is 8080
```

will be displayed. Create a disk file named CPU. Type in the program, assemble it, and execute it.

The program begins with the usual ENTER and VERSN macros. The PRINT macro displays the beginning of the message. The CPU type is determined by the next three lines. If the CPU is 8080, the program branches to the label NOTZ80 and prints the message '8080'. Otherwise the program continues and prints the message 'Z80'.

Before we write our next executable program we will need to add two macros to our library. The first macro converts binary numbers to hexadecimal characters and displays them on the console. The second macro determines the CP/M version number.

```
TITLE    'CPU tells if 8080 or Z80'
;
;(Put current date here)
;
;Usage:    CPU
```

Figure 5.11: Program CPU to Determine whether CPU Is 8080 or Z80

```
;
FALSE     EQU     0
TRUE      EQU     NOT FALSE
;
BOOT      EQU     0           ;system reboot
BDOS      EQU     5           ;BDOS entry point
FCB1      EQU     5CH         ;input FCB
FCB2      EQU     6CH         ;2nd parameter
DBUFF     EQU     80H         ;default buffer
TPA       EQU     100H        ;transient program area
;
;Set flags in main program so only one
;copy of certain subroutines will be generated.
;Place set lines before MACLIB call.
;
COFLAG    SET     FALSE       ;console output
PRFLAG    SET     FALSE       ;print console buffer
;end of flags
;
          MACLIB  CPMMAC
;
ORG       TPA
;
START:
          ENTER
          VERSN   '(current date).CPU'
          PRINT   'CPU is'
          XRA     A           ;zero
          DCR     A
          JPE     NOTZ80
          PRINT   ' Z80'
          JMP     DONE
NOTZ80:
          PRINT   ' 8080'
DONE:
          EXIT
;
          END     START
```

Figure 5.11 (continued)

A MACRO TO CONVERT BINARY TO HEXADECIMAL

A binary-to-hexadecimal conversion is needed in many of the programs in this book. Any eight-bit binary value can be represented as two hexadecimal characters; the resulting hex number is in the range 0—FF hex.

As you know, information is stored in a computer as a sequence of binary digits (0 or 1), with each digit being called a bit, and a group of eight bits being called a byte. Sometimes we need to determine the value of a particular byte. However, we cannot simply transfer the byte to the console, because the console uses ASCII, a seven-bit code. For example, the binary number

0100 1011

has a hexadecimal value of 4B. However, this bit pattern corresponds to the ASCII letter K. So if this byte were sent to the console, we would see the letter K. We need a routine to transmit the ASCII numeral 4 and then the ASCII letter B. This is called a binary-to-hexadecimal routine.

Notice that for the above binary number, the upper four bits correspond to the left hex character (4) and the lower four bits correspond to the right hex character (B):

Nibble	*ASCII pattern*	*Character*
0100	0011 0100	4
1011	0100 0010	B

Notice that we need to display the left character before the right character. Consequently, we must rotate the upper four bits to the lower position. The new upper bits are then zeroed. When this happens, the pattern

0100 1101

becomes

1101 0100

and then

0000 0100

We must be careful to save the original byte prior to the rotation and zeroing, or the right nibble will be lost.

Copy macro OUTHEX, shown in Figure 5.12, into your macro library. This macro is used to convert a binary number to two hex characters that are printed on the console screen. If the optional parameter is omitted, the

```
OUTHEX      MACRO   REG
;;(Put current date here)
;;Inline macro to convert binary number in
;;REG to two hex characters and print them.
;;Byte initially in A if REG omitted.
;;Macro needed: PCHAR
;;
            LOCAL   AROUND,HEX1?,HEX2?
            IF      NOT NUL REG
            MOV     A,REG
            ENDIF
            CALL    OUTHX?
            IF      NOT CXFLAG
            JMP     AROUND
OUTHX?:     PUSH    B
            MOV     C,A             ;save
            RAR
            RAR
            RAR
            RAR
            CALL    HEX1?           ;high byte
            MOV     A,C
            CALL    HEX1?           ;low byte
            MOV     A,C             ;restore
            POP     B
            RET
HEX1?:      ANI     0FH             ;output hex byte
            ADI     '0'             ;make ASCII
            CPI     '9'+1           ;0-9?
            JC      HEX2?           ;yes
            ADI     'A'-'9'-1       ;make A-F
HEX2?:
            PCHAR                   ;to console
            RET
CXFLAG      SET     TRUE
            ENDIF
AROUND:                             ;;OUTHEX
            ENDM
```

Figure 5.12: Macro OUTHEX to Display a Binary Byte in Hexadecimal

binary number in the accumulator is converted. If the binary number is located in another register or in memory, the parameter references the location.

Two different algorithms can be used to convert a four-bit nibble to an ASCII character. The basic problem is to convert binary numbers from 0–15 to the ASCII digits 0–9 and the ASCII letters A–F. We need to convert the binary numbers to their ASCII equivalent expressed in hexadecimal notation, that is, to the base 16.

Let us study the bit patterns for the first ten numbers. The following list gives the values in decimal, binary, ASCII, and hex:

Decimal	*Binary*	*ASCII*	*Hex*
0	0000	0011 0000	0
1	0001	0011 0001	1
2	0010	0011 0010	2
3	0011	0011 0011	3
4	0100	0011 0100	4
5	0101	0011 0101	5
6	0110	0011 0110	6
7	0111	0011 0111	7
8	1000	0011 1000	8
9	1001	0011 1001	9

You can see from this list that there is a constant difference between these binary numbers and their ASCII counterparts. The ASCII zero has a hexadecimal value of 30 and the binary zero is 0, leaving a difference of 30 hex. We call this difference the ASCII *bias*. Thus a binary number in the range 0–9 can be converted to its ASCII equivalent by adding the ASCII bias. If the number is in the accumulator, the following instruction makes the conversion:

```
ADI   '0'
```

If the nibble has a value greater than 9, the binary-to-hex conversion is different. The patterns for this group are as follows:

Decimal	*Binary*	*ASCII*	*Hex*
10	1010	0100 0001	A
11	1011	0100 0010	B
12	1100	0100 0011	C
13	1101	0100 0100	D
14	1110	0100 0101	E
15	1111	0100 0110	F

By studying this list, we can see that the offset between the binary and ASCII values is 37 hex. Thus, we can make the conversion by adding the offset of 37 hex to this second group of numbers.

We perform the binary-to-hex conversion by first adding the ASCII bias of 30 hex. We use the ADI '0' instruction for this. If the original nibble was in the range 0–9, the result is the corresponding ASCII value from 0–9. However, if the original nibble was in the range 10–15, we add an additional 7, the remainder of the larger bias. This produces the corresponding ASCII characters A–F. This additional amount is one less than the difference between an ASCII A and an ASCII 9. Therefore we use the following instruction:

```
ADI    'A'-'9'-1    ;make A-F
```

The assembler determines that the operand has a value of 7. In this way, a binary two (0010) becomes the ASCII numeral 2. However, a binary ten (1010) becomes the ASCII letter A.

A shorter and faster algorithm is sometimes used for the binary-to-ASCII conversion, but it is more difficult to follow. The instructions from ADI '0' to ADI 'A'–'9'–1 are replaced by the following:

```
ADI     90H
DAA
ACI     40H
DAA
```

This approach uses the decimal adjust accumulator (DAA) operation. The DAA command is designed for BCD arithmetic. After each add instruction, the DAA command is given. This operation adds 6 to each nibble if the value is greater than 9.

Let us consider this method of binary-to-ASCII hex conversion for a binary two and a binary ten (hexadecimal A):

	Binary Two	*Binary Ten*
Original value	0000 0010	0000 1010
90 hex	1001 0000	1001 0000
ADI	1001 0010	1001 1010
DAA	1001 0010	0000 0000
40 hex	0100 0000	0100 0000
ACI	1101 0010	0100 0001
DAA	0011 0010	0100 0001
ASCII value	2	A

For the binary two, the first DAA operation does not change the value.

The second DAA instruction converts the 1101 of the left nibble to a 0011 by adding the value of 6. The result is 32 hex, the ASCII numeral 2. In the case of the binary ten, the first DAA operation converts 1001 1010 to 0000 0000 and sets the carry flag. The second DAA instruction does nothing. The result is 41 hex, the ASCII letter A.

We will now develop a macro to determine the CP/M version number.

A MACRO TO FIND THE CP/M VERSION NUMBER

The original CP/M was given the version number 1.3. Subsequent versions are labeled 1.4, 2.0, 2.1, 2.2, and so forth. Many CP/M programs will run on all versions. However, several powerful features were introduced with version 2, and any program that uses these new features cannot be executed on version 1. In fact, we will write a program in Chapter 8 that uses the built-in disk-parameter tables, and it will not run on version 1 for this reason. Programs that use the features of version 2 should determine the version number of the CP/M they are running on and terminate if it is less than 2.

The CP/M version number is obtained with BDOS function 12. For version 2 and above, the version number is multiplied by 10 and returned in both the accumulator and register L as a packed BCD number. For example, version 2.2 is represented by the number 22 hex. A value of 0 is returned for versions prior to 2.0.

Macro CPMVER can be used to determine whether version 2.0 or later is being used. The macro is shown in Figure 5.13. Add it to your macro library.

A PROGRAM TO DISPLAY THE IOBYTE VALUE

In Chapter 3 we learned how to map the four logical devices—console, reader, punch, and list—into 16 physical devices. Then we incorporated the IOBYTE feature into several BIOS routines. For example, by changing the IOBYTE to 1 we can send console output to the printer.

We learned that it is possible to change the IOBYTE with the debugger, with STAT, or with BASIC. However, it will be more convenient to dedicate an executable program to reading and changing the IOBYTE. We will develop the program in two parts.

We begin with a program to determine the current IOBYTE value and display it in hexadecimal. As an added feature, the program will also display the CP/M version number. Several of the macros we have written are required.

```
CPMVER    MACRO
;;(Put current date here)
;;Inline macro to determine the CP/M version.
;;Accumulator has version in BCD times 10.
;;A = 22 for version 2.2, A = 0 for ver 1.4.
;;
          PUSH    H
          PUSH    D
          PUSH    B
          MVI     C,12
          CALL    BDOS
          MOV     A,L         ;;not necessary
          POP     B
          POP     D
          POP     H
                              ;CPMVER
          ENDM
```

Figure 5.13: Macro CPMVER to Determine the CP/M Version Number

Make a copy of the program shown in Figure 5.14, giving it the name IOBYTE. Assemble it and execute it. The program will give the current hex value of the IOBYTE and the CP/M version number. The program obtains the CP/M version number with macro CPMVER. The version number is obtained as a packed BCD number. However, we use macro OUTHEX, our binary-to-hexadecimal converter, to display the results.

The original value is saved in the C register. Macro UPPER moves the upper character to the lower position and zeros the upper four bits. A logical OR with an ASCII zero converts the binary digit to ASCII so it can be printed by macro PCHAR. A decimal point is printed with PCHAR. Then the original byte is retrieved from the C register and the lower character is similarly converted to ASCII and printed.

Before completing our IOBYTE program we must add two more macros.

A Macro to Read the Console Buffer

Earlier in this chapter we considered two kinds of output routines. One type, using BDOS function 2, displays individual characters one at a time. An alternative approach, using BDOS function 9, prints an entire buffer

```
TITLE   'Show IOBYTE and Version'
;also show CP/M version
;
;(Put current date here)
;
;Usage:  IOBYTE
;
FALSE   EQU     0
TRUE    EQU     NOT FALSE
;
IOBYTE  EQU     3
BOOT    EQU     0               ;system reboot
BDOS    EQU     5               ;BDOS entry point
FCB1    EQU     5CH             ;input FCB
TPA     EQU     100H            ;transient program area
;
;Set flags in main program so only one
;copy of certain subroutines will be generated.
;Place set lines before MACLIB call.
;
COFLAG  SET     FALSE           ;console output
CXFLAG  SET     FALSE           ;binary to hex
HXFLAG  SET     FALSE           ;hex to binary in HL
PRFLAG  SET     FALSE           ;print console buffer
RCFLAG  SET     FALSE           ;read console buffer
;end of flags
;
        MACLIB  CPMMAC
;
ORG     TPA
;
START:
        ENTER
        VERSN   '(current date).IOBYTE1'
        PRINT   ' IOBYTE is '
        MVI     C,7             ;get IOBYTE
        CALL    BDOS
        OUTHEX
```

Figure 5.14: Program to Display the IOBYTE Value and the CP/M Version Number

```
        PRINT   ' hex'
DONE:
        PRINT   ' for CP/M version '
        CPMVER
        MOV     C,A             ;save
        UPPER                   ;move down
        ORI     '0'             ;convert to decimal
        PCHAR                   ;left digit
        PCHAR   PERIOD
        MOV     A,C
        ANI     0FH
        ORI     '0'             ;convert to decimal
        PCHAR                   ;right digit
        EXIT
;
        END     START
```

Figure 5.14 (continued)

of characters at one time. Similarly, we can input console characters one at a time by using BDOS function 1, or we can read an entire line of characters with BDOS function 10.

Macro READCH can be used to read single characters. We will now use BDOS function 10 to develop a macro to input an entire line of characters from the console.

When console characters are read with function 10, they are placed into a memory region known as the console buffer. This buffer area must be established prior to making the BDOS call. Two auxiliary bytes, located immediately in front of the buffer, are associated with the buffer. The first of these two bytes defines the buffer length, the maximum number of characters it can hold. The second byte gives the actual number of characters present in the buffer.

To use BDOS function 10, the DE register is loaded with the address of the first auxiliary byte, register C gets the value of 10, and BDOS is called. As each character is typed, CP/M places it in the buffer and also displays it on the console screen. The function terminates when a carriage return is entered or when the number of characters equals the maximum number specified by the first auxiliary byte. CP/M also sets the second auxiliary byte to the number of characters that were read. The following CP/M

control characters also can be used with this mode of data entry:

Character	*Meaning*
E	Begin new line
H	Backspace
I	Tab
P	Engage/disengage printer
R	Reprint line
U	Cancel line
X	Cancel line

We will now develop macro READB, shown in Figure 5.15, to read the console buffer. The instructions at the beginning of the macro implement the BDOS call to fill the console buffer. The buffer itself is located at the end of the macro; it is given the label RBUF. The auxiliary bytes are called RBUFM and RBUFC.

```
RBUFM:   DB   RBUFE-RBUF   ;maximum count
RBUFC:   DS   1            ;actual count
RBUF:    DS   16           ;buffer start
RBUFE:                     ;buffer end
```

The DE register is loaded with the address of RBUFM, the location of the maximum buffer length. Notice that the assembler calculates this length by subtracting the address of the buffer beginning (RBUF) from the address of the buffer end (RBUFE). When the buffer operation is completed and control returns to the calling program, the location RBUFC

```
READB       MACRO
;;(Put current date here)
;;Inline macro to input a line from console.
;;Buffer is located at end of macro.
;;Get characters from buffer by calling
;;global subroutine GETCH in this macro.
;;Buffer pointer RBUFP is also global.
;;
            LOCAL   AROUND,RBUFM,RBUF,RBUFC,RBUFE
            CALL    RDB2?
            IF      NOT RCFLAG
```

Figure 5.15: Macro READB to Read the Console Buffer

```
                JMP     AROUND
RDB2?:
                PUSH    H
                PUSH    D
                PUSH    B
                LXI     D,RBUFM
                MVI     C,10
                CALL    BDOS
                LXI     H,RBUFM+2
                SHLD    RBUFM-2
                POP     B
                POP     D
                POP     H
                RET
;global routine to get char from buffer
GETCH:
                LDA     RBUFC           ;get count
                SUI     1               ;decr with carry
                RC                      ;no more char
                STA     RBUFC
                PUSH    H
                LHLD    RBUFP
                MOV     A,M             ;get char
                INX     H               ;next
                SHLD    RBUFP
                POP     H
                RET
;
RCFLAG          SET     TRUE            ;only one copy
;
RBUFP:          DW      RBUF            ;buffer pointer
;console buffer address
RBUFM:          DB      RBUFE-RBUF      ;max length
RBUFC:          DS      1               ;actual length
RBUF:           DS      16              ;buffer start
RBUFE:                                  ;buffer end
                ENDIF
AROUND:                                 ;;READB
                ENDM
```

Figure 5.15 (continued)

contains the actual number of characters read during the input step.

After a line of characters has been placed in the console buffer with BDOS function 10, it is necessary to get the characters from the buffer. The middle portion of macro READB contains a separate global subroutine called GETCH for this purpose. Each time subroutine GETCH is called, it returns with the next character in the accumulator.

When subroutine GETCH takes a character from the buffer, it decrements the count of remaining characters stored at location RBUFC. To make this task easier, a buffer pointer, RBUFP, is used. This pointer is set to the first character when the buffer is initially filled. Each time GETCH removes a character, it increments the pointer.

The carry flag is reset each time GETCH returns a valid character. However, if there are no remaining characters when GETCH is called, the carry flag is set. Thus, it is important to check the carry flag immediately after a return from subroutine GETCH. The buffer pointer, RBUFP, is a global symbol. It can therefore be accessed by any other part of the program. Incorporate macro READB into your macro library.

A Macro to Convert Hexadecimal to Binary

Earlier in this chapter we considered a macro to convert a binary number to a hexadecimal number; we will now consider a complementary program to convert a hexadecimal number to a binary number. Macro HEXHL, shown in Figure 5.16, reads ASCII characters from the console buffer and converts them to a 16-bit binary number in the HL register. The characters must first be read, so macro READB must be referenced before macro HEXHL. This macro also requires macro UCASE.

Macro HEXHL operates on a series of valid ASCII-coded hex numbers. A blank character or the end of the buffer normally terminates the operation. If a nonhexadecimal character is encountered, the carry flag is set. Thus, you should check the state of the carry flag at the end of this step. Copy macro HEXHL into your macro library.

```
HEXHL       MACRO
;;(Put current date here)
;;Inline macro to convert ASCII hex characters
;;in buffer to a 16-bit binary number in HL.
```

Figure 5.16: Macro HEXHL to Convert a String of ASCII Hex Characters to a 16-Bit Binary Number

```
;;Character string is addressed by POINTR.
;;Carry flag set if invalid hex character found.
;;Macros needed: READB, UCASE
;;
            LOCAL   AROUND, RDHL2, NIB?
            CALL    RDHL?
;;
            IF      NOT HXFLAG      ;one copy only
            JMP     AROUND
RDHL?:
            LXI     H,0             ;start with 0
RDHL2:
;get character from console buffer
            CALL    GETCH
            CMC
            RNC                     ;end of line
            UCASE                   ;make uppercase
            CALL    NIB?            ;to binary
            RC                      ;error
            DAD     H               ;times 2
            DAD     H               ;times 4
            DAD     H               ;times 8
            DAD     H               ;times 16
            ORA     L               ;combine new
            MOV     L,A             ;put back
            JMP     RDHL2           ;next
;
;convert ASCII to binary
;
NIB?:       SUI     '0'             ;ASCII bias
            RC                      ; < 0
            CPI     'F'-'0'+1
            CMC
            RC                      ; > F
            CPI     10
            CMC
            RNC                     ;a number 0-9
            SUI     'A'-'9'-1
```

Figure 5.16 (continued)

```
        CPI     10
        RET
HXFLAG  SET     TRUE        ;only one copy
        ENDIF
AROUND:                     ;;HEXHL
        ENDM
```

Figure 5.16 (continued)

IOBYTE, Version 2

The program we wrote previously can be used to display the current value of the IOBYTE at address 3. We will now add a new feature to this program—the ability to alter the value of the IOBYTE.

Make a copy of the first IOBYTE program (Figure 5.14) and alter it to look like Figure 5.17. Assemble the new version and execute it. If the program is executed as before, without a parameter on the command line, the

```
TITLE   'IOBYTE: show or change'
;also show CP/M version
;
;(Put current date here)
;
;Usage:  IOBYTE
;        IOBYTE C0
;
;performs warm start to reset memory pointer
;
FALSE   EQU     0
TRUE    EQU     NOT FALSE
;
IOBYTE  EQU     3           ;memory location
BOOT    EQU     0           ;system reboot
BDOS    EQU     5           ;BDOS entry point
FCB1    EQU     5CH         ;input FCB
FCB2    EQU     6CH         ;2nd parameter
```

Figure 5.17: Program IOBYTE to Display and Change the IOBYTE

```
DBUFF   EQU     80H             ;default buffer
TPA     EQU     100H            ;transient program area
;
;Set flags in main program so only one
;copy of certain subroutines will be generated.
;Place set lines before MACLIB call.
;
COFLAG  SET     FALSE           ;console output
CXFLAG  SET     FALSE           ;binary to hex
HXFLAG  SET     FALSE           ;hex to binary in HL
PRFLAG  SET     FALSE           ;print console buffer
RCFLAG  SET     FALSE           ;read console buffer
;end of flags
;
        MACLIB  CPMMAC
;
ORG     TPA
;
START:
        ENTER
        VERSN   '(current date).IOBYTE '
        LXI     H,FCB1+1        ;parameter if any
        MOV     A,M             ;first byte
        CPI     BLANK           ;anything?
        JZ      NOPAR           ;no
;use FCB as buffer
        SHLD    RBUFP
        LDA     80H             ;buffer length + 1
        DCR     A               ;skip the blank
        STA     RBUFP+3         ;save the count
;
AGAIN:
        HEXHL                   ;hex to binary
        JC      BADPAR          ;input error
        MOV     E,L
        MVI     C,8             ;set IOBYTE
        CALL    BDOS
        JMP     DONE
```

Figure 5.17 (continued)

```
BADPAR:
        PRINT   'Enter the hex value: '
        READB                   ;try again
        JMP     AGAIN
NOPAR:
        PRINT   ' IOBYTE is '
        MVI     C,7             ;get IOBYTE
        CALL    BDOS
        OUTHEX
        PRINT   ' hex'
DONE:
        PRINT   ' for CP/M version '
        CPMVER
        MOV     C,A             ;save
        UPPER                   ;move down
        ORI     '0'             ;convert to binary
        PCHAR                   ;left digit
        PCHAR   PERIOD
        MOV     A,C
        ANI     0FH
        ORI     '0'             ;convert to decimal
        PCHAR                   ;right digit
        EXIT    BOOT            ;warm start
;
        END     START
```

Figure 5.17 (continued)

current value of the IOBYTE and the CP/M version will be displayed. Alternatively, if a valid hexadecimal character is given as a parameter, the IOBYTE is changed to the desired value. Finally, if an invalid hexadecimal number is entered, the value is requested again.

If your BIOS incorporates the IOBYTE feature, you can test the new version of this program. (Adding the IOBYTE feature to BIOS is described in Chapter 3.) Suppose, for example, that the current value of the IOBYTE is 0 and a value of 1 sends console output to the printer. Change the IOBYTE to 1 with the command

```
IOBYTE 1
```

Console output should now appear at the printer. To return to the

previous state, give the command

```
IOBYTE 0
```

The program begins with macros ENTER and VERSN. Then a check is made to see if a parameter was included on the command line. CP/M places the first parameter, if present, in the first file control block (FCB1) starting at 5C hex. If there is no disk-drive parameter, as in the present application, the byte at 5C hex is automatically zeroed. The parameter then begins at the next location, 5D hex. If no parameter was entered on the command line, there will be a blank at location 5D hex (FCB1 + 1). The program then prints the current value of the IOBYTE and the CP/M version.

If a parameter was entered on the command line, then the byte at FCB1 + 1 will not be blank. The next step is to convert the one or two ASCII characters to a binary number and store the result in the IOBYTE location at address 3. Macro HEXHL is used for this purpose. Remember, however, that this macro was written to obtain its characters from the console buffer. Therefore, we set the console buffer pointer (RBUFP) to the beginning of the file control block (FCB1 + 1).

We also need to set the number of characters in the buffer. This is obtained from the default console buffer at 80 hex. The first parameter begins at address 82 hex, and the number of characters that was entered appears at address 80 hex. This count is actually one character too large, because it includes the space in front of the parameter. Consequently, the following instructions get the character count, decrease it by one to account for the blank, and store the value in our console buffer at location RBUFC:

```
LDA    80H          ;buffer length + 1
DCR    A            ;skip the blank
STA    RBUFP+3      ;save the count
```

Macro HEXHL is now executed to convert the parameter to a binary number. If an invalid hexadecimal character is encountered, a new value is requested from the user. In this case, macro READB is called to get the desired value. We will now use macro HEXHL in another program to branch to an arbitrary memory location.

A PROGRAM TO GO TO ANY ADDRESS IN MEMORY

A program can be executed under CP/M by typing its name and any parameters. The CP/M system copies the executable image from disk into memory starting at the TPA (address 100 hex). CP/M then branches to

address 100 hex to start the program. Sometimes, however, we may need to go to an address other than 100 hex. For example, there may be a bootstrap loader or a monitor at a high memory location. Suppose that we have two different versions of BIOS, each one saved on a different system disk. We could change from one system to another simply by changing the diskette and branching to the bootstrap loader. We can execute the debugger DDT or SID in this case, using the debugger G command to force a branch to the desired address.

Alternatively, the program shown in Figure 5.18 can be used to branch

```
TITLE   'GO anywhere in memory'
;
;(Put current date here)
;
;Usage:  GO   (address)
;        GO
;        *(address)
;
;macro library for CP/M system calls
;
FALSE   EQU     0
TRUE    EQU     NOT FALSE
;
BOOT    EQU     0           ;system reboot
BDOS    EQU     5           ;BDOS entry point
FCB1    EQU     5CH         ;input FCB
FCB2    EQU     6CH         ;2nd parameter
DBUFF   EQU     80H         ;default buffer
TPA     EQU     100H        ;transient program area
;
;Set flags in main program so only one
;copy of certain subroutines will be generated.
;Place set lines before MACLIB call.
;
COFLAG  SET     FALSE       ;console output
HXFLAG  SET     FALSE       ;hex to binary in HL
PRFLAG  SET     FALSE       ;print console buffer
```

Figure 5.18: Program GO to Branch Anywhere in Memory

```
RCFLAG    SET       FALSE          ;read console buffer
;end of flags
;
          MACLIB    CPMMAC
;
ORG       TPA
;
START:
          VERSN     '(current date).GO '
          LXI       H,FCB1+1       ;parameter if any
          MOV       A,M            ;first byte
          CPI       BLANK          ;anything?
          JZ        NOPAR          ;no
;use FCB as buffer
          SHLD      RBUFP          ;save pointer
          LDA       80H            ;console buffer length + 1
          DCR       A              ;skip the blank
          STA       RBUFP+3        ;save the count
AGAIN:
          HEXHL                    ;hex to binary
          JC        NOPAR          ;input error
          PCHL                     ;go to address
;
;improper parameter, try again
;
NOPAR:
          PRINT     'Enter the hex address: '
          READB                    ;input console line
          JMP       AGAIN          ;try again
;
          END       START
```

Figure 5.18 (continued)

to any memory address. The desired hexadecimal address can be given on the command line, or it can be given after the program has started. For example, the command

```
GO E800
```

will cause a branch to the address E800 hex.

This program is very similar to IOBYTE version 2. Macros VERSN, HEXHL, READB, and PRINT are required. Notice that we did not save the incoming stack pointer in this program.

Create a file named GO. Type in the program, assemble it, and run it. If you have a monitor in memory, branch to it with the GO command. Even if you have nowhere else to go, you can test the program. Give the command of GO without a parameter. When the program requests an address, give a value of 0. This will cause CP/M to perform a warm start.

A PROGRAM TO EJECT PAGES ON THE PRINTER

The last program in this chapter will allow us to eject one or more pages on the printer. We will need a new macro, called LCHAR, for this program. Macro LCHAR performs the same task on the printer that macro PCHAR does on the console. In fact, it would be relatively easy to combine the two macros into one, but referencing the combination macro would then be more complicated. Consequently, we will keep the two separate.

Place a copy of macro LCHAR (Figure 5.19) in your macro library. The easiest way to do this may be to make a duplicate of macro PCHAR. Then change every occurrence of the three letters PCH to LCH on the copy. Also, change the first parameter in the reference to macro SYSF from the value of 2 to the value of 5.

Create a file named PAGE. Type in the program shown in Figure 5.20 and assemble it. The program begins with macros ENTER and VERSN. Then the file control block is tested to see if a parameter was entered on the command line. This time, however, we use the command

```
LDA     FCB1+1
```

for this purpose. If this location is blank, no parameter was entered and one page will be ejected. If a parameter was included in the command, it is used to determine how many pages are ejected. To avoid getting too many pages, only the lower three bits of the input value are used. This allows a maximum of seven pages to be ejected.

There are two loops in the main part of the program. The outer loop counts the number of pages. The inner loop counts the number of lines. Macro LCHAR is used to send line feeds to the printer. This program is very simple, but it demonstrates several important features. For example, in previous programs we checked the location 5D hex (FCB+1) to see whether a file name was given as a parameter on the command line. In this program, however, we expect the parameter to be a decimal number.

```
LCHAR       MACRO   PAR
;;(Put current date here)
;;Inline macro to send one char to list.
;;Optional PAR is loaded into A.
;;Macro needed: SYSF
;;
;;Usage:    LCHAR   '*'
;;          LCHAR   CR
;;          LCHAR
;;
            LOCAL   AROUND
            IF      NOT NUL PAR
            MVI     A,PAR
            ENDIF
            CALL    LCH2?
            IF      NOT LOFLAG
            JMP     AROUND
LCH2?:      SYSF    5, AE           ;list char
LOFLAG      SET     TRUE
            ENDIF
AROUND:                             ;;LCHAR
            ENDM
```

Figure 5.19: Macro LCHAR to Print Characters on the Printer

```
TITLE   'PAGE: eject pages on printer'
;
;(Put current date here)
;
;Usage:  PAGE
;        PAGE  3
;
FALSE   EQU     0
TRUE    EQU     NOT FALSE
```

Figure 5.20: Program PAGE to Eject Pages on the Printer

```
;
BDOS    EQU     5       ;BDOS entry point
TPA     EQU     100H    ;transient program area
FCB1    EQU     5CH     ;parameter
LPAG    EQU     66      ;lines per page
;
;Set flags in main program so only one
;copy of certain subroutines will be generated.
;Place set lines before MACLIB call.
;
LOFLAG  SET     FALSE   ;list output
;end of flags
;
        MACLIB  CPMMAC
;
ORG     TPA
;
START:
        ENTER
        VERSN   '(current date).PAGE '
        MVI     C,1     ;set for one page
        LDA     FCB1+1  ;parameter?
        CPI     BLANK
        JZ      NPAGE   ;no
        ANI     3       ;maximum number
        MOV     C,A
NPAGE:
        MVI     B,LPAG
LINES:
        LCHAR   LF
        DCR     B
        JNZ     LINES
        DCR     C       ;more pages?
        JNZ     NPAGE   ;yes
DONE:
        EXIT
;
        END     START
```

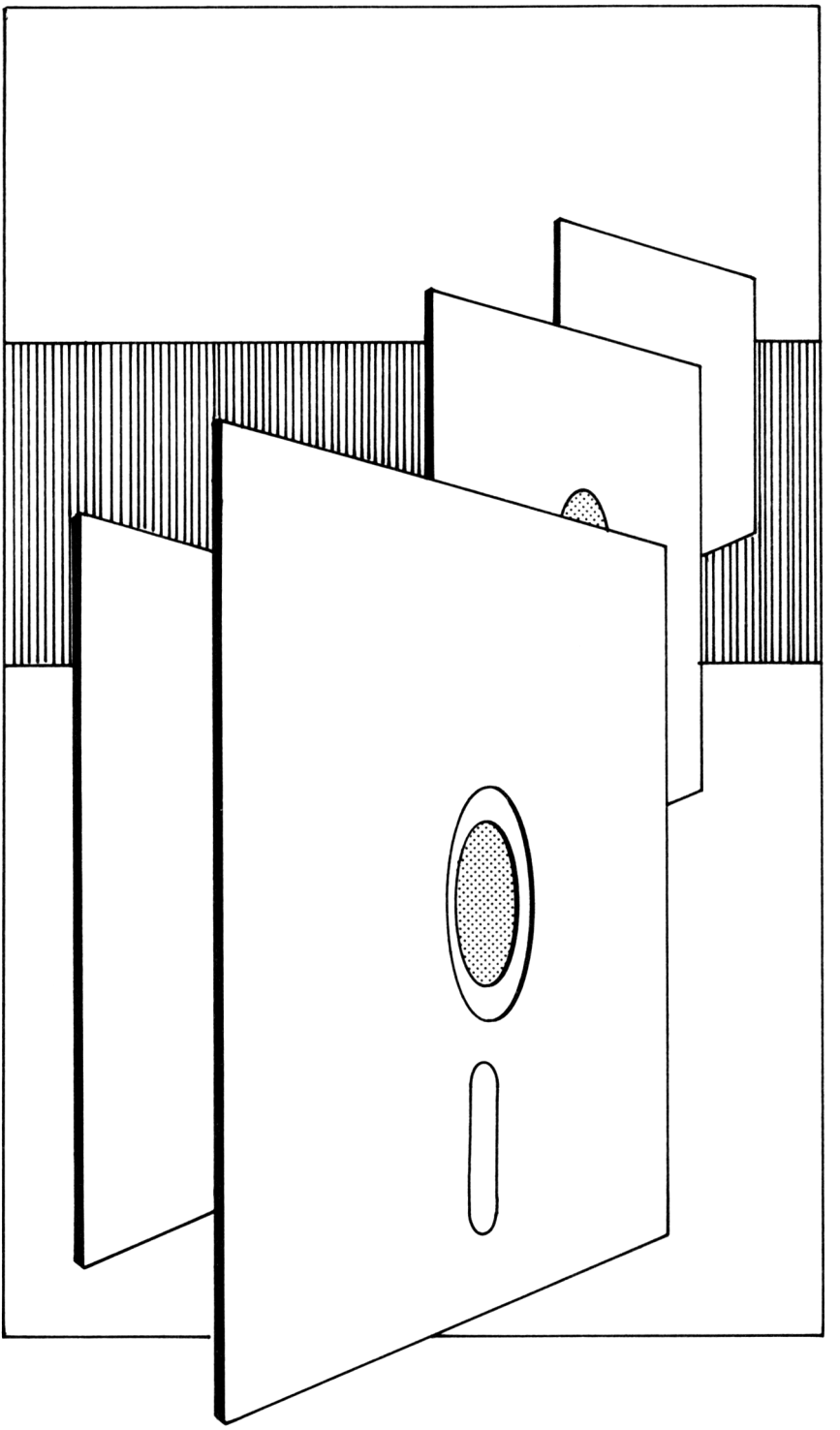
Figure 5.20 (continued)

SUMMARY

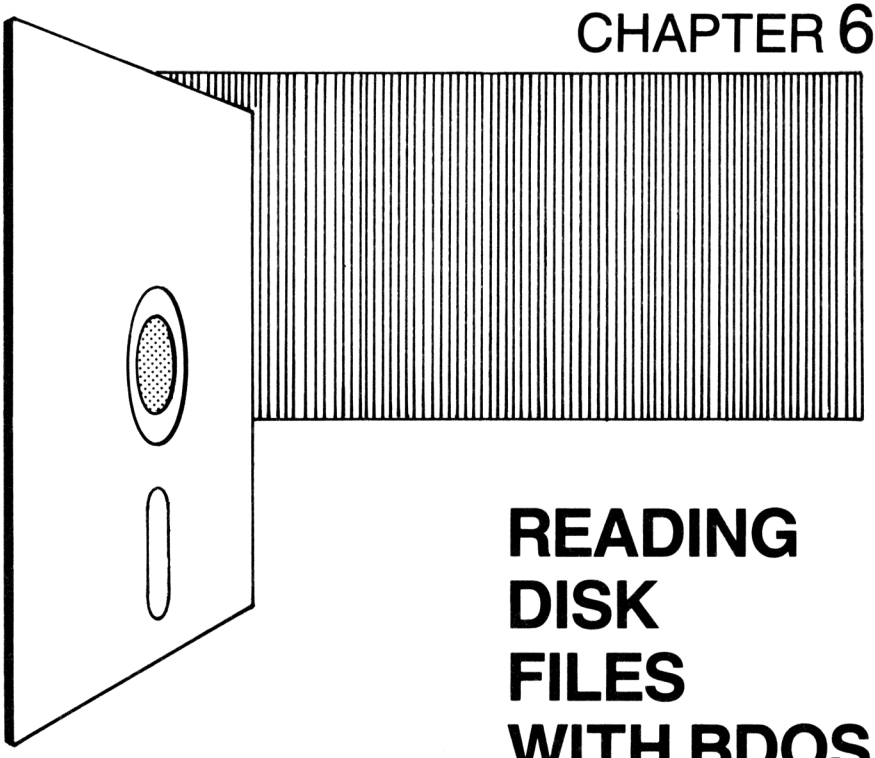
We began our macro library in Chapter 4 with several general-purpose routines. In this chapter we have added macros that interact with the peripherals through the CP/M BDOS. These include macros to write characters on the console, read characters from the console, read and write the console buffer, and make base conversions. We wrote four executable programs, primarily to learn more about how CP/M is organized. Of course, these programs are useful in their own right.

The directory of your macro library should now look like this:

```
;;Macros in this library                          Flags
;;AMBIG      MACRO   OLD, NEW                    (none)
;;COMPAR     MACRO   FIRST, SECOND, BYTES        CMFLAG
;;COMPRA     MACRO   FIRST, SECOND, BYTES        CMFLAG
;;CPMVER     MACRO                               (none)
;;CRLF       MACRO                               CRFLAG, COFLAG
;;ENTER      MACRO                               (none)
;;EXIT       MACRO   SPACE?                      (none)
;;FILL       MACRO   ADDR, BYTES, CHAR           FLFLAG
;;HEXHL      MACRO   POINTR                      HXFLAG, RCFLAG
;;LCHAR      MACRO   PAR                         LOFLAG
;;MOVE       MACRO   FROM, TO, BYTES             MVFLAG
;;OUTHEX     MACRO   REG                         CXFLAG, COFLAG
;;PCHAR      MACRO   PAR                         COFLAG
;;PRINT      MACRO   TEXT, BYTES                 PRFLAG, COFLAG
;;READB      MACRO   BUFFR                       RCFLAG
;;READCH     MACRO   REG                         CIFLAG, COFLAG
;;SBC        MACRO                               (none)
;;SYSF       MACRO   FUNC, AE                    (none)
;;UCASE      MACRO   REG                         (none)
;;UPPER      MACRO   REG                         (none)
;;VERSN      MACRO    NUM                        (none)
```



CHAPTER 6



READING DISK FILES WITH BDOS

INTRODUCTION

In Chapter 5 we developed macros and programs for performing CP/M operations using BDOS calls. However, we did not consider disk operations. In this chapter we will expand our capabilities to include reading disk files. We will learn how to write disk files in Chapter 7. We begin by summarizing the organization of the disk and the way CP/M stores information on it. Then we will develop several important macros for disk operations. To demonstrate the use of these macros, we will write several executable programs: SHOW to display ASCII files on the console, DUMP to display a COM disk file in ASCII and hexadecimal, ADDRESS to address an envelope from an existing letter file, and PAIR to count pairs of control characters.

THE FILE CONTROL BLOCK

The disk surface is partitioned into concentric tracks, which are further subdivided into sectors. The disk hardware is able to address these sectors

individually. However, CP/M accesses a larger unit called a *block*. A single-density disk has a block size of 1024 (1K) or 2048 (2K) bytes. Double-density disks have a block size of 2K, 4K, 8K, or 16K bytes. There are 128 bytes of information on each sector. Therefore, a 1K block contains 8 sectors and a 2K block contains 16 sectors.

Each file on a CP/M disk is described by a 32-byte file control block (FCB) that is written into the disk directory. The first 16 bytes of the FCB give the name of the file, its length, and other characteristics. The remaining 16 bytes of the FCB specify the disk location of each block contained in the file.

Before a disk file can be accessed, a second copy of the FCB must be created in memory. As the disk file is altered, the memory version of the FCB changes. At the end of a write operation, the disk version of the FCB will be updated from the memory version. We must be able to distinguish the two versions of the FCB, because they are sometimes different. Unfortunately, there is no standard terminology for this distinction. However, in this book we will use the expressions *memory* FCB and *disk* FCB when this distinction is necessary. We will use the unqualified expression FCB when both versions are the same.

Before we look at the details of the FCB, let us review binary coding. Data are written onto the disk as a sequence of binary digits (0 or 1), just as in memory. As we saw in Chapter 5, information is represented sometimes in regular binary form and other times in ASCII. For example, the bit patterns for a binary five and an ASCII five are as follows:

Binary Five	*ASCII Five*
0000 0101	0011 0101

Some of the bytes of the FCB are coded in regular binary; other bytes are in ASCII. We generally express regular binary numbers in hexadecimal form. Thus a binary three would be shown as 03. We could also express the ASCII characters in hexadecimal form, but it is more useful to show them in ASCII. Therefore, an ASCII three is shown as 3 rather than its hexadecimal value of 33.

We now turn to the details of the FCB. The first byte of the disk FCB contains the user number. This is a binary number from 0 – F hex. By contrast, the first byte of the memory FCB specifies the disk drive. This is a binary number from 0 – 10 hex. Drives A, B, and C correspond to values of 1, 2, and 3. The maximum allowable value is 10 hex, corresponding to drive P. A value of 0 in this first position refers to the default or currently logged-in drive. The next eight bytes of the FCB (bytes 1 – 8) contain the file name in ASCII. This field is filled out with blanks if necessary. The

file name extension is stored in the next three bytes (bytes 9–11). This is an optional field that is used to describe the nature of the file. We have seen that the extension BAS is used for BASIC files, FOR for FORTRAN files, BAK for backup files, and so forth. This field is also filled with blanks if necessary.

Large files require more than one FCB for the complete specification of all the blocks. In this case there will be more than one FCB with the same file name. The next byte (12) distinguishes FCBs with the same file name. The number in this position is called the *extent.* It will be zero for small files. The next two bytes need not concern us. The last byte in this half of the FCB (15) gives the number of records (128-byte sectors) in the FCB. The remaining 16 bytes of the FCB give the location of each block of sectors on the disk.

Five sample disk FCBs are shown in Figure 6.1. Remember, the information is actually present as a sequence of bits. However, in this figure the file names are shown in ASCII, while the other information is given in hexadecimal notation. Three columns of space have also been added for clarity.

```
00 CPMIO   ASM 00000055 02030405060708090A0B0C0000000000
00 CPMIO   HEX 0000000C 0D000000000000000000000000000000
00 SORT    COM 00000080 121314151617181 91A1B1C1D1E1F2021
00 SORT    COM 01000080 22232425262728292A2B2C2D2E2F3031
00 SORT    COM 0200000A 32330000000000000000000000000000
```

Figure 6.1: Five File Control Blocks for Three Files

The initial byte of each entry is zero, indicating that all of these files were saved by user zero. The first file, CPMIO.ASM, contains 55 (85 decimal) records and is found on blocks 02–0C. The next file, CPMIO.HEX, contains 0C records and is located entirely in block 0D.

The third, fourth, and fifth entries in the figure are named SORT.COM; they all refer to the same file. Each of these entries has the same file name but a different extent number. The file is so large that one FCB is not sufficient to describe it. The first 80 records (blocks 12–21) are referenced by the first extent (0). Blocks 22–31 are referenced by the second extent (1). The remaining 0A records (blocks 32 and 33) are referenced by the third extent (2).

Later in this chapter we will write macros to activate and read disk files. However, we must consider the possibility of a misspelled file name. We will therefore add a macro to handle error messages.

A MACRO TO DISPLAY AN ERROR MESSAGE AND ABORT THE PROGRAM

Each time a program requests data from the console, a check should be made to see that the information is meaningful. An error message should be given if it is not. For example, suppose that an alphabetic character is given when a decimal number is needed. The operator should be informed of the problem.

There is a second matter we must consider. The statements of a computer program are normally executed in order. However, when an error is discovered, we will want to execute an alternate set of instructions and perhaps terminate the program. Let us combine these two ideas—displaying an error message and branching to an alternate location—into a macro called ERRORM.

We previously wrote macro PRINT to send messages to the console. We will reference macro PRINT to display the error message. (We have seen that one macro can reference another.) Then we will branch to our alternate location. Macro ERRORM is given in Figure 6.2. Add it to your macro library. Macros PRINT and CRLF are referenced within it.

This macro has two parameters. The first parameter is the text of the error message. The second parameter is the branch address after the error message is printed. If this parameter is omitted from the macro reference, a warm start is performed by a branch to the address of BOOT.

Notice that the parameter in the reference to macro PRINT is enclosed in angle brackets. This construction is necessary if the first parameter to ERRORM is enclosed in angle brackets. The macro assembler removes one set of angle brackets each time a macro is expanded. Thus, one pair of brackets is removed when macro ERRORM is expanded and a second pair of brackets is removed when macro PRINT is expanded. The angle brackets are necessary because commas are sometimes used in the text of the parameter as well as in separating one parameter from another. The assembler interprets commas as parameter separators unless they are within angle brackets. For example, the expression

```
ERRORM    <CR,LF,'?File exists'>
```

contains only one parameter. However, the expression

```
ERRORM    CR,LF,'?File exists'
```

contains three parameters.

Now that we have reviewed the fundamentals of CP/M file organization and written a macro to display error messages, we can learn how to access an existing disk file.

```
ERRORM    MACRO    TEXT, WHERE
;;(Put current date here)
;;Macro to print message on console.
;;Message is enclosed in apostrophes.
;;Optional second parameter has branch address.
;;If no second parameter, go to BOOT.
;;Macros needed: PRINT, CRLF
;;
;;Usage:    ERRORM    'Message'
;;
          CRLF
          PRINT     <TEXT>
          IF        NUL WHERE
          JMP       BOOT          ;quit
          ELSE
          JMP       WHERE
          ENDIF                   ;;ERRORM
          ENDM
```

Figure 6.2: Macro ERRORM to Display an Error Message and Abort the Program

OPENING AN EXISTING DISK FILE

An existing disk file must be opened with BDOS function 15 before it can be referenced. A memory FCB must be allocated and partially filled out prior to the function call. The open operation fills out the remainder of the memory FCB from the disk FCB. The necessary information is as follows:

Byte	*Data*
0	Disk drive number
1–8	File name
9–11	File name extension
12	Extent (set to zero)
32	Record number (set to zero)

We saw earlier in this chapter that the memory FCB contains the drive number at position 0. The value is set to 0 for the default drive, 1 for drive A, 2 for drive B, and so on. The file name and extension are placed in the

next 11 bytes; they have the usual ASCII form. The extent byte at position 12 is set to zero. If the usual sequential access is desired, the record number byte at position 32 must be zeroed as well.

It will usually be necessary to provide all of the above information each time we open an existing disk file. Consequently, we will want to write a macro to make this task easier. But before we do this, let us see how CP/M can help us construct the memory FCB.

Constructing a Memory FCB with CP/M

When a program is executed from the command level of CP/M, there may be one or more parameters. The parameters given on the command line are known collectively as the tail. They are automatically placed in the console buffer starting at address 82 hex. CP/M also begins a memory FCB for the first parameter, including the disk drive, file name, and file type (bytes 0–11). The FCB is located at address 5C hex. If a second parameter is given on the command line, CP/M also begins a second FCB at address 6C hex.

We can use DDT or SID to see how CP/M sets up the memory FCB. We will execute the debugger with a single parameter. Then we can display the appropriate regions of memory to see what CP/M has done. When the debugger is executed with a parameter, it will attempt to access the requested file. However, if the debugger finds the requested file, it will be loaded into memory and the FCB will be deleted. Therefore, you must use a *nonexistent* disk file for this example.

Suppose that DDT is located on drive A. Go to drive B and give the command

```
A:DDT FIRST.EXT
```

or

```
A:SID FIRST.EXT
```

(Remember that there must be no file named FIRST.EXT on drive B.) The command may be entered in either uppercase or lowercase letters. This command instructs CP/M to load DDT into memory and execute it. CP/M also begins a memory FCB for the file named FIRST.EXT, starting at address 5C hex. When DDT gets control, it will attempt to copy FIRST.EXT into memory. A question mark will appear because the file does not exist.

At this point, CP/M has started an FCB at 5C hex and placed the command line tail in the console buffer at 82 hex. Examine this region of

memory with the command

```
D50,8F
```

The following display should appear:

```
0050 00 00 00 00 00 00 00 00 00 00 00 00 00 46 49 52 .............FIR
0060 53 54 20 20 20 45 58 54 00 00 00 26 00 20 20 20 ST   EXT...&.
0070 20 20 20 20 20 20 20 20 00 00 00 00 00 00 00 00          ........
0080 0A 20 46 49 52 53 54 2E 45 58 54 00 00 00 00 00 . FIRST.EXT.....
```

Remember, the debugger display has three parts. The first number on each line is the address in hexadecimal. The next 16 bytes are the contents of the corresponding memory expressed in hexadecimal. The ASCII representation of these same bytes is then given if it is printable. A dot is shown if a character is not printable.

Look at the last line in the display. From the ASCII representation, we can see that the command tail, FIRST.EXT, has been placed in the console buffer at 82 hex. The length of the command tail (0A hex in this case) is placed at location 80 hex. There will always be a blank (20 hex) at location 81 hex. If you type the command line in lowercase letters, CP/M will convert the characters to uppercase. The result will be the same.

Now consider the FCB at 5C hex. The first byte designates the disk drive. It has a value of 0 in this example, indicating that the default drive has been selected. The file name, FIRST, appears next. Because it contains less than eight characters, the remainder of the field is filled out with blanks. The decimal point separating the file name from the file type is apparent in the console buffer, but it does not appear in the FCB. The file type is in its proper place starting at position 9 of the FCB. If less than three characters are given for the file type, the field is filled with blanks.

Let us try a slight variation of the previous example. Return to CP/M with control-C and give the command

```
A:DDT B:FIRST.EXT
```

or

```
A:SID B:FIRST.EXT
```

This command is functionally equivalent to the previous one, except that drive B is specifically included. Examine the memory region from 50 to 8F hex again with the command

```
D50,8F
```

In the resulting display, the command tail starting at address 82 hex shows

that drive B was specifically requested:

```
0050 00 00 00 00 00 00 00 00 00 00 00 00 02 46 49 52 .............FIR
0060 53 54 20 20 20 45 58 54 00 00 00 26 00 20 20 20 ST   EXT...&.
0070 20 20 20 20 20 20 20 20 00 00 00 00 00 00 00 00          ........
0080 0C 20 42 3A 46 49 52 53 54 2E 45 58 54 00 00 06 . B:FIRST.EXT...
```

In the previous example the drive was omitted, so the memory FCB began with the value of 0. In this example, specifying drive B causes the first byte of the memory FCB at address 5C hex to have a value of 2.

A Macro to Open a Disk File

The previous examples demonstrate that CP/M can construct a memory FCB if the file name is given as a parameter on the command line. In order to access the file, we must still zero the extent byte and the record number byte, and then we must open the file with BDOS function 15.

After the return from BDOS, the accumulator contains the value of FF hex if the requested file could not be found. We must be ready to either continue with the program or display an error message and terminate the program.

We will now write a macro to construct a memory FCB and call BDOS function 15. Add macro OPEN (shown in Figure 6.3) to your library and place the name in the directory at the beginning.

Let us look at the details of macro OPEN. The first opcode loads DE with the FCB address:

```
LXI    D,POINTR
```

The symbol POINTR is a dummy parameter. The corresponding parameter in the macro reference is required. The next three instructions store a 0 at positions 12 and 32 of the memory FCB. The global subroutine OPEN2? is then called to perform BDOS function 15. After returning from BDOS, the accumulator contains FF hex if the file could not be opened. The next instruction increments the accumulator. This will reset the zero flag if the file was opened. The program then branches to the local label AROUND and continues. However, if the file could not be found, the remaining code is executed.

The macro reference to OPEN will normally omit the second parameter corresponding to WHERE, because we want to ensure that the file name in question actually exists. In this case, the expression IF NUL WHERE will be true and macro ERRORM will be referenced. It will generate the error message

```
?No source file
```

```
OPEN        MACRO     POINTR, WHERE
;;(Put current date here)
;;Inline macro to open an existing disk file.
;;POINTR refers to file control block.
;;Extent and current record number are zeroed.
;;Branch to location WHERE if file not found or
;;print error message and branch to DONE otherwise.
;;Macros needed: SYSF, ERRORM
;;
            LOCAL     AROUND
            LXI       D,POINTR
            XRA       A              ;zero
            STA       POINTR+12      ;extent
            STA       POINTR+32      ;current record
            CALL      OPEN2?
            INR       A              ;0=ok, FF means error
            JNZ       AROUND
            IF        NUL WHERE
            ERRORM    'No source file', DONE
            ELSE
            JMP       WHERE
            ENDIF
            IF        NOT OPFLAG
OPEN2?:     SYSF      15             ;open disk file
OPFLAG      SET       TRUE           ;only one copy
            ENDIF
AROUND:                              ;;OPEN
            ENDM
```

Figure 6.3: Macro OPEN to Open a Disk File

and then terminate the program with a branch to the global label DONE.

Sometimes, however, we will want to ensure that a particular file does *not* exist. Then we include a second parameter in the reference. For example, consider the following macro references:

```
            OPEN      FCB1, CONT2
            ERRORM    '?File name exists'
CONT2:
```

In this case, FCB1 refers to the memory FCB. However, the expression IF

NUL WHERE is false, so macro ERRORM is omitted from the expansion of macro OPEN. The program branches to the label CONT2 if the file is not found. If the file *is* located, an error message is printed and the program is terminated.

We need to create three additional macros to facilitate disk operation before we can write the next program. One will set the location of the memory buffer for reading a disk file, the second will actually read the file, and the third will request a file name and then create a memory FCB. We begin with macro SETDMA.

A MACRO TO SET THE DMA ADDRESS

BDOS function 20 is used to read a sector (128 bytes) from disk to memory. We saw previously that a 1K or larger block of sectors is the smallest amount of information that CP/M can read from a disk. However, when a sector is requested, CP/M finds the block in which it is located and copies the desired sector to a 128-byte sector buffer. The memory location of the sector buffer is called the DMA (disk memory access) address.

Each time a warm start occurs, the DMA address is automatically reset to 80 hex. However, this will not always be a convenient location. We saw earlier in this chapter that CP/M places the console buffer at 80 hex, and the debugger initially places its stack in this region as well. Furthermore, we will sometimes want to read an entire disk file into memory starting at 100 hex. In that case we will want the DMA address to be 100 hex for the first sector, 180 hex for the second sector, 200 hex for the third sector, and so forth. Therefore, we must be able to alter the DMA address. We may also want to reset the DMA address to 80 hex, in case the previous program set it somewhere else.

Macro SETDMA, given in Figure 6.4, uses BDOS function 26 to set the DMA address. The macro reference will usually give the DMA address as a parameter, in which case the assembler loads the DE register with the parameter. However, sometimes it will be more convenient to load the DE register from a memory location prior to the macro reference, in which case the parameter will be omitted. Copy the macro into your library.

Let us now construct a macro to read a disk sector.

A MACRO TO READ ONE DISK SECTOR

Before a disk file can be read, it is necessary to construct a memory FCB containing the file name and open the file with BDOS function 15. It may

```
SETDMA      MACRO   POINTR
;;(Put current date here)
;;Inline macro to set the DMA address where
;;next sector will be read or written.
;;Macro needed: SYSF
;;
            LOCAL   AROUND
            IF      NOT NUL POINTR
            LXI     D,POINTR
            ENDIF
            CALL    DMA2?
            IF      NOT DMFLAG
            JMP     AROUND
DMA2?:
            SYSF    26              ;set DMA address
DMFLAG      SET     TRUE            ;only one copy
            ENDIF
AROUND:                             ;;SETDMA
            ENDM
```

Figure 6.4: Macro SETDMA to Set the DMA Address

also be necessary to set the DMA address with BDOS function 26. At this point, a 128-byte sector can be read from the disk using BDOS function 20. The information is placed into memory starting at the current DMA address.

We will use macro READS, shown in Figure 6.5, whenever we need to read a disk sector. Copy the macro into your library. There are two parameters for this macro. The first parameter is the address of the memory FCB. The assembler loads this address into the DE register if the parameter is present. In this book the first parameter will usually be given the symbol FCB1. If the parameter is omitted from the macro, it is assumed that DE has been previously loaded.

The second parameter, if present, is printed after each sector is read. This will allow us to watch the action during the loading of a large file, but it also greatly slows the process.

Our next macro will request a file name and set up a memory FCB after a program has begun operation.

```
READS       MACRO   POINTR, STAR
;;(Put current date here)
;;Inline macro to read a disk sector.
;;POINTR refers to file control block.
;;Optional second parameter is symbol
;;to be printed after sector is read.
;;Zero flag is reset if end of file.
;;Macros needed: SYSF, PCHAR
;;
;; Usage:    READS   FCB1
;;           READS   FCBS, '*'
;;
            LOCAL   AROUND
            IF      NOT NUL STAR
            PCHAR   STAR            ;to console
            ENDIF
            IF      NOT NUL POINTR
            LXI     D,POINTR
            ENDIF
            CALL    READ2?
            ORA     A               ;set flags
            IF      NOT RDFLAG
            JMP     AROUND
READ2?:     SYSF    20              ;read disk sector
RDFLAG      SET     TRUE            ;only one copy
            ENDIF
AROUND:                             ;;READS
            ENDM
```

Figure 6.5: Macro READS to Read a Disk File into Memory

A MACRO TO INPUT A FILE NAME

We saw at the beginning of this chapter that a file name entered as a parameter on the command line is placed in the default console buffer at 82 hex. The parameter is also converted into a memory FCB at address 5C hex. The first byte of this FCB refers to the requested drive. For example, a value of 0 at this location refers to the default drive, drive A is referenced by 1, drive B is 2, and so on. CP/M also raises any lowercase letters to

uppercase, fills out the file name and file type with blanks, and removes the decimal point in the file name.

However, once a program has begun execution, CP/M cannot convert a file name into an FCB. Many of the programs we will write in this book expect a file name to be entered. If the parameter is given on the command line, CP/M creates the necessary memory FCB. However, if the file name was not given on the command line, the program must request one. The program itself must now process the characters that are entered. That is, byte 0 of the FCB must be set to 0 if no drive was specified, or set to 1 if drive A was specified. Lowercase letters must be converted to uppercase, and so forth.

The macro GFNAME, shown in Figure 6.6, asks for a file name and then sets up a memory FCB. The instructions it generates will only be used after a program has begun execution. Copy the macro into your library. The memory FCB is referenced through the parameter. This will usually be 5C hex, but we will sometimes use another address. The file name may be entered in either uppercase or lowercase letters. A disk drive also may be specified if desired.

We will now use the macros we have created to write a program to display ASCII files on the console.

```
GFNAME  MACRO   FCB
;;(Put current date here)
;;Inline macro to get file name from console
;;and place in FCB. Lowercase raised to uppercase.
;;Macros needed: READB, FILL, UCASE, PRINT, CRLF
;;Subroutine GETCH is part of macro READB.
;;
        LOCAL   AROUND,PNAME,ENAME,EXTEN,GNAM2
        PUSH    H
        PUSH    D
        PUSH    B
        LXI     H,FCB
        SHLD    FCBS?
        CALL    GNAM?
        POP     B
```

Figure 6.6: Macro GFNAME to Input a File Name after a Program Has Begun Executing

```
        POP     D
        POP     H
        IF      NOT FNFLAG
        JMP     AROUND
FCBS?:  DS      2               ;save orig pointer
GNAM?:
        CRLF
GNAM2:
        PRINT   <'                                      ',CR>
        PRINT   'Enter file name: '
        LHLD    FCBS?
        XRA     A               ;zero
        MOV     M,A             ;default drive
        INX     H
        FILL    , 11, BLANK
        XCHG
        READB                   ;console buffer
        CALL    GETCH           ;first char
        JC      GNAM2           ;try again
        CPI     BLANK
        JZ      GNAM2           ;try again
        UCASE
        STAX    D               ;maybe first
        CALL    GETCH           ;second char
        RC                      ;short name
        CPI     BLANK
        RZ                      ;ditto
        MVI     B,7             ;name length-1
        UCASE
        CPI     PERIOD
        JZ      ENAME
        CPI     ':'             ;drive?
        JNZ     PNAME           ;no
        LDAX    D               ;get drive
        SUI     'A'-1           ;make binary
        STAX    D               ;put it
        CALL    GETCH           ;start file name
        JC      GNAM2           ;drive only
```

Figure 6.6 (continued)

```
        UCASE
        INR     B
        DCX     D
PNAME:                          ;primary name
        INX     D
        STAX    D
        CALL    GETCH
        RC
        CPI     BLANK
        RZ
        UCASE
        CPI     PERIOD
        JZ      ENAME
        DCR     B
        JNZ     PNAME           ;ok
        JMP     GNAM2           ;if 9 char
ENAME:
        LHLD    FCBS?           ;get FCB
        LXI     D,9             ;ext offset
        DAD     D
        XCHG
        MVI     B,3
EXTEN:                          ;file name extension
        CALL    GETCH
        RC
        CPI     BLANK
        RZ
        UCASE
        STAX    D
        INX     D
        DCR     B
        JNZ     EXTEN
        RET                     ;done
;
FNFLAG  SET     TRUE
        ENDIF
AROUND:                         ;;GFNAME
        ENDM
```

Figure 6.6 (continued)

DISPLAYING AN ASCII FILE ON THE CONSOLE

We have seen that information is stored on disk and in memory as a sequence of bits. However, there are several different coding schemes. Source files are coded entirely in ASCII. Executable files are primarily binary with messages in ASCII. The distinction is important if we want to look at a file. An ASCII file can be sent directly to the console or printer, because these are ASCII devices. However, if we transmit a binary file to the console, it will be largely unintelligible.

An ASCII file can be viewed on the console screen by giving the CP/M command TYPE followed by the file name. But there are several disadvantages to this command. First, the file may scroll so quickly that the desired location is missed. Control-S can be pressed to freeze the screen, and any key can be pressed to resume scrolling. Control-S can be pressed again to freeze the screen. If any key other than control-S is pressed during scrolling, the command is terminated and we must start over.

Another disadvantage is that TYPE is a built-in CCP command. It cannot be given from the no-file level of a word processor such as WordStar. Program SHOW, given in Figure 6.7, solves both of these problems.

SHOW displays an ASCII file on the console one screenful at a time. Each time the space bar is pressed, the next screen is displayed. Pressing the carriage return key will display the next line. The program is terminated by pressing any other key.

SHOW is an executable program. Consequently, it can be run from the no-file level of WordStar. For example, to display the source program for SHOW, give the command

```
SHOW SHOW.ASM
```

Disk drive names can be used as needed. If the executable file is on drive A and the source file is on drive B, you can give the command

```
A:SHOW B:SHOW.ASM
```

SHOW can also be executed without a parameter. The program will then request the file name. If an error is made in entering the file name (too many characters or no characters), the request will be repeated. If the requested file name does not exist or if an attempt is made to display a COM file, the appropriate error message is printed and the program is terminated.

SHOW is designed for the usual video screen of 24 lines. If your video screen has a different number of lines, change the definition of the symbol LMAX from 24 to the proper number.

Type in the program given in Figure 6.7, assemble it, and execute it by displaying the source program of SHOW. If only the command SHOW is

```
TITLE   'SHOW ASCII file on console'
;
;(Put current date here)
;
;Usage: SHOW DISKFILE.EXT
;Press space bar to display next screen.
;Carriage return to scroll up one line
;performs same function as TYPE, but
;SHOW can be executed from WordStar.
;
FALSE   EQU     0
TRUE    EQU     NOT FALSE
;
BOOT    EQU     0               ;system reboot
BDOS    EQU     5               ;BDOS entry point
FCB1    EQU     5CH             ;input FCB
DBUFF   EQU     80H             ;default buffer
TPA     EQU     100H            ;transient program area
LMAX    EQU     24              ;lines per screen
;
;Set flags in main program so only one
;copy of certain subroutines will be generated.
;Place set lines before MACLIB call.
;
CIFLAG  SET     FALSE           ;input console char
CMFLAG  SET     FALSE           ;ASCII compare
COFLAG  SET     FALSE           ;output console char
CRFLAG  SET     FALSE           ;carr-ret/line-feed
DMFLAG  SET     FALSE           ;set DMA
FLFLAG  SET     FALSE           ;fill characters
FNFLAG  SET     FALSE           ;read file name
OPFLAG  SET     FALSE           ;open disk file
PRFLAG  SET     FALSE           ;print console buffer
RCFLAG  SET     FALSE           ;read console buffer
RDFLAG  SET     FALSE           ;read disk file
;end of flags
;
        MACLIB  CPMMAC
```

Figure 6.7: Program SHOW to Display an ASCII File on the Console

```
ORG     TPA
;
START:
        ENTER
        VERSN   '(current date).SHOW '
        LDA     FCB1+1
        CPI     BLANK           ;file name?
        JNZ     OPEND           ;yes
        GFNAME  FCB1            ;get file name
OPEND:
        COMPRA  'COM', FCB1+9   ;COM file?
        JZ      NOCOM           ;yes
        OPEN    FCB1            ;source file
        SETDMA  DBUFF           ;use default
        LXI     H,100H          ;set pointer
NEXTSC:
        CALL    SCREEN          ;next screen
FREE2:
        READCH                  ;wait for input
        CPI     BLANK           ;space?
        JZ      NEXTSC          ;next screen
        CPI     CR
        JNZ     DONE            ;abort
        PCHAR   CR
        MVI     B,1             ;set one line
        CALL    LINE            ;one line
        JMP     FREE2
;
;routine to fill console screen
;
SCREEN:
        MVI     B,LMAX          ;line count
        PCHAR   CR
NEXTLN:
        CALL    LINE
        DCR     B               ;count
        JNZ     NEXTLN          ;keep going
        RET
```

Figure 6.7 (continued)

```
;
;routine to display one line
;
LINE:
        MOV     A,H             ;check pointer
        ORA     A               ;still 80-FF?
        JZ      LIN3            ;yes
        READS   FCB1            ;read a sector
        JNZ     EOFILE          ;end of file
        LXI     H,DBUFF         ;reset pointer
        JMP     LINE
LIN3:
        MOV     A,M
        INX     H
        ANI     7FH             ;mask parity
        CPI     EOF             ;file end
        JZ      EOFILE          ;yes
        MOV     D,A             ;save
        CPI     CR              ;line end?
        JNZ     LIN2            ;no
        MOV     A,B             ;check position
        CPI     1               ;last line?
        RZ                      ;yes, skip CR
        MOV     A,D             ;retrieve CR
LIN2:
        PCHAR                   ;send to console
        MOV     A,D             ;restore
        CPI     CR              ;line end?
        JNZ     LINE            ;no
        RET
NOCOM:
        ERRORM  'Use DUMP for a COM file',DONE
EOFILE:
        READCH                  ;last page
DONE:
        EXIT
;
        END     START
```

Figure 6.7 (continued)

given, the program will request the file name. Press the space bar to view the next screen or the carriage return key to see the next line. Press any other key to terminate the program.

When SHOW begins execution, it checks the second byte of FCB1 (the first character of the file name) to see if a file name was entered on the command line. A blank in this position indicates that no file name was given. In that case, a file name is requested. Macro GFNAME is used for this purpose.

Then the file type is checked to ensure that it is not a COM file. If everything is all right, the requested file is opened. The default buffer at 80 hex is used to read the disk sectors, but we specifically set the DMA address to this address just in case it was set to some other location by the previous program.

We need to add one more macro to our library before we are ready to write the next program.

A MACRO TO ABORT THE PROGRAM FROM THE CONSOLE

Sometimes it is necessary to prematurely terminate a program for one reason or another—perhaps a number was entered incorrectly from the console, or perhaps the program provided enough information at the beginning that the remainder of the program is not needed. For these reasons, many operating systems allow an executing program to be prematurely terminated. Unfortunately, the CP/M operating system does not provide this feature. Let us therefore write macro ABORT to prematurely terminate a program. Enter the macro shown in Figure 6.8 into your macro library.

Let us see how macro ABORT works. The macro reference is placed in a program where you would like to check for termination. The console status is determined with BDOS function 11. On return from this function, the accumulator has a value of FF hex if a console key was pressed; the accumulator is zero otherwise. The macro then generates instructions to rotate the accumulator into the carry flag and check the carry flag. If the status indicates that no console key was pressed, the program branches around the remainder of the macro.

If the carry flag is set, then a console key has been pressed. If the parameter was omitted from the macro reference, then the program will terminate. However, if a parameter was included in the macro reference, then the character typed at the console is compared to this parameter. If they match, the program is terminated by a branch to the label DONE.

```
ABORT     MACRO     CHAR
;;(Put current date here)
;;Inline macro to abort program when
;;console key given by CHAR is pressed.
;;Any key will do if CHAR omitted.
;;Branch to DONE on abort.
;;Usage:    ABORT     ESC
;;
;;Macro needed: READCH
;;
          LOCAL     AROUND
          PUSH      H
          PUSH      D
          PUSH      B
          MVI       C,11          ;console status
          CALL      BDOS
          POP       B
          POP       D
          POP       H
          RRC
          JNC       AROUND        ;no character
          READCH                  ;get char
          IF        NUL CHAR
          JMP       DONE
          ELSE
          CPI       CHAR
          JZ        DONE
          ENDIF
AROUND:                           ;;ABORT
          ENDM
```

Figure 6.8: Macro ABORT to Terminate a Program from the Console

Let us use an example to clarify this. We will always use the macro reference

```
ABORT    ESC
```

for the programs in this book. The macro will then generate instructions to terminate the program only if the escape key has been pressed. Any other key will be ignored.

We are now ready to write our next program.

DISPLAYING A BINARY FILE ON THE CONSOLE

The program in Figure 6.7 will display an ASCII file on the video screen, but it cannot be used for a binary file. Sometimes, however, it is necessary to study a binary executable (COM) file. This can be accomplished with the program given in Figure 6.9. Of course, this program can also display an ASCII file, but the output is not as readable as the output from SHOW. Type the program using the file name DUMP. Assemble it and execute it. The command line is the same as that used for SHOW.

The output from DUMP is similar to that from DDT. Each line begins with the corresponding memory location (starting at the beginning of the TPA). Then 16 bytes are given in hexadecimal. The ASCII equivalents of the characters are also given if they are printable; a decimal point is shown otherwise. The display freezes after the screen is filled. Pressing the space bar displays the next screen, while a carriage return shows the next line. Press the escape key to terminate the program. (We check for termination at the end of each line.)

The remaining two programs in this chapter further demonstrate the use of our disk-related macros. Both programs use our new macro to read existing disk files.

```
TITLE   'DUMP binary file to console'
;
;(Put current date here)
;
;Usage: DUMP (file name)
;space bar = next screen
;<CR> = next line
;<ESC> = abort
;
FALSE   EQU     0
TRUE    EQU     NOT FALSE
;
BOOT    EQU     0           ;system reboot
BDOS    EQU     5           ;BDOS entry point
FCB1    EQU     5CH         ;input FCB
DBUFF   EQU     80H         ;default buffer
TPA     EQU     100H        ;program start here
LMAX    EQU     23          ;maximum lines
```

Figure 6.9: Program DUMP to View a Binary File

```
;
;Set flags in main program so only one
;copy of certain subroutines will be generated.
;Place set lines before MACLIB call.
;
CIFLAG    SET     FALSE     ;input console char
COFLAG    SET     FALSE     ;output console char
CRFLAG    SET     FALSE     ;carr-ret/line-feed
CXFLAG    SET     FALSE     ;binary in C to hex
DMFLAG    SET     FALSE     ;set DMA
FLFLAG    SET     FALSE     ;fill characters
FNFLAG    SET     FALSE     ;read file name
GTFLAG    SET     FALSE     ;get char from buffer
OPFLAG    SET     FALSE     ;open disk file
PRFLAG    SET     FALSE     ;print console buffer
RCFLAG    SET     FALSE     ;read console buffer
RDFLAG    SET     FALSE     ;read disk sector
;end of flags
;
          MACLIB  CPMMAC
;
ORG       100H
;
START:
          ENTER
          VERSN   '(current date).DUMP '
          LDA     FCB1+1
          CPI     BLANK     ;file name?
          JNZ     OP3       ;yes
          GFNAME  FCB1      ;get file name
OP3:
          OPEN    FCB1      ;input disk file
          SETDMA  DBUFF     ;sector location
          LXI     H,TPA     ;display pointer
          SHLD    PNTR
          PRINT   'Space bar for next screen, '
          PRINT   '<CR> next line, <ESC> to abort'
NEWLN:                      ;start new line
```

Figure 6.9 (continued)

```
        CRLF
        PUSH    H           ;buffer pointer
        LHLD    PNTR        ;display pointer
        OUTHEX  H           ;show address
        OUTHEX  L
        LXI     D,10H       ;next line
        DAD     D
        SHLD    PNTR        ;save
        POP     H           ;buffer pointer
        PCHAR   BLANK
NEXT:
        MOV     A,H         ;check pointer
        ORA     A           ;still 80-FF hex?
        JZ      NEXT2       ;yes
        READS   FCB1        ;read a sector
        JNZ     DONE        ;end of file
        LXI     H,DBUFF
NEXT2:
        OUTHEX  M
        INX     H
        MOV     A,L
        ANI     0FH         ;line end?
        JZ      PASC        ;yes
        ANI     3           ;space
        JNZ     NEXT        ;no
        PCHAR   BLANK
        JMP     NEXT
PASC:                       ;ASCII dump
        PRINT   '   '
        PUSH    H           ;buffer pointer
        LXI     D,-10H
        DAD     D           ;back up pointer
PAS2:
        MOV     A,M
        INX     H
        CPI     7FH         ;high bit on?
        JNC     PAS3        ;yes
        CPI     BLANK       ;control char?
```

Figure 6.9 (continued)

```
        JNC     PAS4        ;no
PAS3:
        MVI     A,PERIOD    ;change
PAS4:
        PCHAR               ;print it
        MOV     A,L
        ANI     0FH         ;line end?
        JNZ     PAS2        ;no
        POP     H           ;buffer pointer
        ABORT   ESC         ;quit?
        LDA     LINE
        DCR     A
        STA     LINE
        JNZ     NEWLN
        MVI     A,LMAX
        STA     LINE
;
;freeze line until space bar pressed
FREEZ:
        READCH              ;wait for input
        CPI     BLANK       ;space bar?
        JZ      NEWLN
        ANI     1FH         ;convert to control
        CPI     CR          ;next line?
        JNZ     FREZ2       ;no
        MVI     A,1         ;one line
        STA     LINE
        JMP     NEWLN
LINE:
        DB      LMAX        ;line count
FREZ2:
        CPI     ESC         ;abort?
        JNZ     FREEZ       ;no
DONE:
        EXIT
;
PNTR:   DS      2           ;display pointer
        END     START
```

Figure 6.9 (continued)

AUTOMATIC ENVELOPE ADDRESSING

If you use a word processor such as WordStar to write letters, you can print the letter on the computer list device. However, you will still need a typewriter to address the envelope. With the program given in Figure 6.10, you can automatically print the envelope after you have printed the letter.

A WordStar-compatible file for the beginning of a letter might look like this:

```
..Name of sender
.op   (omit page numbers)
(blank line)
Today's date
(blank line)
Name of addressee
Street address
City, State   Zip code
(blank line)
Salutation
```

Word processors typically interpret a special character in column 1 as the beginning of a command line. This character is frequently a period, because a period will not otherwise appear in the first column. The first line of this file begins with two periods, the WordStar symbol for a comment line.

The program given in Figure 6.10 can extract the recipient's name and address and print it onto an envelope. If the originator's name is included at the beginning of the file as a comment, it will be printed in the return-address area. Create a file named ADDRESS and enter the text shown in Figure 6.10. Assemble the program and run it.

The ADDRESS program is executed by typing its name and the name of the letter file. Alternatively, the file name can be given after the program has started. This program has an additional feature. The originator's name is normally placed in the upper left corner of the envelope. If, however, a separate letter L is given after the file name, then the sender's name is aligned with the recipient's name and address. This form is more suitable for addressing labels.

CHECKING FOR PAIRED CONTROL CHARACTERS

Our final program will check for paired control characters in disk files. We use macro GFNAME to input a file name from the console, macro

```
TITLE   'ADDRESS envelope from letter'
;
;(Put current date here)
;
;Usage:
;   ADDRESS DISKFILE.EXT     (for envelope)
;   ADDRESS DISKFILE.EXT L   (for label)
;
;Letter file has the form:
;..Author (for return address)
;.op and other dot commands (optional)
;blank line (optional)
;Date (one line)
;blank line (one or more)
;Address
;blank line
;
FALSE   EQU     0
TRUE    EQU     NOT FALSE
;
BOOT    EQU     0               ;system reboot
BDOS    EQU     5               ;BDOS entry point
;
FCB1    EQU     5CH             ;first parameter
FCB2    EQU     6CH             ;second parameter
DBUFF   EQU     80H             ;default buffer
TPA     EQU     100H            ;transient program area
BEL     EQU     7
;
;Set flags in main program so only one
;copy of certain subroutines will be generated.
;Place set lines before MACLIB call.
;
CMFLAG  SET     FALSE           ;ASCII compare
COFLAG  SET     FALSE           ;output console char
CRFLAG  SET     FALSE           ;carr-ret/line-feed
FLFLAG  SET     FALSE           ;fill characters
FNFLAG  SET     FALSE           ;get file name
LOFLAG  SET     FALSE           ;list output
OPFLAG  SET     FALSE           ;open disk file
```

Figure 6.10: Program ADDRESS to Automatically Address an Envelope

```
PRFLAG    SET     FALSE           ;printer output
RCFLAG    SET     FALSE           ;read console
RDFLAG    SET     FALSE           ;read disk file
;end of flags
;
          MACLIB  CPMMAC
;
ORG       TPA
;
START:
          ENTER
          VERSN   '(current date).ADDRESS '
          LDA     FCB1+1
          CPI     BLANK           ;file name?
          JNZ     OPEND           ;yes
          GFNAME  FCB1            ;get file name
OPEND:
          COMPRA  'COM', FCB1+9   ;COM file?
          JZ      NOCOM           ;yes
          MVI     A,35            ;envelope indentation
          STA     INDNC           ;save count
          LDA     FCB2+1          ;2nd parameter?
          CPI     BLANK
          JZ      NOPAR           ;no
          MVI     A,14
          STA     INDNC           ;label indentation
NOPAR:
          OPEN    FCB1            ;source file
          READS   FCB1            ;first sector
          LXI     H,DBUFF         ;text buffer
;find period with author
          MOV     A,M
          CPI     PERIOD
          JNZ     NOPER           ;no author
          INX     H
          MOV     A,M
          CPI     PERIOD          ;second dot?
          JNZ     FPER            ;no
          INX     H               ;skip dots
          MVI     B,14            ;indentation
```

Figure 6.10 (continued)

```
        CALL    PLINE           ;for author
FPER2:                          ;find other periods
        MOV     A,M
        CPI     PERIOD
        JNZ     FBLNK
        CALL    LINE
        JMP     FPER2
;
;no author, process other dot commands
;
FPER:
        CALL    LINE            ;next line
        CPI     PERIOD
        JZ      FPER
NOPER:
        MVI     B,1
        CALL    LINEFD          ;skip author
FBLNK:                          ;find blank
        MOV     A,M
        CPI     BLANK+1
        JNC     FDATE           ;not blank or CR
        CALL    LINE
        JMP     FBLNK
FDATE:                          ;find date
        CALL    LINE            ;skip to blank
        MOV     A,M
        CPI     BLANK+1
        JC      FDATE           ;additional blanks
;space down to address
        MVI     B,9
        CALL    LINEFD
;
;print address
;
ADDR2:
        MOV     A,M
        CPI     BLANK+1         ;additional
        JC      DONE
        LDA     INDNC           ;indentation
        MOV     B,A             ;for address
```

Figure 6.10 (continued)

```
            CALL    PLINE
            JMP     ADDR2
;
;send line feeds to printer, B has number
;
LINEFD:
            LCHAR   LF
            DCR     B
            JNZ     LINEFD
            RET
;
;output line to printer and console
;
PLINE:
            CALL    INDEN
            MOV     A,M             ;first character
            PCHAR
            LCHAR
PLINE2:
            CALL    CPOINT          ;check pointer
            MOV     A,M             ;next character
            PCHAR
            LCHAR
            ANI     7FH             ;mask parity
            CPI     LF
            JNZ     PLINE2          ;yes
            INX     H
            RET
;
;move to beginning of next line, after LF
;
LINE:
            CALL    CPOINT          ;check pointer
            MOV     A,M             ;next character
            ANI     7FH             ;mask parity
            CPI     LF
            JNZ     LINE            ;yes
            INX     H
            RET
;
;Increment HL pointer, see if past 80+80 hex.
;Read another sector if so.
```

Figure 6.10 (continued)

```
;
CPOINT:
        INX     H               ;pointer
        MOV     A,H             ;check pointer
        ORA     A               ;<100H?
        RZ                      ;yes, ok
        READS   FCB1            ;next sector
        JNZ     DONE            ;end of file
        LXI     H,DBUFF         ;reset pointer
        RET
INDEN:
        MVI     A,BLANK
INDEN2:
        PCHAR
        LCHAR
        DCR     B
        JNZ     INDEN2
        RET
;
NOCOM:  ERRORM  '?COM file',DONE
;
DONE:
        EXIT
INDNC:  DS      1               ;indentation
;
        END     START
```

Figure 6.10: (continued)

OPEN to open a disk file, and macro ERRORM to print an error message and terminate the program. We also introduce the inline macros REPT, IRP, and IRPC.

Some word processors use paired control characters to indicate special operations during printing. For example, in WordStar files, a passage to be underlined is enclosed in control-S characters. Other control characters are used for boldface, superscript, and subscript indicators. If the second member of the pair is inadvertently omitted, the resulting document will be unusual. For example, if the second underline character is omitted, all of the remaining words will be underlined.

The program given in Figure 6.11 can be used to check a document for paired control characters. The program is designed for use with WordStar, but it can be altered easily for use with other word processing programs.

```
TITLE   'PAIR checks pairs of control char'
;
;(Put current date here)
;
;Usage: PAIR DISKFILE.EXT
;
;Checks that control-B, -D, -S, -T, -V, and -X are paired
;
FALSE   EQU     0
TRUE    EQU     NOT FALSE
;
BOOT    EQU     0               ;system reboot
BDOS    EQU     5               ;BDOS entry point
FCB1    EQU     5CH             ;input FCB
DBUFF   EQU     80H             ;default buffer
TPA     EQU     100H            ;transient program area
;
;Set flags in main program so only one
;copy of certain subroutines will be generated.
;Place set lines before MACLIB call.
;
CIFLAG  SET     FALSE           ;input console char
CMFLAG  SET     FALSE           ;ASCII compare
COFLAG  SET     FALSE           ;output console char
CRFLAG  SET     FALSE           ;carr-ret/line-feed
DMFLAG  SET     FALSE           ;set DMA
FLFLAG  SET     FALSE           ;fill characters
FNFLAG  SET     FALSE           ;read file name
OPFLAG  SET     FALSE           ;open disk file
PRFLAG  SET     FALSE           ;print console buffer
RCFLAG  SET     FALSE           ;read console buffer
RDFLAG  SET     FALSE           ;read disk file
;end of flags
;
        MACLIB  CPMMAC
;
ORG     TPA
;
START:
```

Figure 6.11: Program PAIR to Count Pairs of Control Characters

```
        ENTER
        VERSN   '(current date).PAIR '
        LDA     FCB1+1
        CPI     BLANK           ;file name?
        JNZ     OPEND           ;yes
        GFNAME  FCB1            ;get file name
OPEND:
        COMPRA  'COM', FCB1+9   ;COM file?
        JZ      NOCOM           ;yes
        PRINT   <CR,LF,'Looking for unbalanced '>
        PRINT   '^B, ^D, ^S, ^T, ^V, ^X'
        OPEN    FCB1            ;source file
        SETDMA  DBUFF           ;use default
        LXI     H,100H          ;set pointer
LINE:
        MOV     A,H             ;check pointer
        ORA     A               ;still 80-FF?
        JZ      LIN3            ;yes
        READS   FCB1            ;read a sector
        JNZ     EOFILE          ;end of file
        LXI     H,DBUFF         ;reset pointer
        JMP     LINE
LIN3:
        MOV     A,M
        INX     H
        ANI     7FH             ;mask parity
        CPI     EOF             ;file end
        JZ      EOFILE          ;yes
;
;;inline macro to count occurrences of control char
        IRPC    X?, BDSTVX
        LOCAL   AROUND
CTR&X?  EQU     '&X?' - '@'
        CPI     CTR&X?
        JNZ     AROUND
        LDA     CNT&X?
        INR     A
        STA     CNT&X?
        JMP     LINE
```

Figure 6.11 (continued)

```
CNT&X?:  DB       0
AROUND:
         ENDM
         JMP      LINE              ;no
NOCOM:
         ERRORM   'COM file?',DONE
UFLAG:   DB       0
;
EOFILE:
;;inline macro to show unbalanced control char
         IRPC     X?, BDSTVX
         LOCAL    AROUND
         LDA      CNT&X?
         RAR                        ;;odd?
         JNC      AROUND            ;;no
         PRINT    <CR,LF,'Unbalanced ^'>
         PCHAR    '&X?'
         LDA      UFLAG
         INR      A
         STA      UFLAG
AROUND:
         ENDM
         LDA      UFLAG
         ORA      A                 ;ok
         JNZ      DONE              ;no
         PRINT    <CR,LF,'No unbalanced pairs'>
DONE:
         EXIT
;
         END      START
```

Figure 6.11 (continued)

In particular, the program counts the number of control-B, -D, -S, -T, -V, and -X characters. If there is an odd number of any of these, an error is reported. If the count is even, the message 'No unbalanced pairs' is given. Of course, if two terminal control characters of the same type are omitted, the program will not notice it.

This program, like the others in this chapter, is executed by giving its name and the name of the file to be read. The instructions are similar, but

we introduce a new feature. Two indefinite repeat macros, IRPC, are used. These macros make it easy to program sets of instruction that differ only in one letter.

The macros we have used previously are defined at the beginning of the program or in a separate macro library. Then the macro name and any parameters are placed in the program wherever they are needed. The repeat macros are different. They are defined directly within the program as they are needed.

The inline macros begin with the name REPEAT, IRP, or IRPC and end with the usual ENDM statement. There is no other name associated with this type of macro. Following is the first of the two repeat macros:

```
;
;;inline macro to count occurrences of control char
            IRPC    X?, BDSTVX
            LOCAL   AROUND
CTR&X?      EQU     '&X?' - '@'
            CPI     CTR&X?
            JNZ     AROUND
            LDA     CNT&X?
            INR     A
            STA     CNT&X?
            JMP     LINE
CNT&X?:     DB      0
AROUND:
            ENDM
```

This macro generates six slightly different sets of instructions. The first parameter, X?, is a dummy variable. The second parameter contains the reference parameters—six characters, the letters B, D, S, T, V, and X. The macro is therefore expanded six times. For the first copy, the parameter B replaces the X? symbol. The ampersand is a linking character. Its occurrence next to the original dummy parameter indicates that the reference parameter is to be joined with the adjacent text. For example, the first expansion will produce the following passage:

```
0002+=          CTRB    EQU     'B' - '@'
0396+FE02               CPI     CTRB
0398+C2A603             JNZ     ??0037
039B+3AA503             LDA     CNTB
039E+3C                 INR     A
039F+32A503             STA     CNTB
03A2+C36903             JMP     LINE
03A5+00         CNTB:   DB      0
```

There will be five similar sections following this one. At each macro expansion, the ampersand characters are removed by the assembler after joining the actual parameter. Some macro assemblers leave the ampersand character in the final listing. The JNZ ??0037 instruction causes a branch to the end of this passage, address 3A6. Notice that control-B, binary two, is obtained by subtracting the at-sign from the letter B. The other control characters are created similarly.

SUMMARY

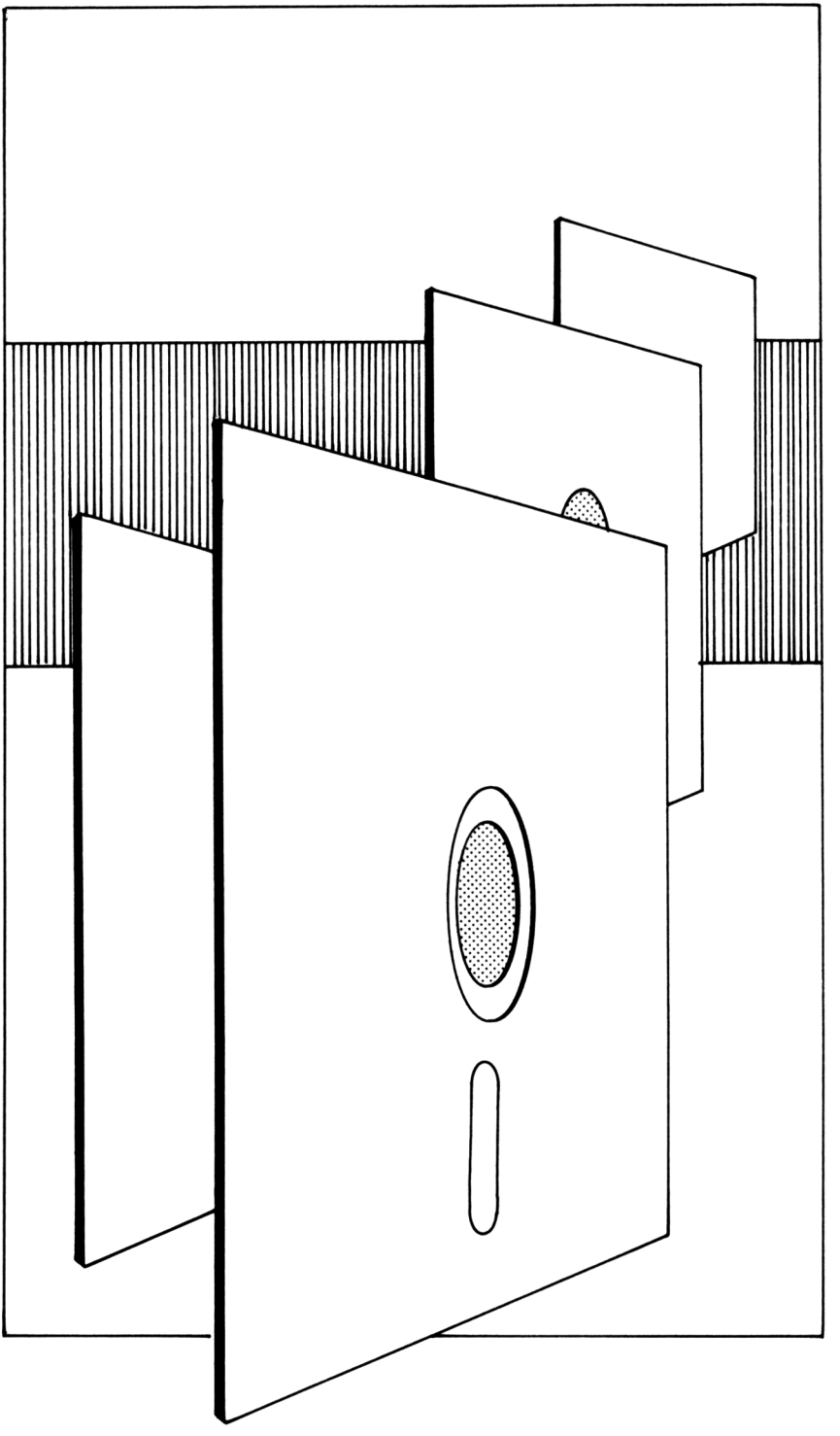
In this chapter we have learned how the file control block describes and manages files on the disk, and we have learned how to read a disk file. We wrote a macro to print an error message, one to open an existing disk file, one to set the DMA address, one to read a disk sector, one to input a file name after a program has begun executing, and another to abort a program from the console. We also looked briefly at the inline repeat macro IRPC.

We wrote several executable programs that demonstrate uses for these macros. SHOW prints an ASCII file on the console; DUMP displays a binary file in hex and ASCII; ADDRESS copies the address from a letter file onto an envelope; and PAIR checks a text file for balanced control characters. In the next chapter we will develop macros and programs that deal with writing disk files.

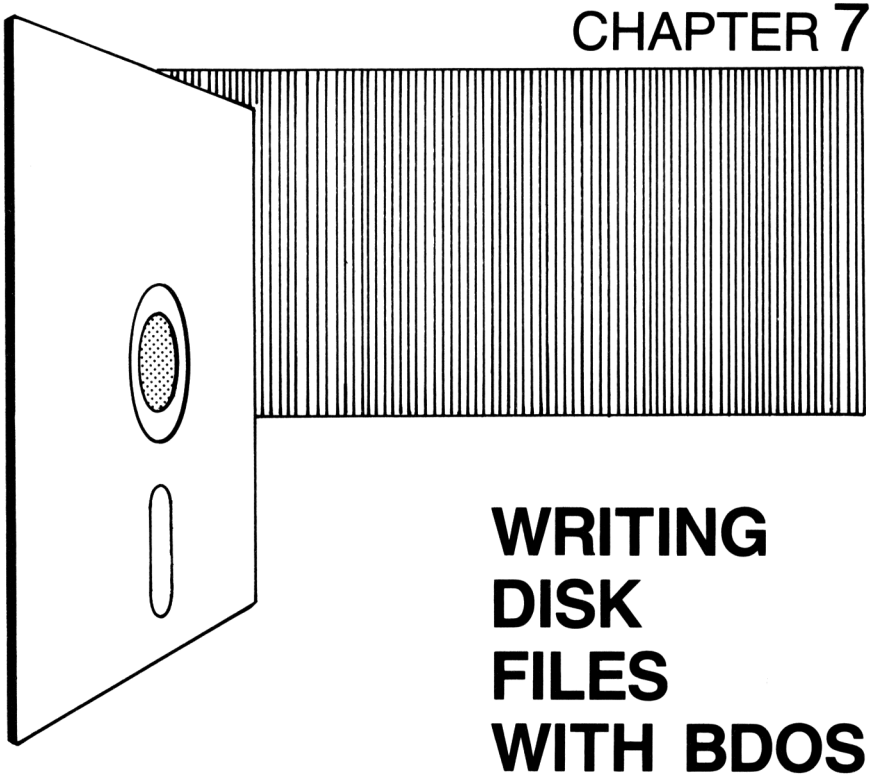
Your macro library directory should now look like this:

```
;;Macros in this library                          Flags
;;ABORT    MACRO   CHAR                  CIFLAG, COFLAG
;;AMBIG    MACRO   OLD, NEW              (none)
;;COMPAR   MACRO   FIRST, SECOND, BYTES  CMFLAG
;;COMPRA   MACRO   FIRST, SECOND, BYTES  CMFLAG
;;CPMVER   MACRO                         (none)
;;CRLF     MACRO                         CRFLAG, COFLAG
;;ENTER    MACRO                         (none)
;;ERRORM   MACRO   TEXT, WHERE           COFLAG, CRFLAG, PRFLAG
;;EXIT     MACRO   SPACE?                (none)
;;FILL     MACRO   ADDR, BYTES, CHAR     FLFLAG
;;GFNAME   MACRO   FCB                   FNFLAG, FLFLAG, REFLAG
;;                                       COFLAG, CRFLAG, PRFLAG
;;HEXHL    MACRO   POINTR                HXFLAG, RCFLAG
;;LCHAR    MACRO   PAR                   LOFLAG
;;MOVE     MACRO   FROM, TO, BYTES       MVFLAG
;;OPEN     MACRO   POINTR, WHERE         OPFLAG, COFLAG, PRFLAG
```

```
;;OUTHEX   MACRO   REG          CXFLAG, COFLAG
;;PCHAR    MACRO   PAR          COFLAG
;;PRINT    MACRO   TEXT, BYTES  PRFLAG, COFLAG
;;READB    MACRO   BUFFR        RCFLAG
;;READCH   MACRO   REG          CIFLAG, COFLAG
;;READS    MACRO   POINTR, STAR RDFLAG, COFLAG
;;SBC      MACRO                (none)
;;SETDMA   MACRO   POINTR       DMFLAG
;;SYSF     MACRO   FUNC, A,E    (none)
;;UCASE    MACRO   REG          (none)
;;UPPER    MACRO   REG          (none)
;;VERSN    MACRO   NUM          (none)
```



CHAPTER 7



WRITING DISK FILES WITH BDOS

INTRODUCTION

The programs we have written so far have not changed the disk itself. We have developed programs for reading disk files, but not for writing them. In this chapter we will write macros MAKE, UNPROT, PFNAME, DELETE, SETUP2, RENAME, CLOSE, and WRITES for creating and altering disk files. We will also write several executable programs: COPY for duplicating an existing disk file, CRYPT for encrypting a file, RENAME for renaming files, and DELETE for deleting files. Notice that we use RENAME and DELETE both as program names and macro names. CP/M uses the program name and the assembler uses the macro name, so there is no conflict. Let us begin with macro MAKE.

A MACRO TO CREATE A NEW DISK FILE

We saw in Chapter 6 that it is necessary to open an existing disk file with BDOS function 15 before it can be read. To create a new disk file, we must use BDOS function 22. The first part of a file control block (FCB) is

created in memory, just as it is when opening an existing disk file. The first byte of the memory FCB designates the disk drive. A value of 0 indicates the default drive, 1 is drive A, 2 is drive B, and so on. The file name and file type are placed in the next 11 bytes. The DE register is loaded with the FCB address, and register C is given the value of 22. A call to address 5 completes the operation.

Macro MAKE, shown in Figure 7.1, can be used to create a new disk file by allocating an FCB in the disk directory. The parameter POINTR references the location of the memory FCB. Add macro MAKE to your macro library.

One or more blocks of sectors on each disk are allocated to the disk directory. The exact number is fixed, but it will differ from one disk format to another. The number of directory entries is also fixed, because there are four disk FCBs for each 128-byte sector. At some point, all of the allocated directory space may be in use. Consequently, when BDOS function 22 is used to create a new disk file, it determines whether there is room for another FCB.

On return from BDOS function 22, the accumulator is set to a value of FF hex if the directory is filled. Macro MAKE therefore checks the accumulator after return from BDOS. If the directory is full, an error message is printed and the program branches to location DONE. The flag MKFLAG ensures that only one copy of subroutine MAKE2? is created. Macro SYSF performs the BDOS call, and macro ERRORM prints the appropriate error message if there is no directory space.

Our next macro changes the read-only attribute of a disk file to read/write.

UNPROTECTING A DISK FILE

CP/M disk files can be protected against accidental erasure by setting the read-only feature. If we want to alter or erase a file, we must make sure that it is set to read/write. This feature is implemented in CP/M version 2 by coding the first character of the file type. If the high-order bit of this character is set to 1, then the file is considered to be write protected. If this bit is reset to 0, the file can be altered or erased.

Let us observe this phenomenon with DDT or SID. Go to drive A and determine the attributes of the executable files with the command

```
STAT *.COM
```

This will list all COM files in alphabetical order. The symbol R/O will appear in front of those files that are protected (read only). If the symbol

```
MAKE    MACRO   POINTR
;;(Put current date here)
;;Inline macro to create a new disk file.
;;POINTR refers to file control block.
;;Extent and current record number are zeroed.
;;Macros needed: SYSF, ERRORM
;;
        LOCAL   AROUND
        LXI     D,POINTR
        XRA     A               ;zero
        STA     POINTR+12       ;extent
        STA     POINTR+32       ;current record
        CALL    MAKE2?
        INR     A               ;0=ok, FF means error
        JNZ     AROUND
        ERRORM  'No directory space', DONE
        IF      NOT MKFLAG
MAKE2?: SYSF    22              ;make new disk file
MKFLAG  SET     TRUE            ;only one copy
        ENDIF
AROUND:                         ;;MAKE
        ENDM
```

Figure 7.1: Macro MAKE to Create a New Disk File

R/W (read/write) is shown instead, the file is not protected. The listing might look like this:

```
 Recs  Bytes Ext  Acc
    6    2k   1  R/O A:SAVEUSER.COM
    6    2k   1  R/W A:SHOW.COM
   42    6k   1  R/O A:STAT.COM
   10    2k   1  R/O A:SUBMIT.COM
   12    2k   1  R/O A:SYSGEN.COM
Bytes Remaining On A: 6k
```

We will need a protected file for the next step. If all of the files are unprotected, use STAT to change the protection of one of them—STAT itself, for example. Give the command

```
STAT STAT.COM $R/O
```

Be sure to place a space in front of the dollar sign but not afterward. Give the STAT command again to ensure that STAT is protected.

Execute the debugger DDT or SID by typing its name, but do not give any parameters at this time. We will now write a small program in memory starting at 4000 hex, using the A command:

```
A4000
4000 LXI D,5C
4003 MVI C,0F
4005 CALL 5
4008 RST 7
```

(Type an extra carriage return to finish the program.) Do not execute this program just yet. When it is executed, it will open the disk file named in the FCB at address 5C hex, the value in register DE. Register C is loaded with the value 0F hex (15 decimal), the BDOS open function. After returning from the BDOS call, the routine branches to restart 7 at address 37 hex, the normal return to the debugger.

We will now create the first part of an FCB at location 5C hex. The debugger is used for this step. Give the command

```
ISTAT.COM
```

The I command initializes a memory FCB for file name STAT.COM on the default disk. Observe the results by displaying the FCB region with the command D50,6F:

```
0050: 00 00 00 00 00 00 00 00 00 00 00 00 00 53 54 41 .............STA
0060: 54 20 20 20 20 43 4F 4D 00 00 00 00 00 20 20 20 T    COM.....
```

Notice that the four remaining characters in the file name STAT are blanks. Now give the debugger command G4000. This will execute the program we wrote at 4000 hex. The default drive will start up, and then control will return to the debugger. Examine memory again with the debugger command D50,6F:

```
0050: 00 00 00 00 00 00 00 00 00 00 00 00 00 53 54 41 .............STA
0060: 54 20 20 20 20 C3 4F 4D 00 00 80 2A 06 07 08 00 T    .OM...*....
```

In this example we see that the ASCII representation of the file name has been changed from

```
STAT    COM
```

to

```
STAT    .OM
```

When the BDOS open function (15) was executed, CP/M changed the file type of the memory FCB to match the file type of the disk FCB. The first character of the file type, the letter C, has been changed. Now look at the hexadecimal representation of this character (address 65 hex). The original value of 43 hex has been changed to C3 hex. The hexadecimal and corresponding ASCII values are as follows:

```
43 4F 4D   COM
C3 4F 4D   .OM
```

Comparing the two, we see that they differ in the high-order bit used to indicate write protection:

Hex	*Binary*
43	0100 0011
C3	1100 0011

You can return to the system level of CP/M, change the protection attribute of STAT, and repeat the above steps. In this case, the file extension will remain COM after the open function is executed.

We will now write macro UNPROT (shown in Figure 7.2) to unprotect a disk file with BDOS function 30. This BDOS function can set the four file attributes—read only (R/O), read/write (R/W), system (SYS), and directory (DIR). We will only use it to unprotect a file. This macro resets the high-order bit of the memory FCB referenced by the parameter POINTR. The accumulator is loaded with the first character of the file type in position 9. The high-order bit is reset by performing a logical AND with the value of 7F hex (0111 1111). The result is put back into place. Macro SYSF is used to perform BDOS function 30, which changes the extension of the disk FCB to match the memory FCB.

Add macro UNPROT to your library. The flag UNFLAG ensures that only one copy of subroutine UNPR2? will be created. Our next macro displays the file name of a memory FCB on the console screen.

A MACRO TO PRINT AN FCB FILE NAME

We have seen that the first part of a memory FCB specifies the disk drive in position 0 and the file name in positions 1 – 8. Names shorter than eight characters are filled out with blanks. Positions 9 – 11 contain the file type. Thus the file name LONGNAME.EXT is actually stored as LONGNAMEEXT. A shorter name, such as A.TYP, will be coded as A_____TYP (the underline characters represent blanks). Because we will occasionally need to display the file name associated with an FCB, let us now write a macro for this purpose.

```
UNPROT   MACRO POINTR
;;(Put current date here)
;;Inline macro to convert R/O file to R/W.
;;POINTR refers to file control block.
;;Macro needed: SYSF
;;
         LOCAL   AROUND
         LXI     D,POINTR
         LDA     POINTR+9      ;load from file type
         ANI     7FH           ;set for R/W
         STA     POINTR+9      ;store at beginning of file type
         CALL    UNPR2?
         IF      NOT UNFLAG
         JMP     AROUND
UNPR2?:
         SYSF    30            ;set file attributes
UNFLAG   SET     TRUE          ;only one copy
         ENDIF
AROUND:                        ;;UNPROT
         ENDM
```

Figure 7.2: Macro UNPROT to Unprotect a Disk File

Macro PFNAME, shown in Figure 7.3, displays the referenced file name in its usual CP/M form rather than the way it is stored in the FCB. Blank characters are removed and a period is placed between the primary name and the extension. Macros PRINT and PCHAR are used. Add macro PFNAME to your library.

A MACRO TO DELETE A DISK FILE

We have seen that the first byte of the memory FCB begins with the drive type, while the first byte of the disk FCB contains the user number. To delete a file, the initial byte of the disk FCB must be changed to a value of E5 hex. The remainder of the FCB and the actual file are not altered. This new value allows the disk space allotted for that file to be written over. Only when this happens is the file actually changed.

BDOS function 19 is used to delete a disk file. We will perform this operation with macro DELETE, shown in Figure 7.4. The macro begins

```
PFNAME    MACRO    FCB
;;(Put current date here)
;;Inline macro to print file name as
;; FIRST.EXT
;;FCB is file control block.
;;Macros needed: PCHAR, PRINT
;;
          LOCAL    PFNA2?, PFNA3?
          PUSH     H
          PUSH     B
          MVI      B,8          ;name length
          LXI      H,FCB + 1    ;start
PFNA3?:
          MOV      A,M          ;get char
          CPI      BLANK
          JZ       PFNA2?       ;end
          PCHAR                 ;print
          INX      H
          DCR      B
          JNZ      PFNA3?
PFNA2?:
          POP      B
          POP      H
          PCHAR    '.'
          PRINT    FCB+9, 3     ;exten
                                ;;PFNAME
          ENDM
```

Figure 7.3: Macro PFNAME to Print the File Name Associated with an FCB

```
DELETE    MACRO    POINTR, WHERE
;;(Put current date here)
;;Inline macro to delete an existing disk file.
;;POINTR refers to file control block.
;;If file is protected, branch to WHERE or DONE.
;;
;;Macros needed: SYSF, UNPROT, READCH,
```

Figure 7.4: Macro DELETE to Delete a Disk File

```
;; PFNAME, PRINT, UCASE, CRLF
;;
            LOCAL   AROUND, DEL3?
            LXI     D,POINTR
            LDA     POINTR+9
            ANI     80H             ;protected?
            JZ      DEL3?           ;no
            CRLF
            PFNAME  POINTR
            PRINT   ' is READ ONLY. Delete? '
            READCH
            UCASE
            CPI     'Y'
            IF      NOT NUL WHERE
            JNZ     WHERE
            ELSE
            JNZ     DONE            ;quit
            ENDIF
            UNPROT  POINTR
DEL3?:
            CALL    DEL2?
            IF      NOT DEFLAG
            JMP     AROUND
DEL2?:
            SYSF    19              ;delete disk file
DEFLAG      SET     TRUE            ;only one copy
            ENDIF
AROUND:                             ;;DELETE
            ENDM
```

Figure 7.4 (continued)

by loading the DE register with the FCB address. Then the first character of the file type (at FCB1+9) is inspected to see whether the file is write protected. If it is, the file name is displayed on the console and permission to delete it is requested. If the user enters a Y, the file is unprotected with macro UNPROT. If any other character is entered, the macro terminates with a branch to the second parameter WHERE if it has been provided. Otherwise, the program branches to DONE. Notice that macro DELETE

references several other macros in our library. Add this macro to the library.

There is a further complication if the file is protected. An unprotected file can be deleted without first performing an open function, but a protected file cannot. We saw previously in this chapter that the first character of the file type is altered if the file is protected. It is very important that a CP/M open command be issued prior to executing the delete function, or BDOS will not be able to find the file. For example, if you want to delete a protected file called FIRST.COM, you must search for a file that looks like FIRST..OM. The open function will convert the requested file name to the form needed by BDOS. The open function is not incorporated into macro DELETE, but in our programs we will always be careful to open a file prior to using macro DELETE.

INVESTIGATING TWO FILE CONTROL BLOCKS WITH THE DEBUGGER

We have already learned how CP/M can help us construct a memory FCB from a parameter given on the command line. We used the debugger DDT or SID in this investigation. Let us continue this study by using two parameters on the command line. Be sure to choose file names that do not exist, or the debugger will load the requested files and delete the memory FCBs. Give the command

```
A:DDT FIRST.EXT SECOND.TYP
```

or

```
A:SID FIRST.EXT SECOND.TYP
```

Look at the results with the command

```
D50,9F
```

The console screen should look like this:

```
0050 00 00 00 00 00 00 00 00 00 00 00 00 00 46 49 52 .............FIR
0060 53 54 20 20 20 45 58 54 00 00 00 00 00 53 45 43 ST   EXT.....SEC
0070 4F 4E 44 20 20 54 59 50 00 00 00 00 00 FF 00 BF OND  TYP........
0080 15 20 46 49 52 53 54 2E 45 58 54 20 53 45 43 4F . FIRST.EXT SECO
0090 4E 44 2E 54 59 50 00 00 00 00 00 00 00 00 00 00 ND.TYP..........
```

Notice that the first parameter, FIRST.EXT, appears as an FCB starting at 5C hex. The first byte is a binary zero, specifying the default drive, because no disk drive was included in the file name. The primary name

FIRST appears next in uppercase letters. Three blanks fill out the eight-character field. The three letters of the extension appear next.

The second parameter, SECOND.TYP, has been treated similarly. The first part of another FCB begins at 6C hex. The command line tail containing both parameters begins at location 82 hex. The length of this tail, 15 hex, is stored at location 80 hex.

Return to CP/M by typing control-C; then type the command

```
A:DDT B:FIRST.EXT B:SECOND.TYP
```

or

```
A:SID B:FIRST.EXT B:SECOND.TYP
```

Again, examine the beginning of memory with the debugger. The result should look like this:

```
0050 00 00 00 00 00 00 00 00 00 00 00 00 02 46 49 52 .............FIR
0060 53 54 20 20 20 45 58 54 00 00 00 00 02 53 45 43 ST   EXT.....SEC
0070 4F 4E 44 20 20 54 59 50 00 00 00 00 00 FF 00 BF OND  TYP........
0080 19 20 42 3A 46 49 52 53 54 2E 45 58 54 20 42 3A . B:FIRST.EXT B:
0090 53 45 43 4F 4E 44 2E 54 59 50 00 00 00 00 00 00 SECOND.TYP......
```

Notice that the command line tail shows that drive B was specifically requested. Furthermore, the drive types at addresses 5C and 6C hex contain a value of 2, indicating that drive B was requested.

Return to CP/M with control-C, and for the third test give the command

```
DDT FIRST.EXT *.TYP
```

or

```
SID FIRST.EXT *.TYP
```

Examine memory from 50 to 9F hex:

```
0050 00 00 00 00 00 00 00 00 00 00 00 00 00 46 49 52 .............FIR
0060 53 54 20 20 20 45 58 54 00 00 00 00 00 3F 3F 3F ST   EXT.....???
0070 3F 3F 3F 3F 3F 54 59 50 00 00 00 00 00 FF 00 BF ?????TYP........
0080 10 20 46 49 52 53 54 2E 45 58 54 20 2A 2E 54 59 . FIRST.EXT *.TY
0090 50 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 P...............
```

In this example the first FCB, starting at address 5C hex, looks as it did in the previous tests. However, the first part of the second FCB, starting at address 6C hex, is filled with question marks. When an asterisk appears in a file name, CP/M expands the field in the FCB with question marks, the wild-card character. However, the command tail starting at 82 hex still shows the asterisk.

OPENING A FILE WHEN TWO FILE NAMES ARE GIVEN

Later in this chapter we will write a program to create a new file that is a duplicate of an existing file. When the command line

```
COPY FIRST SECOND
```

is typed, CP/M will automatically set up the beginnings of two FCBs starting at 5C and 6C hex. Another program we will write compares two files. The command line will be as follows:

```
VERIFY FIRST SECOND
```

The programs we have written up to now require a single parameter. We have used an FCB at 5C hex for the file. However, when there are two parameters the situation is more complicated.

CP/M has created the beginning of two FCBs starting at 5C hex and 6C hex. However, if our program opens the first file at this point, the second file name will be destroyed. Remember, a complete FCB is 32 bytes long. The programmer constructs the first part of the FCB, and CP/M fills in the remainder when the file is actually opened. Thus, if the first FCB begins at address 5C hex, it will extend to address 7B hex after the open function is executed. The second half of the first FCB will overwrite the first part of the second FCB.

You can investigate this problem with the debugger. Execute DDT or SID but do not provide a parameter. Then give the command

```
ISTAT.COM
```

as we did previously in this chapter. This command will initialize an FCB for STAT.COM at address 5C hex. The second FCB at address 6C hex is automatically filled with blanks because a second parameter was not given. Fill the second FCB area with the value of 40 hex, an ASCII @, by giving the command

```
F6C,7F,40
```

Observe the results with the command

```
D50,7F
```

The results should look like this:

```
0050: E0 D9 00 FF 00 FF 00 FF 00 FF 00 00 00 53 54 41 .............STA
0060: 54 20 20 20 20 43 4F 4D 00 00 00 00 40 40 40 40 T    COM....@@@@
0070: 40 40 40 40 40 40 40 40 40 40 40 40 40 40 40 40 @@@@@@@@@@@@@@@@
```

Notice that the at-signs coincide with the second FCB starting at 6C hex.

Using the debugger A command, write the following short program:

```
A4000
4000 LXI D,5C
4003 MVI C,0F
4005 CALL 5
4008 RST 7
```

Execute this program with the command G4000. The program calls BDOS function 15 to open a disk file. After control returns to the debugger, display memory with the command

```
D50,7F
```

Notice that the asterisks in the second FCB are gone. The open operation destroyed the information in the second FCB.

The solution to this problem is to relocate the second FCB before the first file name is opened. Macro SETUP2, given in Figure 7.5, is designed for this purpose.

Macro SETUP2 expects to find two parameters in the program command line—one will be found at 5C hex and the other at 6C hex. For example, suppose we want to alter a file in some way. The first parameter gives the name of the existing file. The second parameter is the name of the new file. Macro SETUP2 will open the first file and create a directory entry for the second file.

For some applications, it will be convenient for the user to enter only one parameter. For example, suppose we want to encrypt a file named PAYROLL.AUG. The encrypted file will be given the name of the original file and the original file will be named PAYROLL.BAK.

Macro SETUP2 begins by setting flag S2FLAG true. Macro CLOSE, which we will write later in the chapter, uses this flag. The flag signals the assembler when generating macro CLOSE to look for the DUPL flag.

Macro SETUP2 then checks to be sure that a second parameter was entered. If not, the first file name is duplicated into the second FCB at location 6C hex. Macro SETUP2 then checks for question marks in the second file name. Remember, question marks are used for ambiguous characters in the file name. If an asterisk is typed in a parameter, CP/M fills out the remainder of the field with question marks. Macro SETUP2 uses macro AMBIG to replace question marks in the second parameter with the corresponding characters of the first parameter.

The next step is to see whether the source and destination file names are identical. This of course includes the case where only one file name was originally given. In this case, the file type for the destination file is changed to $$$, the standard CP/M temporary file type. A duplicate-name flag, DUPL, is also set at this time.

```
SETUP2  MACRO
;;(Put current date here)
;;Inline macro to open two disk files.
;;Input file is the first parameter of command
;;line. The file control block is FCB1 at 5C hex.
;;The output file is the second parameter.
;;The file control block is initially FCB2 at
;;6C hex. The destination file name is moved into
;;the macro area.
;;If only one file is entered or both are the same,
;;the second file is typed $$$. Macro CLOSE
;;will rename original file BAK and give original
;;name to the destination file when S2FLAG is true.
;;
;;Other macros needed: MOVE, OPEN, MAKE, DELETE,
;; ERRORM, AMBIG, COMPAR
;
        LOCAL   AROUND, SET2?, SET3?, SET4?
S2FLAG  SET     TRUE                    ;used by macro CLOSE
;
        LDA     FCB2+1                  ;second parameter
        CPI     BLANK                   ;anything?
        JNZ     SET4?
;duplicate file name and type, keep disk name
        MOVE    FCB1+1, FCB2+1, 11      ;keep disk
SET4?:
        AMBIG   FCB1, FCB2              ;fix ??? in name?
        COMPAR  FCB1, FCB2, 12          ;both same?
        JZ      DUPNM?                  ;yes
SET2?:
        MOVE    FCB2, DFCB, 16          ;new destination
        OPEN    FCB1                    ;source file
        OPEN    DFCB, SET3?             ;destination
SET3?:
        DELETE  DFCB                    ;existing file name
        MAKE    DFCB                    ;new one
        JMP     AROUND                  ;error messages
```

Figure 7.5: Macro SETUP2 to Handle Two Disk Files

```
DUPNM?:
        MVI     A,TRUE
        STA     DUPL                    ;set dup flag
        MOVE    '$$$', FCB2+9           ;source file
        JMP     SET2?                   ;continue
;
DUPL:   DB      FALSE                   ;duplicate-name flag
;
;file control block for destination file
;
DFCB:   DS      33                      ;file 2 FCB
;
AROUND:                                 ;continue main code
                                        ;SETUP2
        ENDM
```

Figure 7.5 (continued)

Macro CLOSE will check the DUPL flag to see if only one file name was given or if identical names were given. In this case, macro CLOSE changes the file type of the first name to BAK. It also changes the file type of the second name from $$$ to the type of the first name.

The second parameter is now moved from location 6C hex to a default file control block named DFCB, which is located within macro SETUP2. The directive DS (define storage) 33 sets aside 33 bytes for the FCB. Now that the way is clear, the first file name can be opened safely. Macro OPEN, which we wrote in the previous chapter, is used for this purpose. It will terminate the program and give the appropriate error message if the source file cannot be found.

The next step is also very important. When we save a file with the CP/M command SAVE, any existing file with the same name is automatically erased. However, we are going to create a disk FCB from a memory FCB using BDOS function 22. In this case CP/M will allow us to create a disk file name that duplicates an existing file name. There would then be two identical names in the directory. So before you create a new disk FCB, you must ensure that another file with the same name does not exist. This is most easily accomplished by using the BDOS delete function. This step will delete the file name if it exists. If the name does not exist, no harm is done. (The delete command does not alter the memory FCB.) Macro SETUP2, therefore, deletes the file name given as the second parameter.

A MACRO TO RENAME A DISK FILE

Each CP/M file is referenced by one or more FCB entries in the disk directory. We can change the name of a file by changing the FCB. BDOS function 23 is used for this purpose. This operation does not alter the file itself. It only changes the disk FCB. The programmer sets up the first 12 bytes of a memory FCB for the original file name and then opens the file with BDOS function 15. A memory FCB for the new file name is placed 16 bytes beyond the original name. The drive code for the original file name is the usual value, 0 for default, 1 for drive A, and so on, but the drive code for the new name is set to 0. If the FCB for the original file name is located at address 5C hex, the FCB for the new name is located at address 6C hex.

The macro shown in Figure 7.6 can be used to rename a disk file. Add macro RENAME to your library. The parameter POINTR refers to the memory FCB for the original file name. The programmer must open the original file and then place a memory FCB for the new name 16 bytes beyond the original name. At this point, macro RENAME can be referenced. Notice that the new name must not be in place before the original file name is opened, or the new name will be destroyed by the open function.

BDOS function 23 locates a disk FCB that matches the memory FCB referenced by POINTR. It then changes the disk FCB to match the memory FCB referenced by POINTR+16.

Macro RENAME first checks to see whether the original file is protected. If so, the file is unprotected with macro UNPROT. BDOS is then called to rename the file. Macro RENAME displays both file names on the console. A right-pointing arrow indicates that the original name was changed to the new name. For example, if SORT.ASM is renamed to SORT.BAK you will see the following statement on the console:

```
SORT    ASM ==> SORT    BAK
```

A MACRO TO WRITE A DISK SECTOR

In Chapter 6 we wrote macro READS to read a sector from disk into memory. The sector is placed in the default buffer area starting at address 80 hex, unless the DMA address has been redefined by BDOS function 26. The complementary operation, writing a disk sector from the console buffer, is similar. It is performed with BDOS function 21. The default memory location is again 80 hex unless it is changed by BDOS function 26.

Add macro WRITES, given in Figure 7.7, to your macro library. There are two parameters to this macro, both of which are optional. The first parameter, POINTR, references the FCB where the file name is given. If this parameter is omitted, the macro assumes that DE has been previously

```
RENAME    MACRO    POINTR
;;(Put current date here)
;;Inline macro to rename an existing disk file.
;;POINTR refers to original name.
;;New name is at POINTR + 10H.
;;Macros needed: SYSF, PRINT, UNPROT, CRLF
;;
          LOCAL    AROUND, REN2?
          LXI      D,POINTR
          LDA      POINTR+9
          ORI      80H              ;;file R/O?
          JZ       REN2?            ;;no
          UNPROT   POINTR           ;;make R/W
REN2?:
          CALL     RENAM?
          CRLF
          PRINT    POINTR+1, 11
          PRINT    '==>'
          PRINT    POINTR+11H, 11
          IF       NOT RNFLAG
          JMP      AROUND
RENAM?:   SYSF     23               ;rename file
RNFLAG    SET      TRUE             ;only one copy
          ENDIF
AROUND:                             ;;RENAME
          ENDM
```

Figure 7.6: Macro RENAME to Rename a Disk File

loaded with the FCB address.

The second parameter, STAR, is the ASCII character to be printed on the console after each sector is written. This allows the user to follow the operation when several sectors are written. (As we learned from the operation of macro READS, printing a symbol after each sector is written greatly slows the process.) If there is no room on the disk, macro ERRORM prints the appropriate error message.

A MACRO TO CLOSE A DISK FILE

When a disk file is created, it is written sector by sector from the memory image. As each sector is written to the disk, the memory FCB is

```
WRITES    MACRO    POINTR, STAR
;;(Put current date here)
;;Inline macro to write a disk sector.
;;POINTR refers to file control block.
;;STAR is symbol to print for each sector.
;;Macros needed: SYSF, PCHAR, ERRORM
;;
          LOCAL    AROUND
          IF       NOT NUL STAR
          PCHAR    STAR
          ENDIF
          IF       NOT NUL POINTR
          LXI      D,POINTR
          ENDIF
          CALL     WRIT2?
          ORA      A              ;set flag
          IF       WRFLAG
          JNZ      NROOM?
          ELSE                    ;first time
          JZ       AROUND         ;ok
NROOM?:
          ERRORM   'No disk space', DONE
;
WRIT2?:   SYSF     21             ;write disk sector
WRFLAG    SET      TRUE           ;only one copy
          ENDIF                   ;;WRFLAG
AROUND:                           ;;WRITES
          ENDM
```

Figure 7.7: Macro WRITES to Write a Disk Sector

updated to show where the sector is located. The disk FCB, however, is not altered at this time. After the final sector has been written, you must close the file with BDOS function 16. This action will update the disk FCB from the memory FCB.

Macro CLOSE, shown in Figure 7.8, can be used to close a disk file. While this macro can be used by itself, it is also used in conjunction with macro SETUP2. In particular, if a source file name but no destination file name is given in the original command, macro CLOSE will take care of all the necessary details. For example, if the original file name is

```
COPY.ASM
```

then macro SETUP2 creates the temporary file COPY.$$$. Macro CLOSE will delete the file COPY.BAK if it exists. Then it will rename COPY.ASM to COPY.BAK. Finally, COPY.$$$ will be renamed to COPY.ASM.

Add macro CLOSE to your library. This macro references seven other macros: SYSF, ERRORM, OPEN, PRINT, MOVE, DELETE, and RENAME.

```
CLOSE     MACRO     POINTR
;;(Put current date here)
;;Inline macro to close a new file.
;;POINTR refers to file control block.
;;If file is not found, branch to DONE.
;;If S2FLAG from SETUP2 is true, check if
;;duplicate file name flag DUPL is set. Change
;;source file to BAK and new file to orig name.
;;Set S2FLAG false at beginning.
;;Usage:  CLOSE     DFCB
;;
;;Macros needed: SYSF, ERRORM, OPEN,
;; PRINT, MOVE, DELETE, RENAME
;;
          LOCAL     AROUND, CLOSE3
          IF        NOT NUL POINTR
          LXI       D,POINTR
          ENDIF
          CALL      CLOS2?
          INR       A               ;FF hex is error
          IF        NOT S2FLAG      ;SETUP2 macro
          JNZ       AROUND          ;ok
          ELSE
          JZ        CLOS3?
          LDA       DUPL            ;duplicate name?
          ORA       A
          JZ        AROUND          ;no
          MOVE      'BAK', FCB1+10H+9
          MOVE      FCB1+9, DFCB+10H+9, 3
          MOVE      FCB1, FCB1+10H, 9
```

Figure 7.8: Macro CLOSE to Close a Disk File

```
        MOVE    DFCB, DFCB+10H, 9
        DELETE  FCB1+10H        ;BAK name if any
        RENAME  FCB1            ;orig to BAK
        RENAME  DFCB            ;$$$ to orig
        MOVE    'BAK', FCB1+9   ;restore
        OPEN    FCB1
        JMP     AROUND
        ENDIF                   ;S2FLAG
        IF      NOT CLFLAG      ;one copy
CLOS3?: ERRORM  '?File not found?', DONE
CLOS2?: SYSF    16              ;close disk file
CLFLAG  SET     TRUE            ;only one copy
        ENDIF                   ;CLFLAG
AROUND:                         ;;CLOSE
        ENDM
```

Figure 7.8 (continued)

DUPLICATING A DISK FILE

We are ready to write a program for copying disk files. We have created an extensive macro library to make this task easier. This program is not in itself very useful, because the CP/M program PIP can be used for this purpose. Nevertheless, such a program will be a starting point for other useful programs, such as a program to encrypt a file.

Our COPY program will expect two parameters on the command line—a source file and a destination file. For example, the command line might look like this:

```
COPY FIRST SECOND
```

Program COPY will then generate a new disk file called SECOND that is an exact copy of an existing file called FIRST. Notice that the command line is more natural than the one used by PIP. The source file name is given first, followed by the destination file name. Furthermore, there is no equal sign between the two file names.

In Chapter 6 we saw that it is necessary to open an existing disk file before it can be accessed. We therefore will need an instruction to open the source file called FIRST.

The destination file is handled differently from the source file. The programmer must ensure that the file with the name SECOND does not exist on the disk. If it does exist, it must be erased.

If we look ahead to possible variations of our copy program, we will want to consider the possibility of a single parameter such as

```
COPY FIRST.EXT
```

In this example, the given file name is both the source file and the destination file. A temporary destination file will be created to receive the result. At the conclusion of the program, the file type of the source file will be changed to BAK, and the temporary name given to the destination file will be changed to the original file name.

Make a copy of the source program given in Figure 7.9. Give it the file name COPYS.ASM or COPYS.MAC, depending on your assembler (COPYS stands for copy sector). You might want to start with a copy of one of the programs from the last chapter, altering it to match Figure 7.9.

Most of this program consists of definitions of symbols and flags. The actual instructions and macros occupy only the last dozen or so lines of the program. Assemble the program and try it out. Use COPY to duplicate its own source program, using STAT first to ensure that there is sufficient space on the diskette. Give the command

```
COPYS COPYS.ASM CRYPT.*
```

Look at the new copy by using SHOW, which we wrote in Chapter 6, or use the CP/M command TYPE. Do not erase this copy; we will use it in the next section.

When COPYS is executed, it reads one sector (128 bytes) into memory and prints an * symbol. It then writes that sector to the new disk file and prints a # symbol. These two symbols will be printed alternately across the console, giving you a report on the progress. Alternately reading and writing a single sector is an inefficient way to make a copy, but it does have one advantage—the size of the file is not limited by the available memory space. It would be faster to read the entire source file, then write the entire new file. We will consider this method shortly.

We have incorporated the macro ABORT, so you can interrupt the copying process at any point by pressing the escape key. The new file will not be created in this case. There will be a directory entry, but it will be empty because the program did not perform the close function.

ENCRYPTING AN ASCII FILE

With a few modifications to the copy program we just wrote, we can convert it to an encrypting (coding) program. Such a program can be very useful. For example, if you have a computer in a public place, you may

```
TITLE   'Copy file sector by sector'
;
;(Put current date here)
;
;Usage: COPYS SOURCE DESTINATION
;
FALSE   EQU     0
TRUE    EQU     NOT FALSE
;
BOOT    EQU     0
BDOS    EQU     5       ;BDOS entry point
TPA     EQU     100H
;
FCB1    EQU     5CH     ;first file name
FCB2    EQU     6CH     ;second file name
DBUFF   EQU     80H     ;default buffer
;
;Set flags in main program so only one
;copy of certain subroutines will be generated.
;Place set lines before MACLIB call.
;
CIFLAG  SET     FALSE   ;input console char
CLFLAG  SET     FALSE   ;close disk file
CMFLAG  SET     FALSE   ;compare
COFLAG  SET     FALSE   ;output console char
CRFLAG  SET     FALSE   ;carr-ret/line-feed
DEFLAG  SET     FALSE   ;delete disk file
MKFLAG  SET     FALSE   ;create new disk file
MVFLAG  SET     FALSE   ;block move
OPFLAG  SET     FALSE   ;open disk file
PRFLAG  SET     FALSE   ;print console
RDFLAG  SET     FALSE   ;read disk sector
RNFLAG  SET     FALSE   ;rename disk file
S2FLAG  SET     FALSE   ;SETUP2 macro
UNFLAG  SET     FALSE   ;unprotect
WRFLAG  SET     FALSE   ;write disk sector
;end of flags
;
```

Figure 7.9: Program COPYS to Duplicate a Disk File

```
            MACLIB   CPMMAC
;
ORG         TPA
;
START:
            ENTER
            VERSN    '(current date).COPYS '
            SETUP2              ;input and output files
COPY:                           ;file 1 to file 2
            READS    FCB1,'*'   ;read a sector
            JNZ      EOFILE     ;done
            ABORT    ESC        ;quit?
            WRITES   DFCB,'#'   ;write new sector
            JMP      COPY       ;yes, next sector
EOFILE:
            CLOSE    DFCB       ;destination file
DONE:
            EXIT
;
            END      START
```

Figure 7.9 (continued)

want to ensure the privacy of certain files (such as those dealing with payroll or other personnel matters). If these files are coded, they cannot be inspected by someone who does not know how to decode them.

Use the copy of the source program we made in the previous section, and give the new copy the file name CRYPT.ASM or CRYPT.MAC. Alter the program to look like that in Figure 7.10.

Near the beginning of the instructions we add a reference to macro GFNAME. This will ask the user for a file name if none was entered on the command line. Macro SETUP2 prepares two memory FCBs using the parameters given on the command line. Then macros PRINT and READCH are used to request the encrypting key. This can be any keyboard character.

One sector of the source file is read into memory. Then each byte of the sector is coded by performing an exclusive OR with the desired key. This converts the file into an unreadable form. The advantage of the exclusive OR operation is the ease of decoding. A second exclusive OR operation, using the same coding key, returns the byte to its original form. Thus the encrypting program is also the decrypting program.

After each byte of a sector is coded (or decoded), the sector is written

```
TITLE   'Encrypt file with XOR'
;
;Feb 8.0, 1982
;
;Usage: CRYPT SOURCE DESTINATION
;
FALSE   EQU     0
TRUE    EQU     NOT FALSE
;
BOOT    EQU     0
BDOS    EQU     5       ;BDOS entry point
TPA     EQU     100H
FCB1    EQU     5CH     ;first file name
FCB2    EQU     6CH     ;second file name
DBUFF   EQU     80H     ;default buffer
;
;Set flags in main program so only one
;copy of certain subroutines will be generated.
;Place set lines before MACLIB call.
;
CIFLAG  SET     FALSE   ;input console char
CLFLAG  SET     FALSE   ;close disk file
CMFLAG  SET     FALSE   ;compare
COFLAG  SET     FALSE   ;output console char
CRFLAG  SET     FALSE   ;carr-ret/line-feed
DEFLAG  SET     FALSE   ;delete disk file
FLFLAG  SET     FALSE   ;fill characters
FNFLAG  SET     FALSE   ;read file name
MKFLAG  SET     FALSE   ;create new disk file
MVFLAG  SET     FALSE   ;block move
OPFLAG  SET     FALSE   ;open disk file
PRFLAG  SET     FALSE   ;print console
RCFLAG  SET     FALSE   ;read console
RDFLAG  SET     FALSE   ;read disk sector
RNFLAG  SET     FALSE   ;rename disk file
S2FLAG  SET     FALSE   ;SETUP2 macro
UNFLAG  SET     FALSE   ;set file attributes
WRFLAG  SET     FALSE   ;write disk sector
```

Figure 7.10: Program CRYPT to Encrypt a File with the XOR Operation

```
;end of flags
;
            MACLIB    CPMMAC
;
ORG         TPA
;
START:
            ENTER
            VERSN     '2.08.82.CRYPT '
            LDA       FCB1+1
            CPI       BLANK       ;first file name?
            JNZ       FIRN        ;yes
            GFNAME    FCB1        ;get file name
FIRN:
            SETUP2                ;input and output files
;
;get encrypting character from console
;
            PRINT     <CR,LF,' Press ESC to abort',CR,LF,LF>
            PRINT     'Input one letter for encoding key: '
            READCH                ;console char
            ANI       7FH         ;strip parity
            CPI       ESC
            JZ        DONE
            STA       KEY         ;save
            CRLF
COPY:                             ;file 1 to file 2
            READS     FCB1,'*'    ;read a sector
            JNZ       EOFILE      ;done
            ABORT     ESC         ;quit?
;
;perform XOR with key for each byte
;HL is pointer to sector buffer
;
            PUSH      H           ;save pointer
            LXI       H,DBUFF     ;disk buffer
            LDA       KEY         ;get it
            MOV       B,A         ;save in B
```

Figure 7.10 (continued)

```
        MVI     C,80H       ;sector length
CODE:
        MOV     A,M         ;get byte
        XRA     B           ;XOR with key
        MOV     M,A         ;put byte back
        INX     H           ;increment pointer
        DCR     C           ;count
        JNZ     CODE        ;keep going
        POP     H           ;restore
        WRITES  DFCB,'#'    ;write new sector
        JMP     COPY        ;next sector
EOFILE:
        CLOSE   DFCB        ;destination file
        PRINT   <CR,LF,' Delete original file? '>
        READCH
        UCASE
        CPI     'Y'
        JNZ     DONE
        DELETE  FCB1        ;gone
DONE:
        EXIT
;
KEY:    DS      1           ;encrypting key
;
        END     START
```

Figure 7.10 (continued)

to the destination file. Another sector is then read from the source file. The program continues in this way until the entire file has been coded or until the escape key is pressed, aborting the program.

If only one file name was entered at the beginning of the program, the new file is given the original file name and the file type of the source file is changed to BAK. At the conclusion of the program, the user is given the option of deleting the original file.

Encrypt a copy of the source file using the letter M. Give the coded copy the file name CRYPT.COD. The execution will be faster if the new copy is on a different drive. For example:

```
CRYPT CRYPT.ASM B:*.COD
```

Be careful not to delete the original file, although if you do, you can regenerate it by running CRYPT again and giving the same encrypting character:

```
CRYPT B:CRYPT.COD *.ASM
```

If you examine the coded file with SHOW or the CP/M TYPE command, the console screen will be filled with meaningless information. However, you can use the program DUMP, which we wrote in Chapter 6, to study the result. For example, the command DUMP B:CRYPT.COD will give you something like this:

```
Space bar for next screen, <CR> next line, <ESC> to abort
0100 39243921 28446A08 232E3F34 3D396D2B   9$9!(Dj.#.?4=9m+
0110 2421286D 3A243925 6D15021F 6A404776   $!(m:$9%m...j@Gv
0120 4047766D 0B282F6D 6D75637D 616D7C74   @Gvm.(/mmuc}am|t
0130 757F4047 76404776 6D183E2C 2A28776D   u.@Gv@Gvm.>,*(wm
0140 08030E1F 141D196D 6D1E0218 1F0E086D   .......mm......m
0150 6D09081E 1904030C 19040203 40477640   m...........@Gv@
0160 472B2C21 3E284428 3C38447D 4047393F   G+,!>(D(<8D}@G9?
0170 38284428 3C384423 22396D2B 2C213E28   8(D(<8D#"9m+,!>(
0180 40477640 472F2222 3944283C 38447D40   @Gv@G/""9D(<8D}@
```

On the other hand, if you examine the original file with the command DUMP CRYPT.ASM, you will see the following:

```
Space bar for next screen, <CR> next line, <ESC> to abort
0100 5449544C 45092745 6E637279 70742066   TITLE.'Encrypt f
0110 696C6520 77697468 20584F52 270D0A3B   ile with XOR'..;
0120 0D0A3B20 46656220 20382E30 2C203139   ..; Feb  8.0, 19
0130 38320D0A 3B0D0A3B 20557361 67653A20   82 ..;..; Usage:
0140 454E4352 59505420 20534F55 52434520   ENCRYPT  SOURCE
0150 20444553 54494E41 54494F4E 0D0A3B0D    DESTINATION..;.
0160 0A46414C 53450945 51550930 0D0A5452   .FALSE.EQU.0..TR
0170 55450945 5155094E 4F542046 414C5345   UE.EQU.NOT FALSE
0180 0D0A3B0D 0A424F4F 54094551 55093000   ..;..BOOT.EQU.0.
```

Examining the ASCII representation of the coded file, you can see that the lowercase letter m appears frequently. Remember that the uppercase letter M was used as the encrypting key. Obviously, it would not be too difficult to discover the encrypting character by studying the coded file.

If a more secure encryptation is desired, the process can be repeated using a different key. For example, encrypt the coded file a second time with the uppercase letter A. Give the command CRYPT CRYPT.COD. Look at the result with the command

```
A:DUMP CRYPT.COD
```

The result now will be as follows:

```
Space bar for next screen, <CR> next line, <ESC> to abort
0100 78657860 69052B49 626F7E75 7C782C6A   xex`i.+Ibo~u|x,j
0110 6560692C 7B657864 2C54435E 2B010637   e`i,{exd,TC^+..7
0120 0106372C 4A696E2C 2C34223C 202C3D35   ..7,Jin,,4"< ,=5
0130 343E0106 37010637 2C597F6D 6B69362C   4>..7..7,Y.mki6,
0140 49424F5E 555C582C 2C5F4359 5E4F492C   IBO^U\X,,_CY^OI,
0150 2C48495F 5845424D 58454342 01063701   ,HI_XEBMXECB..7.
0160 066A6D60 7F690569 7D79053C 0106787E   .jm`.i.i}y.<..x
0170 79690569 7D790562 63782C6A 6D607F69   yi.i}y.bcx,jm`.i
0180 01063701 066E6363 7805697D 79053C01   ..7..nccx.i}y.<.
```

This file was first coded with the letter M, then it was coded a second time with the letter A. Neither of these characters is prominent in the ASCII representation. This file must be decoded twice, once with the letter M and once with the letter A. However, it does not matter which key is given first.

When you encrypt a file by giving only the source file name, for example, CRYPT.COD, the program fully demonstrates its operation. At the end of the process there will still be only one file with this name. During operation the console will display the following lines:

```
Press ESC to abort
Input one letter for encoding key: M
*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*#*
CRYPT COD ==> CRYPT BAK
CRYPT $$$ ==> CRYPT COD
Delete original file? y
```

Both CRYPT from this section and COPY from the previous section print an interlaced sequence of * and # symbols as the file sectors are being read or written. Because the printing of these characters is very time consuming for larger files, you may want to remove the second parameter from macro READS and WRITES after you become familiar with the operation of these programs. That is, change

```
        READS   FCB1,'*'    ;read a sector
```

to

```
        READS   FCB1        ;read a sector
```

and change

```
        WRITES  DFCB,'#'    ;write new sector
```

to

```
        WRITES  DFCB        ;write new sector
```

We will now consider a more efficient way to read a disk file.

COPYING A FILE BY BUFFERING INTO MEMORY

The COPY and CRYPT programs we just wrote use macro READS to read one disk sector and macro WRITES to write one disk sector. Alternately reading and writing one sector at a time is an easy way to program disk operations, and it does not require a large amount of memory. However, a disk file can be copied more rapidly if the entire file is read into memory at one time. A new file is then written from memory all at once. The disadvantage of this technique is that very large files cannot be loaded into memory, at least not all at once. However, this limitation is not serious. Most commercial executable programs are small enough to fit into a moderately sized memory. Furthermore, it is better to limit text files to a size that will fit into memory, as this will speed up the editing process.

To enable us to copy files more efficiently, we must add two macros to our library. One will read an entire disk file into memory at once, and the other will perform the complementary operation—it will write an entire disk file from a memory image.

Reading an Entire File into Memory

Macro LDFILE, shown in Figure 7.11, is used to read a disk file into memory. Add it to your macro library. This macro has three parameters. The first parameter gives the location of the memory FCB for the file to be read. The second parameter is the pointer to the memory image of the file itself. The third parameter is the character to be displayed on the console as each sector is read.

It appears that the first parameter, FCB, is required, but in fact it is not. This parameter is simply passed along to macro READS. If the actual parameter is omitted, macro READS will assume that the DE register is already loaded with the address of the FCB.

The second parameter to macro LDFILE is required, but you can rewrite the macro to make it optional. The optional third parameter is also passed along to macro READS. If it is omitted, no character is displayed while the sectors are being read.

```
LDFILE      MACRO    FCB, POINTR, CHAR
;;(Put current date here)
;;Inline macro to load a disk file into
;;memory starting at POINTR.
```

Figure 7.11: Macro LDFILE to Read an Entire File into Memory

```
;;POINTR initially points to memory buffer.
;;Place buffer at end of program.
;;HL points to end of loaded program.
;;Optional 3rd parameter is printed after
;;each sector is read.
;;CCP area may be overlaid but
;;FDOS is protected.
;;Carry flag is set if file is too big.
;;DMA address is reset to 80H on exit.
;;Macros needed: SETDMA, READS
;;
;;Usage:  LDFILE   FCB1, DBUFFP, '*'
;;        LDFILE   FCB1, BUFFP
;;
LOAD2?:
        LHLD     POINTR
        XCHG                    ;move to DE
        SETDMA                  ;set next sector
        READS    FCB, CHAR
        JNZ      LOAD3?         ;done if nonzero
        LHLD     POINTR
        LXI      D,80H          ;one sector
        DAD      D              ;DE has pointer
        SHLD     POINTR         ;save pointer
;
;see if file is entering CCP area
;
        LDA      7              ;FDOS
        SUI      2              ;2 blocks down
        CMP      H              ;file too big?
        JNC      LOAD2?         ;no keep going
LOAD3?:                         ;done
        PUSH     PSW
        SETDMA   80H            ;reset
        POP      PSW
                                ;;LDFILE
        ENDM
```

Figure 7.11 (continued)

Macros SETDMA and READS are needed by macro LDFILE. We have learned that CP/M reads disk sectors into a memory region designated by the DMA address, and that this location is automatically reset to the value of 80 hex each time a warm start is performed. We used this location in the two previous programs. We also learned that the DMA address can be set to any desired memory location with BDOS function 26.

A program that uses macro LDFILE will set up the memory buffer at the end of the program. Macro LDFILE initially sets the DMA address to the beginning of this buffer. After each sector is read into memory, macro LDFILE advances the DMA address by 80 hex, the length of a sector. In this way, the entire file will be read sequentially into memory. At the end of the load step, macro LDFILE resets the DMA address to the usual value of 80 hex.

Most of the executable programs we have written save the incoming stack pointer and set up a new one. At the conclusion of the program, the original stack pointer is restored and a return instruction is executed. This approach is faster than performing a warm start when the program is finished. However, a different method must be used for larger programs. Large executable programs can use the memory space occupied by the console command processor (CCP). In this case, however, a warm start must be performed when the program is finished. This will reload the CCP and the BDOS. We use this technique whenever we need macro LDFILE, because it may have to overlay the CCP.

The address for the beginning of BDOS is coded at memory locations 6 and 7. For example, BDOS begins at the address 3C00 hex for a 20K-byte system; for a 64K system BDOS starts at FA00 hex. Thus, any executable program can determine the size of the CP/M that is currently being used. Macro LDFILE reads the high-order byte of the BDOS address at location 7. This value is compared to the high-order byte of the pointer as each sector of the file is read into memory. Macro LDFILE will allow the CCP to be overwritten, but it will protect the remainder of the CP/M system.

If a file is so large that is begins to overlay the BDOS, macro LDFILE will stop reading the file and set the carry flag. No error message is printed, however, so the programmer must test the state of the carry flag after the file has been loaded to see if the file is too large. We will now consider the complementary macro WRFILE.

Writing an Entire File from Memory

Macro WRFILE, shown in Figure 7.12, is similar to macro LDFILE. The three parameters are the same as those for macro LDFILE. Add this macro to your library.

```
WRFILE    MACRO    FCB, POINTR, STAR
;;(Put current date here)
;;Inline macro to write a disk file from
;;a memory image. Buffer starts at POINTR+2.
;;POINTR marks end of file.
;;Optional star symbol is printed for each sector.
;;Macros needed: WRITES, SBC, SETDMA, ERRORM
;;
          LOCAL    WRFIL?,EVEN?
          LHLD     POINTR          ;end
          XCHG                     ;to DE
          LXI      H,POINTR+2      ;start
          SHLD     POINTR          ;reset
          XCHG
          SBC      HL,DE           ;program length
          MOV      A,L
          MOV      L,H             ;just upper part
          MVI      H,0
          DAD      H               ;double=# sectors
          ORA      A               ;odd # of sectors?
          JZ       EVEN?           ;no
          INX      H
EVEN?:
          PUSH     B
          MOV      B,H
          MOV      C,L
WRFIL?:
          LHLD     POINTR
          XCHG                     ;move to DE
          SETDMA                   ;next sector
          WRITES   FCB,STAR
          LHLD     POINTR
          LXI      D,80H           ;one sector
          DAD      D               ;next location
          SHLD     POINTR
          DCX      B               ;number of sectors
          MOV      A,C
```

Figure 7.12: Macro WRFILE to Write an Entire File from Memory

```
        ORA     B
        JNZ     WRFIL?
        POP     B
                                ;;WRFILE
        ENDM
```

Figure 7.12 (continued)

After macro LDFILE has loaded a file into memory, the pointer will reference the end of the memory image of the file. Macro WRFILE begins by copying this pointer to the DE register. The pointer is then reset to the beginning of the memory image. The length of the file is computed by subtracting the address of the beginning of the file from the address at the end. Macro SBC is used for the 16-bit subtraction.

The Copy Program, Version 2

The program shown in Figure 7.13 uses macros LDFILE and WRFILE to copy disk files more rapidly. Duplicate the copy program in Figure 7.9 (COPYS), giving the new version the file name COPYB (for buffered copy). The command is as follows:

```
COPYS COPYS.ASM COPYB.*
```

Alter the new version to look like Figure 7.13. Assemble the program and execute it. Test COPYB by using it to make a copy of itself.

You will find that this version runs much faster than the previous one, which copies one sector at a time. A further increase in speed will occur if you remove the * and # symbols from macros

```
LDFILE    FCB1,BUFFP,'*'
```

and

```
WRFILE    DFCB,BUFFP,'#'
```

Macro LDFILE is programmed to terminate reading if a disk file is too large. You can test this feature in the following way. Create a very large file by giving the command

```
SAVE 220 DUMMY
```

(The information we are saving is simply the contents of memory.) Be sure that there is enough room on the disk (about 55K bytes). Try to copy this

file with the command

```
COPYB DUMMY
```

The copy program will begin to read the file, but it should terminate with the error message

```
?File too big
```

```
TITLE   'Copy file with buffer'
;
;(Put current date here)
;
;Usage: COPYB SOURCE DESTINATION
;
FALSE   EQU     0
TRUE    EQU     NOT FALSE
;
BOOT    EQU     0
BDOS    EQU     5               ;BDOS entry point
TPA     EQU     100H
;
FCB1    EQU     5CH             ;first file name
FCB2    EQU     6CH             ;second file name
DBUFF   EQU     80H             ;default buffer
;
;Set flags in main program so only one
;copy of certain subroutines will be generated.
;Place set lines before MACLIB call.
;
CIFLAG  SET     FALSE           ;input console char
CLFLAG  SET     FALSE           ;close disk file
CMFLAG  SET     FALSE           ;compare
COFLAG  SET     FALSE           ;output console char
CRFLAG  SET     FALSE           ;carr-ret/line-feed
DEFLAG  SET     FALSE           ;delete disk file
DMFLAG  SET     FALSE           ;set DMA address
MKFLAG  SET     FALSE           ;create new disk file
```

Figure 7.13: Program COPYB to Copy a Disk File by Buffering in Memory

```
MVFLAG  SET     FALSE           ;block move
OPFLAG  SET     FALSE           ;open disk file
PRFLAG  SET     FALSE           ;print console
RDFLAG  SET     FALSE           ;read disk sector
RNFLAG  SET     FALSE           ;rename disk file
S2FLAG  SET     FALSE           ;SETUP2 macro
UNFLAG  SET     FALSE           ;unprotect
WRFLAG  SET     FALSE           ;write disk sector
;end of flags
;
        MACLIB  CPMMAC
;
;
ORG     TPA
;
START:
        ENTER
        VERSN   '(current date).COPYB '
        SETUP2                  ;input and output files
        LDFILE  FCB1,BUFFP,'*'
        JNC     EOFILE          ;file ok
        ERRORM  <CR,LF,'?File too big'>
EOFILE:
        LHLD    BUFFP           ;pointer
        MVI     M,EOF           ;just in case
        ABORT   ESC
        WRFILE  DFCB,BUFFP,'#'
        CLOSE   DFCB            ;destination file
DONE:
        JMP     BOOT            ;warm start
OLDSTK: DS      2
        DS      34
STACK:
BUFFP:  DW      BUFFER
BUFFER: DS      1
;
        END     START
```

Figure 7.13 (continued)

A BUFFERED COPY PROGRAM WITH VERIFICATION

Our copy program needs two more features before we can begin to use it seriously. After we make a copy of a file, we should read back the new file to verify that it was written correctly. We should also be able to designate that the new file is write protected if the original file was.

Comparing Two Disk Files

Before we add the verification feature to the copy program, we will write another executable program. Make a duplicate of the previous program and give it the name VERIFY. Alter the text to look like Figure 7.14.

```
TITLE   'VERIFY two files'
;
;(Put current date here)
;
;Usage: VERIFY SOURCE DESTINATION
;
FALSE   EQU     0
TRUE    EQU     NOT FALSE
;
BOOT    EQU     0
BDOS    EQU     5               ;BDOS entry point
TPA     EQU     100H
;
FCB1    EQU     5CH             ;first file name
FCB2    EQU     6CH             ;second file name
DBUFF   EQU     80H             ;default buffer
;
;Set flags in main program so only one
;copy of certain subroutines will be generated.
;Place set lines before MACLIB call.
;
CIFLAG  SET     FALSE           ;input console char
CMFLAG  SET     FALSE           ;compare
COFLAG  SET     FALSE           ;output console char
CRFLAG  SET     FALSE           ;carr-ret/line-feed
```

Figure 7.14: Program VERIFY to Verify That Two Disk Files Are Identical

```
DMFLAG  SET     FALSE           ;set DMA address
MVFLAG  SET     FALSE           ;block move
OPFLAG  SET     FALSE           ;open disk file
PRFLAG  SET     FALSE           ;print console
RDFLAG  SET     FALSE           ;read disk sector
;end of flags
        MACLIB  CPMMAC
;
ORG     TPA
;
START:
        ENTER
        VERSN   '(current date).VERIFY '
        LDA     FCB2+1          ;second parameter
        CPI     BLANK
        JZ      NOSEC
        AMBIG   FCB1,FCB2
        MOVE    FCB2,DFCB,16    ;destination
        OPEN    FCB1
        OPEN    DFCB
        LDFILE  FCB1,BUFFP
        JNC     EOFILE          ;file ok
        ERRORM  <CR,LF,'?File too big'>
EOFILE:
        LHLD    BUFFP           ;pointer
        MVI     M,EOF           ;just in case
        LXI     H,BUFFER
NSECT:
        ABORT   ESC
        READS   DFCB
        ORA     A               ;zero means more
        JNZ     DONE2
        COMPAR  ,DBUFF,128      ;one sector
        JNZ     DIFFER
        LXI     D,80H
        DAD     D               ;next sector
        JMP     NSECT
DONE2:
        PRINT   <CR,LF,'Files are identical'>
```

Figure 7.14 (continued)

```
DONE:
        JMP     BOOT            ;warm start
NOSEC:
        ERRORM  <CR,LF,'?Second file omitted'>
DIFFER:
        ERRORM  <CR,LF,'?Files are different'>
;
OLDSTK: DS      2
        DS      34
STACK:
DFCB:   DS      33              ;second file
BUFFP:  DW      BUFFER
BUFFER: DS      1
;
        END     START
```

Figure 7.14 (continued)

Assemble the program and try it out. The command line looks like the one for the copy program except that both parameters are source files. For this program the order of the parameters is immaterial. Give a command in which both parameters are the same:

```
VERIFY VERIFY.ASM VERIFY.ASM
```

You should get the statement

```
Files are identical
```

Then give file names for files that are different:

```
VERIFY VERIFY.ASM VERIFY.COM
```

You will get the message

```
?Files are different
```

When this program is executed, the first file is read into memory. The second file is then read into the default buffer at 80 hex, one sector at a time. The program then compares this sector with the corresponding sector of the first file. Thus the TPA is used only by the first file.

The asterisk and question mark symbols can be used as ambiguous characters in the second file name. For example, the following command is valid:

```
VERIFY VERIFY.ASM *.BAK
```

A Macro to Protect Disk Files

You may have noticed that if our copy program is used to duplicate a write-protected file, the copy is not write protected. That is, the new file is not designated as read only. We are going to fix this problem for the next version, so that the new file will have the same protection attribute as the original file.

Macro PROTEC, given in Figure 7.15, can be used to protect a disk file using BDOS function 30. We previously wrote macro UNPROT to unprotect a disk file using the same BDOS function 30. Recall that the high-order bit of the first character of the file type specifies the protection attribute. If this bit is set, the file is protected. If this bit is reset, the file can be altered or erased. Add macro PROTEC to your library.

The Copy Program, Version 3

Our final version of the copy program will read the entire source file into memory. It will then write the new file from this memory image. The new copy is verified by reading the new file sector by sector and comparing each sector to the memory image. If a difference is found, the program

```
PROTEC      MACRO   POINTR
;;(Put current date here)
;;Inline macro to protect FCB at POINTR.
;;Macro needed: SYSF
;;
            LOCAL   AROUND,PROT2?
            LXI     D,POINTR
            LDA     POINTR+9            ;;extension
            ORI     80H                 ;;set for R/O
            STA     POINTR+9
            CALL    PROT2?
            JMP     AROUND
PROT2?:
            SYSF    30
AROUND:                                 ;;PROTEC
            ENDM
```

Figure 7.15: Macro PROTEC to Protect a Disk File

terminates and the error message

```
?Files are different
```

is displayed on the console. The console bell also sounds.

This version of the copy program also transfers the protection attribute of the source file to the destination file. The memory FCB of the source file is checked to see whether the file is protected. If it is, instructions created by macro PROTEC set the protection attribute of the new file.

Make a copy of program COPYB. Give it the name COPYV (copy with verification). Use program VERIFY to ensure that the copy is correct. Alter COPYV to look like Figure 7.16. Assemble the program and try it out.

```
TITLE   'COPY and verify file'
;
;(Put current date here)
;
;Usage: COPYV SOURCE DESTINATION
;
FALSE   EQU     0
TRUE    EQU     NOT FALSE
;
BOOT    EQU     0
BDOS    EQU     5               ;BDOS entry point
TPA     EQU     100H
;
FCB1    EQU     5CH             ;first file name
FCB2    EQU     6CH             ;second file name
DBUFF   EQU     80H             ;default buffer
BEL     EQU     7
;
;Set flags in main program so only one
;copy of certain subroutines will be generated.
;Place set lines before MACLIB call.
;
CIFLAG  SET     FALSE           ;input console char
CLFLAG  SET     FALSE           ;close disk file
```

Figure 7.16: Program COPYV to Copy Disk Files with Verification

```
CMFLAG  SET     FALSE           ;compare
COFLAG  SET     FALSE           ;output console char
CRFLAG  SET     FALSE           ;carr-ret/line-feed
DEFLAG  SET     FALSE           ;delete disk file
DMFLAG  SET     FALSE           ;set DMA address
MKFLAG  SET     FALSE           ;create new disk file
MVFLAG  SET     FALSE           ;block move
OPFLAG  SET     FALSE           ;open disk file
PRFLAG  SET     FALSE           ;print console
RDFLAG  SET     FALSE           ;read disk sector
RNFLAG  SET     FALSE           ;rename disk file
S2FLAG  SET     FALSE           ;SETUP2 macro
UNFLAG  SET     FALSE           ;unprotect
WRFLAG  SET     FALSE           ;write disk sector
;end of flags
;
        MACLIB  CPMMAC
;
ORG     TPA
;
START:
        ENTER
        VERSN   '(current date).COPYV '
        SETUP2                  ;input and output files
        LDA     FCB1+9
        ANI     80H             ;protected
        STA     PROTFL          ;protection flag
        LDFILE  FCB1,BUFFP
        JNC     EOFILE          ;file ok
        ERRORM  <CR,LF,'?File too big'>
EOFILE:
        LHLD    BUFFP           ;pointer
        MVI     M,EOF           ;just in case
        ABORT   ESC
        WRFILE  DFCB,BUFFP
        CLOSE   DFCB            ;destination file
;verify that file is identical with original
        OPEN    DFCB
        LXI     H,BUFFER
```

Figure 7.16 (continued)

```
        SETDMA  DBUFF
NSECT:
        ABORT   ESC
        READS   DFCB
        ORA     A               ;zero means more
        JNZ     DONE2
        COMPAR  ,DBUFF,128      ;one sector
        JNZ     DIFFER
        LXI     D,80H
        DAD     D               ;next sector
        JMP     NSECT
DONE2:
        LDA     PROTFL          ;protected?
        ORA     A
        JZ      DONE            ;no
        PROTEC  DFCB
DONE:
        JMP     BOOT            ;warm start
DIFFER:
        ERRORM  <BEL,'?Files are different'>
PROTFL: DS      1               ;protection flag
OLDSTK:
        DS      2
        DS      34
STACK:
BUFFP:  DW      BUFFER
BUFFER: DS      1
;
        END     START
```

Figure 7.16 (continued)

A PROGRAM TO RENAME DISK FILES

Disk files can be renamed with the CP/M built-in command REN. However, ambiguous file names are not allowed in this command. Thus, if you want to change all BASIC files to backup status, that is, if you want to change the extension from BAS to BAK, you must specifically rename each separate file.

The program shown in Figure 7.17 can be used to rename CP/M disk

files, either individually or in groups. The command line is similar to the other programs in this chapter. For example, the command

```
RENAME OLDNAME NEWNAME
```

changes the name of OLDNAME to NEWNAME. If a file with the new name already exists, the program asks for permission to delete it. Furthermore, if this file is write protected, additional permission is requested to unprotect it before deletion.

The usefulness of this program lies in its ability to rename several files with a single command. For example, the command

```
RENAME *.BAS *.BAK
```

will change the file type of all BASIC files to BAK. If you discover an error in the command, you can terminate the operation by pressing the escape key.

A single RENAME command can combine a delete operation with a renaming step. For example, if you want to delete the backup copy and rename the main copy as the backup copy, you can give the following two CP/M commands:

```
ERA MAIN.BAK
REN MAIN.BAK=MAIN.ASM
```

However, the same result can be obtained with a single command using our RENAME program. Give the command

```
RENAME MAIN.ASM *.BAK
```

Of course, RENAME will request permission to delete the program MAIN.BAK.

Because macro RENAME is used by this program, each renaming step is indicated graphically by right-pointing arrows. An open operation is performed on the original file name to ensure that the name exists. Then an open operation is performed on the new file name to see whether that name is in use. After each file is renamed, an open operation is performed to locate the next occurrence of the requested file name. This method generally works very well. However, it will fail if you decide not to rename one of a group of files. Each succeeding open command will locate the same file. As a consequence, RENAME is programmed to terminate if you decide not to delete a particular file.

A PROGRAM TO DELETE DISK FILES

The program shown in Figure 7.18 can be used to delete disk files. Files that are not write protected can be deleted with the CPM built-in command

ERA, but protected files cannot be deleted this way. DELETE can be used to delete protected files, although permission is requested for deletion. Also, the requested file name can contain asterisks and question marks, the CP/M ambiguous reference characters.

```
TITLE   'RENAME disk file with ambiguous reference'
;
;(Put current date here)
;Abort program with ESC.
;Program quits when a system file is found.
;
;Usage: RENAME OLD NEW
;       RENAME OLD.EXT *.BAK
;       RENAME OLD.EXT NEW.*
;       RENAME OLD.* NEW.*
;       RENAME *.EXT *.BAK
;
FALSE   EQU     0
TRUE    EQU     NOT FALSE
;
BOOT    EQU     0
BDOS    EQU     5                       ;BDOS entry point
TPA     EQU     100H
FCB     EQU     5CH                     ;file control block
;
FCB1    EQU     5CH                     ;first file name
FCB2    EQU     6CH                     ;second file name
DBUFF   EQU     80H                     ;default buffer
;
;Set flags in main program so only one
;copy of certain subroutines will be generated.
;Place set lines before MACLIB call.
;
CIFLAG  SET     FALSE                   ;input console char
CMFLAG  SET     FALSE                   ;compare
CRFLAG  SET     FALSE                   ;carr-ret/line-feed
COFLAG  SET     FALSE                   ;output console char
DEFLAG  SET     FALSE                   ;delete disk file
```

Figure 7.17: Program RENAME to Rename Disk Files

```
MVFLAG  SET     FALSE                   ;block move
OPFLAG  SET     FALSE                   ;open disk file
PRFLAG  SET     FALSE                   ;print console
RNFLAG  SET     FALSE                   ;rename disk file
UNFLAG  SET     FALSE                   ;set file attributes
;end of flags
;
        MACLIB  CPMMAC
;
ORG     TPA
;
START:
        ENTER
        VERSN   '(current date).RENAME '
        LDA     FCB1+1
        CPI     BLANK
        JZ      NOSOUR
        LDA     FCB2+1
        CPI     BLANK
        JZ      NODEST
        COMPAR  FCB1+1, FCB2+1,11
        JZ      SAMEN
        COMPAR  '???????????', FCB1+1
        JZ      IMPROP
        COMPAR  '???????????', FCB2+1
        JZ      IMPROP
        XRA     A                       ;zero
        STA     FIRSTF                  ;reset flag
        PRINT   <LF,'Press ESC to abort',CR,LF>
;
;save original parameters
;
        MOVE    FCB1, FCOPY, 20H
        MOVE    FCB1, OFCB, 20H
NEXTN:                                  ;next name
        OPEN    FCB1, FPASS             ;source file
        ABORT   ESC                     ;quit?
        MVI     A,0FFH
```

Figure 7.17 (continued)

```
            STA     FIRSTF                  ;multiple pass
            LDA     FCB1+10                 ;system file
            ANI     80H                     ;bit 7
            JNZ     SYSFIL                  ;skip
            UNPROT  FCB1
            MOVE    FCB1, OFCB, 12          ;original
            LDA     FCB1                    ;drive code
            STA     DFCB
;
;check for ambiguous original file name
;
            COMPAR  FCB1+1, FCOPY+1, 8
            JZ      NOQ1                    ;no
            MOVE    FCB1+1, OFCB+1, 11      ;actual name
            MOVE    FCB1+1, DFCB+1, 8       ;new primary
            MOVE    F2COPY+9, DFCB+9, 3     ;ext
            JMP     CHEK2
NOQ1:
;
;check for ambiguous original extension
;
            COMPAR  FCB1+9, FCOPY+9, 3
            JZ      NOQ3                    ;no
            MOVE    FCB1+1, OFCB+1, 11      ;actual name
            MOVE    FCB1+9, DFCB+9, 3       ;new ext
            MOVE    F2COPY+1, DFCB+1, 8     ;primary
            JMP     CHEK2
;
;check for ambiguous new name
;
NOQ3:
            AMBIG   FCB1, DFCB
CHEK2:
            OPEN    DFCB, RENAM
            CRLF
            PFNAME  DFCB
            PRINT   ' exists. Delete? '
            READCH
```

Figure 7.17 (continued)

```
        UCASE
        CPI     'Y'
        JNZ     DONE
        LDA     FCB1                    ;drive code
        STA     DFCB
        DELETE  DFCB
RENAM:
        RENAME  OFCB
        MOVE    FCOPY+1, FCB+1, 11      ;reset
        JMP     NEXTN
FPASS:
        LDA     FIRSTF                  ;get pass flag
        ORA     A                       ;first pass?
        JNZ     DONE                    ;no
        ERRORM  'File not found', DONE
NOSOUR:
        ERRORM  'No source file', DONE
NODEST:
        ERRORM  'No destination file', DONE
SAMEN:
        ERRORM  'Same name', DONE
IMPROP:
        ERRORM  'Improper name', DONE
;
FIRSTF: DB      0                       ;first pass
;
SYSFIL:                                 ;found system file
        CRLF
        PFNAME  FCB
        PRINT   ' is a system file'
DONE:
        EXIT
;
FCOPY:  DS      10H                     ;original command
F2COPY: DS      10H                     ;with second name
OFCB:   DS      10H                     ;original name
DFCB:   DS      10H                     ;new name
;
        END     START
```

Figure 7.17 (continued)

```
TITLE   'DELETE disk file with ambiguous reference'
;
;(Put current date here)
;
;Usage: DELETE NAME
;       DELETE NAME.EXT
;       DELETE NAME.*
;       DELETE *.EXT
;
FALSE   EQU     0
TRUE    EQU     NOT FALSE
;
BOOT    EQU     0
BDOS    EQU     5               ;BDOS entry point
TPA     EQU     100H
;
FCB1    EQU     5CH             ;first file name
FCB2    EQU     6CH             ;second file name
DBUFF   EQU     80H             ;default buffer
;
;Set flags in main program so only one
;copy of certain subroutines will be generated.
;Place set lines before MACLIB call.
;
CIFLAG  SET     FALSE           ;input console char
CMFLAG  SET     FALSE           ;compare
CRFLAG  SET     FALSE           ;carr-ret/line-feed
COFLAG  SET     FALSE           ;output console char
DEFLAG  SET     FALSE           ;delete disk file
MVFLAG  SET     FALSE           ;block move
OPFLAG  SET     FALSE           ;open disk file
PRFLAG  SET     FALSE           ;print console
UNFLAG  SET     FALSE           ;unprotect file
;end of flags
;
        MACLIB  CPMMAC
;
ORG     TPA
;
```

Figure 7.18: Program DELETE to Delete Disk Files

```
START:
        ENTER
        VERSN   '(current date).DELETE '
        LDA     FCB1+1
        CPI     BLANK
        JZ      NOSOUR
        PRINT   <LF,' Press ESC to abort',CR,LF>
        LDA     FCB2+1
        STA     QUERY           ;ask about delete
        COMPAR  '???????????', FCB1+1
        JNZ     ALLNAM
        PRINT   'Delete all? (Y/N) '
        READCH
        UCASE
        CPI     'Y'
        JNZ     DONE
ALLNAM:                         ;get first file name
        LXI     D,FCB1
        MVI     C,17            ;search for file name
        CALL    BDOS
        CPI     0FFH            ;found?
        JZ      NOSOUR          ;no
        CALL    GETNAM
NNAME:                          ;get next file name
        LXI     D,FCB1
        MVI     C,18
        CALL    BDOS
        CPI     0FFH            ;more?
        JZ      NNAM2           ;no
        CALL    GETNAM
        JMP     NNAME
NNAM2:
        LXI     H,FNAMES-12
        SHLD    FPNTR
NEXTN:                          ;next name
        LHLD    FPNTR           ;pointer
        LXI     D,12
        DAD     D
        SHLD    FPNTR           ;save
        MOV     A,M
```

Figure 7.18 (continued)

```
        CPI     BLANK
        JZ      DONE
        MOVE    , FCB1, 12
        OPEN    FCB1            ;source file
        ABORT   ESC             ;quit?
        LDA     QUERY           ;ask
        CPI     'Q'
        JNZ     NOASK
        PRINT   <CR,LF,' Delete '>
        PFNAME  FCB1
        PCHAR   '?'
        PCHAR   BLANK
        READCH
        UCASE
        CPI     'Y'
        JNZ     NEXTN
NOASK:
        DELETE  FCB1, NEXTN
        CRLF
        PFNAME  FCB1
        PRINT   ' deleted'
        JMP     NEXTN
GETNAM:                         ;copy name to work area
        RRC                     ;3 bits right = 5 left
        RRC                     ;0 = 0, 1 = 20H
        RRC                     ;2 = 40H, 3 = 60H
        ANI     60H             ;mask
        MOV     E,A
        MVI     D,0
        LXI     H,DBUFF
        DAD     D
        XCHG
        LHLD    FPNTR           ;destination
        XCHG                    ;to DE
        MOVE    , , 12
        LXI     H,12
        DAD     D
        SHLD    FPNTR           ;next name
        MVI     M,BLANK         ;mark end
        RET
```

Figure 7.18 (continued)

```
NOSOUR:
        ERRORM  'No source file', DONE
;
FPNTR:  DW      FNAMES          ;name pointer
QUERY:  DS      1               ;if Q, ask before delete
;
DONE:
        EXIT
;
FNAMES: DS      1               ;stack of file names
;
        END     START
```

Figure 7.18 (continued)

If the letter Q (for query) is given as a second parameter, DELETE requests permission to delete each file name, whether or not it is write protected. This is particularly useful in deleting improper directory entries. Sometimes, for example, a file name contains a nonprinting character or a lowercase character. In this case, the file cannot be specifically deleted with the CP/M command ERA. The troublesome file can be deleted by giving the command

```
DELETE *.* Q
```

This is a command to delete all files on the disk, but only with permission. You will then be presented every file on the disk, one at a time, whether write protected or not. If you answer each case with any character besides a Y, the particular file will not be deleted. The next file name will then appear. If you answer Y and the file is protected, you will be asked permission to unprotect the file.

In this program we use BDOS function 17 to find the first file and BDOS function 18 to find subsequent files. When using function 18, all of the file names must be copied initially into a buffer area. Then the program can work with each file name, one at a time.

SAVING THE MEMORY CACHE ON DISK

In Chapter 3 we altered the BIOS so that printer output could be saved in a memory cache. We then moved the resulting information down to the TPA and created a disk file with the built-in SAVE command. We will now write a program to make this task easier. The program shown in Figure 7.19 can write the information contained in the memory cache

directly to a disk file. The main part of the source program is very short. It uses the macros ENTER, VERSN, and EXIT in the usual way. In addition, macro GFNAME is supplied so that a file name can be entered after program execution has begun.

Recall that we set up two pointers at the beginning of the memory cache. The first points to the beginning of the text and the second refers to the end of the text. The first pointer is F000 hex. Macro WRFILE directly writes the disk file from the memory buffer. The command

```
CACHE (file name)
```

will create a disk file with the requested file name using the information contained in the memory cache.

```
TITLE   'CACHE to save memory on disk'
;
;(Put current date here)
;
;Usage: CACHE DISKFILE
;
;
FALSE   EQU     0
TRUE    EQU     NOT FALSE
;
MPOINT  EQU     0F000H          ;main pointer
MMAX    EQU     MPOINT+2        ;end of text
MBUFF   EQU     MPOINT+2        ;buffer
;
BOOT    EQU     0               ;system reboot
BDOS    EQU     5               ;BDOS entry point
FCB1    EQU     5CH             ;input FCB
DBUFF   EQU     80H             ;default buffer
TPA     EQU     100H            ;transient program area
;
;Set flags in main program so only one
;copy of certain subroutines will be generated.
;Place set lines before MACLIB call.
;
CIFLAG  SET     FALSE           ;input console char
CLFLAG  SET     FALSE           ;close disk file
```

Figure 7.19: Progam CACHE to Create a Disk File from the Memory Cache

```
COFLAG  SET     FALSE           ;output console char
CRFLAG  SET     FALSE           ;carr-ret/line-feed
DEFLAG  SET     FALSE           ;delete disk file
DMFLAG  SET     FALSE           ;set DMA
FLFLAG  SET     FALSE           ;fill characters
FNFLAG  SET     FALSE           ;get file name
MKFLAG  SET     FALSE           ;create new disk file
MVFLAG  SET     FALSE           ;block move
OPFLAG  SET     FALSE           ;open disk file
PRFLAG  SET     FALSE           ;print console buffer
RCFLAG  SET     FALSE           ;read console buffer
RNFLAG  SET     FALSE           ;rename disk file
S2FLAG  SET     FALSE           ;SETUP2 macro not used
UNFLAG  SET     FALSE           ;unprotect
WRFLAG  SET     FALSE           ;write disk file
;end of flags
;
        MACLIB  CPMMAC
;
ORG     TPA
;
START:
        ENTER
        VERSN   '(current date).CACHE '
        LDA     FCB1+1
        CPI     BLANK           ;file name?
        JNZ     OP3             ;yes
        GFNAME  FCB1            ;get file name
OP3:
        DELETE  FCB1            ;existing name
        MAKE    FCB1            ;new one
;
;make disk file starting at MMAX
;
        WRFILE  FCB1, MMAX
        CLOSE   FCB1
DONE:
        EXIT
;
        END     START
```

Figure 7.19 (continued)

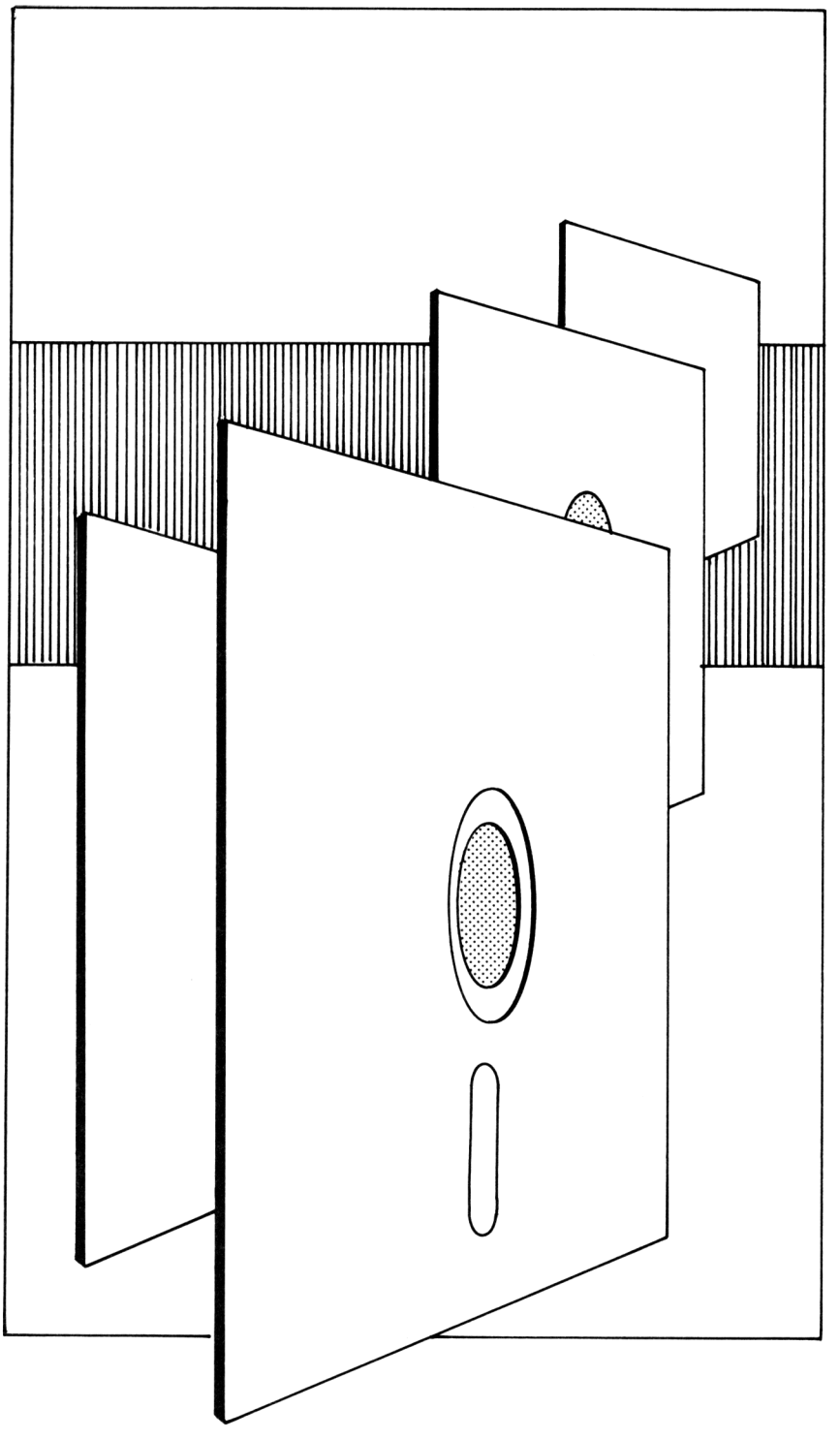
SUMMARY

In this chapter we added several significant macros to our library: MAKE, UNPROTECT, PFNAME, DELETE, SETUP2, RENAME, CLOSE, WRITES, LDFILE, WRFILE, and PROTEC. We then wrote several executable programs to copy, code, verify, rename, and delete disk files.

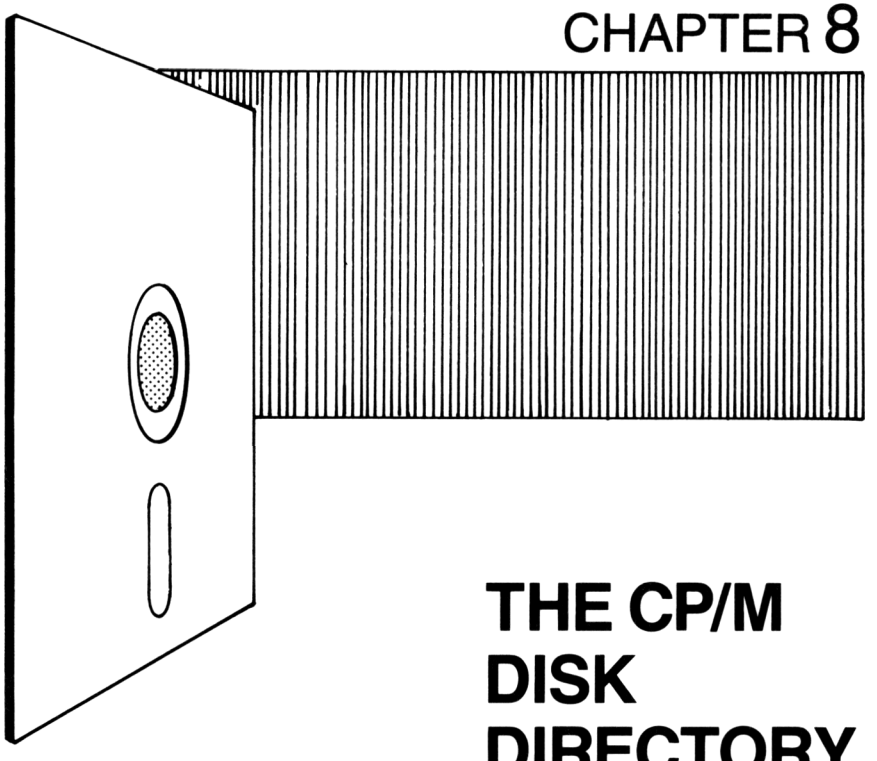
Your macro library directory should now look like this:

```
;;Macros in this library                          Flags
;;ABORT    MACRO   CHAR                  CIFLAG, COFLAG
;;AMBIG    MACRO   OLD, NEW              (none)
;;CLOSE    MACRO   POINTR                CLFLAG, COFLAG, CRFLAG
;;                                       PRFLAG, OPFLAG, MVFLAG,
;;                                       DEFLAG, CIFLAG, UNFLAG,
;;                                       RNFLAG, S2FLAG
;;COMPAR   MACRO   FIRST, SECOND, BYTES  CMFLAG
;;COMPRA   MACRO   FIRST, SECOND, BYTES  CMFLAG
;;CPMVER   MACRO                         (none)
;;CRLF     MACRO                         CRFLAG, COFLAG
;;DELETE   MACRO   POINTR, WHERE         DEFLAG, CIFLAG
;;                                       COFLAG, PRFLAG, UNFLAG
;;ENTER    MACRO                         (none)
;;ERRORM   MACRO   TEXT, WHERE           COFLAG, CRFLAG, PRFLAG
;;EXIT     MACRO   SPACE?                (none)
;;FILL     MACRO   ADDR, BYTES, CHAR     FLFLAG
;;GFNAME   MACRO   FCB                   FNFLAG, FLFLAG, RCFLAG
;;                                       COFLAG, CRFLAG, PRFLAG
;;HEXHL    MACRO   POINTR                HXFLAG, RCFLAG
;;LCHAR    MACRO   PAR                   LOFLAG
;;LDFILE   MACRO   FCB, POINTR, CHAR     COFALG, DMFLAG
;;                                       RDFLAG
;;MAKE     MACRO   POINTR                MKFLAG, COFLAG, CRFLAG,
;;                                       PRFLAG
;;MOVE     MACRO   FROM, TO, BYTES       MVFLAG
;;OPEN     MACRO   POINTR, WHERE         OPFLAG, COFLAG, PRFLAG
;;                                       CRFLAG
;;OUTHEX   MACRO   REG                   CXFLAG, COFLAG
;;PCHAR    MACRO   PAR                   COFLAG
;;PFNAME   MACRO   FCB                   COFLAG, PRFLAG
;;PRINT    MACRO   TEXT, BYTES           PRFLAG, COFLAG
;;PROTEC   MACRO   POINTR                (none)
;;READB    MACRO   BUFFR                 RCFLAG
```

```
;;READCH     MACRO     REG             CIFLAG, COFLAG
;;READS      MACRO     POINTR, STAR    RDFLAG, COFLAG
;;RENAME     MACRO     POINTR          RNFLAG, COFLAG
;;                                     PRFLAG, CRFLAG
;;SBC        MACRO                     (none)
;;SETDMA     MACRO     POINTR          DMFLAG
;;SETUP2     MACRO                     S2FLAG, CIFLAG, COFLAG,
;;                                     CRFLAG, CMFLAG, DEFLAG,
;;                                     MKFLAG, MVFLAG, OPFLAG,
;;                                     PRFLAG, UNFLAG
;;SYSF       MACRO     FUNC, AE        (none)
;;UCASE      MACRO     REG             (none)
;;UNPROT     MACRO     POINTR          UNFLAG
;;UPPER      MACRO     REG             (none)
;;VERSN      MACRO     NUM             (none)
;;WRITES     MACRO     POINTR, STAR    WRFLAG, COFLAG
;;                                     PRFLAG
;;WRFILE     MACRO     FCB, POINTR     COFLAG, CRFLAG
;;                                     DMFLAG, WRFLAG
```



CHAPTER 8



THE CP/M DISK DIRECTORY

INTRODUCTION

In Chapter 6 we briefly looked at the organization of the CP/M disk directory. We will now study the directory in more detail by developing a program that displays several directory functions. These include a display of the disk parameters, an extended listing of the directory with its block numbers, and the block allocation map.

THE DISK PARAMETERS

CP/M was originally written for use with the standard IBM 8-inch floppy disk. This disk's format is single density, single sided, and soft sectored. There are 77 tracks with 26 sectors per track. Each sector contains 128 data bytes. The block size, the smallest amount of data that can be allocated on the disk, is 1024 (1K) bytes. These disk parameters are coded into the BDOS area of CP/M version 1.4. Consequently, it is difficult to alter this version of CP/M to incorporate disks with different parameters.

CP/M version 2 is organized differently. The disk parameters are written into the BIOS rather than the BDOS. Consequently, it is relatively easy to alter this version of CP/M to accommodate any type of disk. The characteristics for each different disk drive are located in an area of memory known as the disk parameter block (DPB). This region can be located with BDOS function 31. Let us investigate this area.

Go to disk A and reset the disk drives by typing control-C. Execute the debugger DDT (or SID) and write the following short program with the A command:

```
A100
0100 MVI C,1F
0102 CALL 5
0105 RST 7
```

The first instruction of this program loads the C register with 31 (1F hex). This is the BDOS function that locates the disk parameters. The second instruction calls BDOS and the third instruction returns to the debugger. Execute this program with the command G100. Now display the registers with the debugger X command. The result might look like this:

```
-Z-E- A=8A B=D400 D=0000 H=D48A S=0100 P=0105 RST 07
```

The first part of this line gives the state of the CPU flags. In this example, the zero flag (Z) and the parity flag (E for even parity) are set. The three minus signs indicate that the other flags (carry, half-carry, and sign) are reset. The next six items give the state of the CPU registers, including the stack pointer (S) and program counter (P). The final item is the last instruction that was executed prior to returning to the debugger.

We are interested in the value contained in the HL register, because it contains a pointer to the beginning of the disk parameter block. In this example, the disk parameters begin at address D48A hex for the currently logged-in drive. However, before we look at these parameters, let us consider

other disk formats. You may have more than one kind of disk drive. For example, one drive might be single sided and another might be double sided. Another possibility is that a double-density drive might be able to read single-density format as well as double-density format. In either of these cases, there will be a different set of disk parameters for each disk format.

Let us assume that drive A reads double-density format and drive B can read either double-density or single-density format. Put a single-density diskette into drive B. Write the following program at 200 hex with the A command:

```
A200
0200 MVI E,1
0202 MVI C,E
0204 CALL 5
0207 RST 7
```

This program performs BDOS function 14 (0E hex), which changes the default drive. Register E refers to the new drive. A value of 0 indicates drive A, 1 refers to drive B, and so on. In this case we load register E with the value of 1 because we are going to change the default drive to B. Register C is given the value of 0E hex. The third instruction calls BDOS, and the final instruction returns to the debugger.

Execute this program with the command G200. The head of disk drive B should load and the activity light on the front of the drive should turn on. Rerun the first program with the command G100. Then display the registers with the X command. The result might look like this:

```
-Z-E- A=7B B=D43F D=003F H=D47B S=0100 P=0105 RST 07
```

This time, the HL register refers to a different memory location. That is, a different set of disk parameters is referenced this time. Notice that the address of the first DPB is exactly 15 bytes larger than the second DPB. The DPBs can be placed anywhere in BIOS, but it is logical to group them together. Because each DPB is 15 bytes long, successive addresses for adjacent DPBs will usually differ by 15 bytes.

We will now study the DPB area. We found a DPB at address D47B hex and another at address D48A hex. However, there might be additional DPBs for other disk formats. These will usually be given in the same general area. Therefore, we will start the display a few lines prior to the DPB area we found. Give the debugger command

```
DD450
```

The resulting output might look like Figure 8.1. There are actually five different DPBs given in this figure. The boldface numbers designate the first byte of each DPB. We will study them in more detail shortly. But first we will consider the information given in the DPB.

```
D450: 00 00 00 00 00 00 DC 5D D4 08 DD F2 DC 28 00 04 .......]....(..
D460: 0F 01 A9 00 3F 00 80 00 10 00 02 00 14 00 03 07 ....?...........
D470: 00 4F 00 3F 00 C0 00 10 00 03 00 28 00 03 07 00 .O.?.......(....
D480: A4 00 3F 00 C0 00 10 00 02 00 28 00 04 0F 01 51 ..?.......(....Q
D490: 00 3F 00 80 00 10 00 02 00 1A 00 03 07 00 F2 00 .?..............
D4A0: 3F 00 C0 00 10 00 02 00                         ?.......
```

Figure 8.1: Five Different Disk Parameter Blocks

THE DISK PARAMETER BLOCK

Nine items describing the format of the disk are specified in the disk parameter block. Some of the entries are one byte long; others are two bytes long. Table 8.1 summarizes these items. The value given in the offset column is the address relative to the beginning of the DPB. That is, the address of each item is the value of the offset plus the DPB address contained in the HL register after BDOS call 31.

Table 8.1: Items Specified in the Disk Parameter Block

Offset	Symbol	Bytes	Explanation
0	SPT	2	Logical sectors per track
2	BSH	1	Block shift
3	BLM	1	Block mask
4	EXM	1	Extent mask
5	DSM	2	Maximum number of blocks
7	DRM	2	Maximum directory entries
9	ALO,1	2	Directory allocation
11	CKS	2	Directory sectors to check
13	OFF	2	Track offset

Let us consider the disk parameters in more detail. The first entry, SPT, gives the number of logical 128-byte sectors per track. It is a two-byte value, with the low-order byte stored first. Many disk controllers are

programmed for sectors that are larger than 128 bytes. For example, the North Star double-density format uses 512-byte sectors. There are 10 of these sectors per track. The SPT value for this disk, however, is 40 rather than 10. That is, the number of logical, 128-byte sectors per track is given.

Both the second and the third entries, the BSH and the BLM, are functions of the block size. Remember, this is the minimum amount of information that can be referenced on the disk. The BSH is the logarithm, base 2, of the number of 128-byte sectors in the block. For example, the standard IBM single-density, 8-inch format uses eight sectors per block. Consequently, the BSH is 3 (since $2^3 = 8$). The BLM is one less than the number of 128-byte sectors per block. The possible values for BSH and BLM are summarized in Table 8.2.

Table 8.2: Possible Values for BSH and BLM

Block size	Number of sectors	BSH	BLM
1K	8	3	7
2K	16	4	15
4K	32	5	31
8K	64	6	63
16K	128	7	127

We have seen that disk files are described by a 32-byte FCB. The first 16 bytes contain the name and size of the file. The remaining 16 bytes give the location of each block of sectors on the disk. The single-density, 8-inch disk has a 1K block size. Each FCB on this disk can reference a maximum of 16K bytes of data, because each pointer is one byte in size.

With double-density disks, the situation is different. We have seen that a double-density disk can have a block size of 2K, 4K, 8K, or 16K bytes. Consider, for example, a disk with a 2K block size. The 16 pointers can now reference 32K bytes. Because CP/M is programmed to handle a 16K extent, each FCB is divided into two 16K byte extents. In a similar way, there can be four 16K extents in one FCB when the block size is 4K bytes. The terminology is sometimes confusing when an FCB is referred to as an extent. For example, we may read about a format that has four logical extents in each physical extent. The writer means that there can be 64K bytes in each FCB.

The situation is further complicated if a disk has more than 255 blocks. In this case the pointers are two bytes in length. Consequently, there can

be no more than eight pointers in an FCB. A disk with a 2K block size and two-byte pointers can only reference 16K bytes in each FCB.

The many possible formats are decoded with the help of the fourth item in the disk parameter block. This is the extent mask, EXM, a one-byte value. This entry is a function of both the block size and the total number of blocks on the disk. It is one less than the maximum number of extents that can fit into each FCB. Table 8.3 shows this relationship. Small disks have less than 256 blocks; large disks have more.

Table 8.3: Possible Values for EXM

	Extent mask	
Block size	Small disk	Large disk
1K	0	--
2K	1	0
4K	3	1
8K	7	3
16K	15	7

The fifth entry gives the largest block number on the disk. It is identified by the symbol DSM. The two-byte value is stored with the low byte first. Because block numbers begin with zero, the actual number of blocks is one larger than the value given as the DSM.

The sixth entry, DRM, has a value that is one smaller than the maximum number of directory entries. It is a two-byte value that is stored with the low byte first. Directory entries are 32 bytes long. Consequently, there are four directory entries for each logical 128-byte sector.

The CP/M directory occupies the first one or more data blocks on the disk. Consequently, these blocks must always be allocated so that data are not accidentally written onto them, destroying the directory. The seventh entry is used for this purpose. The two bytes are considered together as a 16-bit map. Starting at the left side, each bit that is set to 1 reserves one block for the directory. The binary representation in Table 8.4 shows the directory allocation.

When a diskette is removed from the drive and replaced by another, it is necessary to perform a warm start before data can be written on the new diskette. Whenever a write operation is requested, CP/M checks the directory to see if the diskette has been changed. The eighth entry in the DPB specifies the number of directory sectors that should be

Table 8.4: 16-Bit Map Determining Directory Allocation

Number of directory blocks	Binary value AL0	Binary value AL1	Hex value AL0	Hex value AL1
1	10000000	00000000	80	0
2	11000000	00000000	C0	0
3	11100000	00000000	E0	0
4	11110000	00000000	F0	0

checked prior to each write operation.

For floppy disks or other removable media, the CKS will be the number of directory entries, DRM plus one, divided by four (the number of entries per sector). If the disk medium cannot be changed, as a hard disk cannot, then there is no need to make such a check. In that case, the value is set to 0, greatly speeding up the warm-start operation.

The ninth and last entry in the DPB is the track offset. This two-byte value is added to the track number requested by BDOS (the logical track number) to obtain the actual (physical) track number. This parameter can be used to partition one large disk into several logical disks. Each logical disk will have a different track offset. For example, suppose one large disk is partitioned into logical disks A, B, and C. The offsets could be 0, 100, and 200 for drives A, B, and C.

The DPB for a standard 8-inch, single-density floppy disk is given in Table 8.5.

Table 8.5: The DBP for a Standard 8-Inch Floppy Disk

Address	Symbol	Hex	Decimal	Meaning
D499	SPT	1A	26	Logical sectors per track
D49B	BSH	3	3	Block shift
D49C	BLM	7	7	Block mask
D49D	EXM	0	0	Extent mask
D49E	DSM	F2	242	Number of Blocks − 1 (243 actual)
D4AO	DRM	3F	63	Directory entries − 1 (64 actual)
D4A2	AL0	C0		Directory allocation (11000000)
D4A4	CKS	10	16	Directory sectors to check
D4A6	OFF	2	2	Track offset

Consider the DPB for drive A starting at address D48A (line 4 of Figure 8.1). This DPB describes a double-density, 5-inch drive that has 40 sectors per track (28 hex). There are 82 blocks (51 hex + 1), each with a size of 2K bytes (BLM is 15). There are 64 directory entries (3F hex + 1), so one block is reserved for the directory (AL1 is 80 hex or 1000 0000 binary). The track offset is 2.

VIEWING THE DISK PARAMETERS

Before we can write an executable program for displaying the disk parameters, we must add five new macros that will make it easier to program displays of binary numbers, 16-bit base conversions, and multiplication and division.

A Macro to Display a Binary Number in Binary

All information, whether alphanumeric characters, decimal numbers, or hexadecimal numbers, is stored in a computer as a sequence of bits—binary zero or binary one. We have already written routines to convert binary numbers to ASCII and hexadecimal. Sometimes, however, we want to consider the bits themselves. To represent this pattern for a byte, we must display a sequence of eight ASCII zeros and ones. The routine that performs this task is called a binary to ASCII binary program.

In Chapter 3 we used this routine to determine whether our printer incorporates a DTR bit. Let us now use that routine in the form of a macro. Copy macro BINBIN, shown in Figure 8.2, into your macro library. Notice that it uses the flag BNFLAG.

A Macro to Display a 16-Bit Binary Number in Decimal

Sometimes we need to determine the decimal equivalent of a binary number. The largest 8-bit number is 255. Therefore, in converting an 8-bit binary number to decimal, we must consider three powers of ten—10^0, 10^1, and 10^2. But we are going to convert a 16-bit number to decimal, and in this case the largest number is 65,535. We must therefore consider five powers of ten—10^0, 10^1, 10^2, 10^3, and 10^4.

The algorithm we use subtracts powers of ten from the original number until the result becomes negative. Then we add back the most recent term. The net number of subtractions is the decimal power. We continue in this way with 1000, 100, and 10.

```
BINBIN      MACRO
;;(Put current date here)
;;Inline macro to convert binary number in A
;;to a string of ASCII-coded binary characters.
;;
            LOCAL   BIT2,AROUND
            CALL    BINB2?
            IF      NOT BNFLAG
            JMP     AROUND
BINB2?:
            PUSH    B
            MOV     C,A
            MVI     B,8
BIT2:
            MOV     A,C
            ADD     A               ;;set carry
            MOV     C,A
            MVI     A,'0'/2
            ADC     A
            PCHAR
            DCR     B
            JNZ     BIT2
            POP     H
            RET
BNFLAG      SET     TRUE
            ENDIF
AROUND:
                                    ;;BINBIN
            ENDM
```

Figure 8.2: Macro BINBIN to Display a Binary Number as a Sequence of ASCII Zeros and Ones

We have seen that the 8080 CPU cannot directly perform a 16-bit subtraction. We therefore wrote macro SBC for this purpose. We also noticed that we could subtract one number from another by adding the two's complement. Our algorithm uses this technique. We begin by repeatedly adding −10,000, the two's complement of 10,000. When the result becomes negative, we add back that last subtraction by subtracting the two's complement. We use macro SBC for this purpose.

A second complication is the matter of leading zeros. If a number is less than 10,000, the left digit will be 0. If the number is less than 1000, there will be another 0 in the next position. However, it is customary to omit leading zeros in decimal numbers, so we will delete the leading zeros from our resulting decimal number.

Macro HLDEC, shown in Figure 8.3, converts a 16-bit binary number in HL to a string of ASCII-coded decimal digits and displays the result on the console. Add this macro to your library.

A Macro to Display a 16-Bit Binary Number in Hexadecimal

In Chapter 5 we wrote macro OUTHEX to convert an 8-bit binary number to two hexadecimal characters and display them on the console. For the programs in this chapter we will need to convert a 16-bit binary number in HL to hexadecimal characters. We will use macro OUTHEX for this purpose. If the value in HL is larger than 255, we will reference

```
HLDEC       MACRO
;;(Put current date here)
;;Inline macro to print HL as decimal.
;;Macros needed: SBC, PCHAR
;;
            LOCAL   AROUND,SUBTR,SUBT2,NZERO
            CALL    HLDC2?
            IF      NOT DEFLAG
            JMP     AROUND
HLDC2?:
            PUSH    H
            PUSH    D
            PUSH    B
            MVI     B,0             ;;leading-zero flag
            LXI     D,-10000        ;two's complement
            CALL    SUBTR           ;ten thousands
            LXI     D,-1000
            CALL    SUBTR           ;thousands
            LXI     D,-100
            CALL    SUBTR           ;;hundreds
```

Figure 8.3: Macro HLDEC to Display a 16-Bit Binary Number in Decimal

```
                LXI     D,-10
                CALL    SUBTR           ;;tens
                MOV     A,L
                ADI     '0'             ;;ASCII bias
                PCHAR
                POP     B
                POP     D
                POP     H
                RET
;;
;;subtract power of ten and count
;;
SUBTR:          MVI     C,'0'-1         ;;ASCII count
SUBT2:          INR     C
                DAD     D               ;;add neg number
                JC      SUBT2           ;;keep going
;;one too many, add one back
;;by subtracting complement
                SBC     HL,DE
                MOV     A,C             ;;get count
;;check for zero
                CPI     '1'             ;;< 1?
                JNC     NZERO           ;;no
                MOV     A,B             ;;check zero flag
                ORA     A               ;;set?
                MOV     A,C             ;;restore
                RZ                      ;;skip leading 0
                PCHAR
                RET
;;set flag for nonzero character
NZERO:
                MVI     B,0FFH
                PCHAR
                RET
DEFLAG          SET     TRUE
                ENDIF
AROUND:                                 ;;HLDEC
                ENDM
```

Figure 8.3 (continued)

macro OUTHEX twice. This will produce four ASCII characters. If the value in HL is less than 256, the value in H is 0. In this case, we will reference macro OUTHEX only once. Macro OUTHL is shown in Figure 8.4. Add it to your macro library.

A Macro to Multiply a 16-Bit Number by a Power of 2

The 8080 and Z80 CPUs have instructions for addition and subtraction, but they do not have instructions for multiplication and division. We will now write a macro to multiply a 16-bit number in HL by a power of 2, using addition and rotation. Restricting the multiplier to a power of 2 greatly simplifies the algorithm without limiting our applications, because our applications always need a multiplier of this type.

We consider two special cases at the beginning of the macro. If the multiplier is 0, the result in HL is set to 0. If the multiplier is 1, the original value in HL is returned. We place other multipliers into register B and then add HL to itself. This doubles the original multiplicand. We then rotate the multiplier in B and check the carry flag. If the carry flag is set, the multiplier is 2 and the task is finished. However, if the carry flag is reset, the original multiplier was larger than 2. We continue adding HL to itself and rotating B to the right into the carry flag.

Add macro MULT, shown in Figure 8.5, to your macro library. This macro has one optional parameter—the multiplier. If the parameter is omitted in the reference, it is assumed that the multiplier is already loaded in the accumulator.

A Macro to Divide a 16-Bit Number by a Power of 2

The complement of the previous macro is a routine to divide a 16-bit number in HL by a power of 2. When we double the value in HL by adding it to itself, the result is the same as shifting the double register to the left. Division is accomplished by shifting to the right. However, there is no 16-bit shift or rotation instruction, so we must perform two 8-bit rotations instead.

Macro DIVIDE is shown in Figure 8.6; add it to your library. There is one optional parameter, the divisor. If it is omitted from the macro reference, a value of 2 is assumed. Division by zero is undefined, but this macro will leave the dividend unchanged in this case. The result is also unchanged if the divisor is 1.

Now that we have added the necessary macros, we can write a program that will display the disk parameters.

```
OUTHL     MACRO
;;(Put current date here)
;;Inline macro to display HL in hex.
;;Macro needed: OUTHEX
;;
          LOCAL     OVER
          MOV       A,H
          ORA       A
          JZ        OVER
          OUTHEX    H
OVER:
          OUTHEX    L
                              ;;OUTHL
          ENDM
```

Figure 8.4: Macro OUTHL to Display a 16-Bit Binary Number in Hexadecimal

```
MULT          MACRO    TIMES
;;(Put current date here)
;;Inline macro to multiply value in HL by TIMES.
;;Parameter should be a power of 2.
;;0 and 1 are valid operands.
;;Parameter is omitted when A has multiplier.
;;
              LOCAL    LOOP, AROUND, NOTZ
              PUSH     B
              IF       NUL TIMES
              MOV      B,A
              ELSE
              MVI      B,TIMES
              ENDIF
              CALL     MULT2?
              POP      B
              IF       NOT MLFLAG
              JMP      AROUND
MULT2?:
```

Figure 8.5: Macro MULT to Multiply a 16-Bit Number in HL by a Power of 2

```
        MOV     A,B
        ORA     A               ;zero
        JNZ     NOTZ            ;no
        MOV     L,A
        MOV     H,A             ;HL=0
        RET
NOTZ:
        RAR
        RC                      ;times 1
        MOV     B,A
LOOP:
        DAD     H               ;times 2
        MOV     A,B
        RAR
        MOV     B,A
        JNC     LOOP
        RET
MLFLAG  SET     TRUE            ;;one copy
        ENDIF
AROUND:                         ;;MULT
        ENDM
```

Figure 8.5 (continued)

```
DIVIDE      MACRO   DENOM
;;(Put current date here)
;;Inline macro to divide HL register by DENOM.
;;Denom should be power of 2 (2, 4, 8, 16).
;;HL unaltered if DENOM is 0 or 1.
;;
            LOCAL   AROUND, SHFTR?, DIV3?
            PUSH    B
            IF      NUL DENOM
            MVI     B,2             ;default
            ELSE
            MVI     B,DENOM
            ENDIF
```

Figure 8.6: Macro DIVIDE to Divide a 16-Bit Number in HL by a Power of 2

```
        CALL    DIV2?
        POP     B
        IF      NOT DVFLAG
        JMP     AROUND
DIV2?:
        MOV     A,B
        ORA     A               ;clear carry
        RZ                      ;divide by zero?
        RAR
        RC                      ;divide by 1?
        MOV     B,A
DIV3?:
        CALL    SHFTR?          ;shift HL right
        MOV     A,B             ;get divisor
        RAR
        MOV     B,A
        JNC     DIV3?
        RET
SHFTR?:                         ;16-bit shift right
        XRA     A
        MOV     A,H
        RAR
        MOV     H,A
        MOV     A,L
        RAR
        MOV     L,A
        RET
DVFLAG  SET     TRUE            ;;one copy
        ENDIF
AROUND:                         ;;DIVIDE
        ENDM
```

Figure 8.6 (continued)

A Program to Display the Disk Parameters

The program shown in Figure 8.7 can be used to determine the disk parameters for any CP/M disk. The CP/M version must be 2.0 or greater for this program to run. The program begins in the usual way with macros ENTER and VERSN. Then macro CPMVER is used to determine the

CP/M version. If the version is less than 2, the program is terminated with the appropriate error message.

The memory FCB at 5C hex is checked next to see whether a disk drive was specified on the command line. If a specific drive was indicated, subroutine SETDSK is called. This subroutine selects the desired disk with BDOS function 14. If no disk drive was specified, the default drive is used. Subroutine SETDSK concludes in an interesting way—with a jump to BDOS rather than the usual call. The more obvious construction

```
CALL    BDOS
RET
```

performs the same task. However, it requires more code and more stack space than the instruction

```
JMP     BDOS
```

The next step is to determine the address of the disk parameter block using BDOS function 31. The disk parameters are then moved to the end of the program so they can be altered slightly. For example, the greatest block number (DSM) is incremented so that it becomes the total number of blocks. The number of directory entries (DRM) is incremented and then divided by four to find the corresponding number of directory sectors. This is saved as DIRMAX.

The allocation bytes (AL0 and AL1) are interchanged so that the low byte will be in the H register after an LHLD operation. Repeated DAD H instructions will then shift the HL register left into the carry flag. This flag will be set each time there is a corresponding bit set in AL0 and AL1. Remember, each bit corresponds to one reserved directory block. The number of reserved directory blocks is determined in this way. The result is subtracted from the total number of blocks to find the number of data blocks. This is stored as NETBL. The number of directory blocks is converted from binary to ASCII and saved as ALLOCA. This value will be used later.

Type in the program shown in Figure 8.7 and assemble it. The macro library we have developed is needed in the program. First try this program on the default drive with the command

```
DIREC
```

Then try it on another drive by giving a parameter:

```
DIREC B:
```

The second example will display the disk parameters for drive B.

```
TITLE   'DIREC, directory utility'
;
;(Put current date here)
;
;
FALSE   EQU     0
TRUE    EQU     NOT FALSE
;
BOOT    EQU     0
BDOS    EQU     5               ;BDOS entry point
TPA     EQU     100H
FCB     EQU     5CH             ;file control block
FCB1    EQU     5CH             ;first file name
FCB2    EQU     6CH             ;second file name
DBUFF   EQU     80H             ;default buffer
ABUFF   EQU     DBUFF           ;actual buffer
UNUSED  EQU     0E5H            ;dir entry
LMAX    EQU     24              ;max lines/screen
;
;Set flags in main program so only one
;copy of certain subroutines will be generated.
;Place set lines before MACLIB call.
;
BNFLAG  SET     FALSE           ;binary to ASCII bin
COFLAG  SET     FALSE           ;output console char
CRFLAG  SET     FALSE           ;carr-ret/line-feed
CXFLAG  SET     FALSE           ;binary to hex
DEFLAG  SET     FALSE           ;binary to decimal
DVFLAG  SET     FALSE           ;16-bit divide
MLFLAG  SET     FALSE           ;16-bit multiply in HL
MVFLAG  SET     FALSE           ;block move
PRFLAG  SET     FALSE           ;print console
;end of flags
;
        MACLIB  CPMMAC
;
ORG     TPA
;
```

Figure 8.7: Program to Display the Disk Parameters

```
START:
            ENTER
            VERSN   '(current date).DIREC.1'
            CPMVER                  ;check version
            CPI     20H
            JC      ERRVER          ;wrong version
            LDA     FCB1
            ORA     A               ;drive specified?
            CNZ     SETDSK          ;yes
            CALL    GETDPH          ;disk parameters
            CALL    XAMINE
            JMP     DONE
;
;block move disk parameters to end of program
;
GETDPH:
            MVI     C,31            ;disk param address
            CALL    BDOS
            MOVE    ,DPARM,15       ;copy to end
            LHLD    BLKMAX          ;maximum # blocks
            INX     H
            SHLD    BLKMAX          ;starts at zero
            LHLD    DIRENT          ;# of directory entries
            INX     H               ;starts at zero
            DIVIDE  4               ;convert to # sectors
;Save number of directory sectors as 16 bits
            SHLD    DIRMAX          ;and save
            SHLD    DIRMX2          ;count.
;
;Directory block allocation is stored as
;1000 0000 for 1 block, 1100 0000 for 2, etc.
;But we want left byte in H.
;
            LHLD    ALLOC           ;reverse bytes
            MOV     A,L
            MOV     L,H
            MOV     H,A
            SHLD    ALLOC
;get number of directory blocks as ASCII
```

Figure 8.7 (continued)

```
        XRA     A               ;zero A
XAM3:
        DAD     H               ;shift left
        JNC     XAM4
        INR     A
        JMP     XAM3
XAM4:
        MOV     E,A             ;# dir blocks
        MVI     D,0
        LHLD    BLKMAX          ;blocks
        SBC     HL,DE           ;deduct for directory
        SHLD    NETBL           ;net data blocks
        MOV     A,E
        ORI     '0'             ;ASCII bias
        STA     ALLOCA          ;save
        RET
;
;display disk parameters
;
XAMINE:
        PRINT   <CR,LF,'Sectors/track: '>
        LHLD    NUMSEC
        HLDEC                   ;decimal
        PCHAR   BLANK
        PCHAR   '('
        OUTHL
        PRINT   <' hex)',CR,LF,'Sectors/block: '>
        LDA     BLM
        INR     A
        MOV     L,A
        MVI     H,0
        HLDEC
        PCHAR   BLANK
        PCHAR   '('
        OUTHEX  L
        PRINT   ' hex)'
        PRINT   <CR,LF,'Block size: '>
        DIVIDE  8
        MOV     B,L             ;save block size
```

Figure 8.7 (continued)

```
        HLDEC
        PRINT   'K bytes'
        LHLD    NETBL           ;# data blocks
        MOV     A,B             ;block size
        MULT
        PRINT   <CR,LF,'Disk size: '>
        HLDEC
        PRINT   'K bytes'
        PRINT   <CR,LF,'Extents/entry: '>
        LDA     EMASK
        INR     A
        MOV     L,A
        MVI     H,0
        HLDEC
        PRINT   <CR,LF,'Number of blocks: '>
        LHLD    BLKMAX
        HLDEC
        PCHAR   BLANK
        PCHAR   '('
        OUTHL
        PRINT   <' hex)',CR,LF,'Max directory entries: '>
        LHLD    DIRENT
        INX     H
        HLDEC
        PCHAR   BLANK
        PCHAR   '('
        OUTHL
        PRINT   <' hex)',CR,LF,'Directory blocks: '>
        LDA     ALLOCA
        PCHAR
        PCHAR   BLANK
        PCHAR   '('
        LDA     ALLOC+1
        BINBIN                  ;alloc in binary
        LDA     ALLOC
        ORA     A
        JZ      XAM2
        BINBIN                  ;2nd if needed
XAM2:
```

Figure 8.7 (continued)

```
        PRINT   <')',CR,LF,'Track offset: '>
        LHLD    TRKOFF
        HLDEC
        MOV     A,H
        ORA     A
        JZ      XAM5            ;skip hex
        PCHAR   BLANK
        PCHAR   '('
        OUTHL
        PRINT   ' hex)'
XAM5:
        CRLF
        RET
SETDSK:                         ;set disk drive
        DCR     A               ;0=A, 1=B
        MOV     E,A
        MVI     C,14            ;select new drive
        JMP     BDOS
ERRVER:
        ERRORM  '?CP/M version must be 2 or greater'
DONE:
        EXIT
DPARM:                          ;copy of disk parameters
NUMSEC: DS      2               ;sectors per track
BSHIFT: DS      1               ;block shift
BLM:    DS      1               ;block mask
EMASK:  DS      1               ;extent mask
BLKMAX: DS      2               ;max # blocks on disk
DIRENT: DS      2               ;max # dir entries
ALLOC:  DS      2               ;AL1, AL0 reversed
CKS:    DS      2               ;check size
TRKOFF: DS      2               ;track offset
;
DIRMAX: DS      2               ;max # directory sectors
NETBL:  DS      2               ;number of data blocks
ALLOCA: DS      1               ;directory blocks (ASCII)
DIRMX2: DS      2               ;remaining dir sectors
;
        END     START
```

Figure 8.7 (continued)

The results for an 8-inch, single-density floppy will look like Figure 8.8. The display shows that there are 26 sectors per track and 8 sectors per block of 1K bytes. There are 64 directory entries that are stored in 2 blocks. The disk can hold a maximum of 241K bytes of data, exclusive of the directory. The track offset is 2; that is, the first two tracks are reserved.

On the other hand, a 5-inch, double-density, double-sided, hard-sectored floppy might give the results shown in Figure 8.9. In this example, there are 40 logical sectors per track and 16 sectors per 2K block. The disk can store a maximum of 338K bytes, exclusive of the one block reserved for the directory.

```
Sectors/track: 26 (1A hex)
Sectors/block: 8 (8 hex)
Block size: 1K bytes
Disk size: 241K bytes
Extents/entry: 1
Number of blocks: 243 (F3 hex)
Max directory entries: 64 (40 hex)
Directory blocks: 2 (11000000)
Track offset: 2
```

Figure 8.8: The Disk Parameters for an 8-Inch Floppy

```
Sectors/track: 40 (28 hex)
Sectors/block: 16 (10 hex)
Block size: 2K bytes
Disk size: 338K bytes
Extents/entry: 2
Number of blocks: 170 (AA hex)
Max directory entries: 64 (40 hex)
Directory blocks: 1 (10000000)
Track offset: 2
```

Figure 8.9: The Disk Parameters for a 5-Inch Floppy

THE DISK DIRECTORY BLOCKS

The first one or more data blocks on each CP/M disk contain a directory of the files that are present on the remainder of the disk. As we saw in Chapter 6, each directory entry is 32 bytes long. Consequently, a logical 128-byte sector can reference a maximum of four disk files.

We saw that the first byte of each directory entry refers to the user who created the file. This is a binary number from 0 to 15. A value of E5 hex in this position indicates that the file has been deleted. The file name and extent are coded in ASCII in the next 11 bytes. Then there are four bytes that contain the extent number and the number of records.

The actual location of the file is indicated by the remaining 16 bytes. For smaller disks, each block is identified as a one-byte binary number. Larger disks use two-byte block numbers with the low-order byte given first.

Let us now see how a group of files is stored on three different types of disks. The three disk types are as follows:

- 1K-byte block size, 1-byte block addresses
- 2K-byte block size, 1-byte block addresses
- 2K-byte block size, 2-byte block addresses

Our last program in the book will investigate the FCB. For a 1K block size we might obtain a listing as follows:

```
00 CPMIO    ASM 00000055 02030405060708090A0B0C00
00 DUMP     COM 00000007 0D00
00 GO       COM 00000002 0E00
00 LOAD     COM 0000000E 0F1000
00 CPMIO    HEX 00000007 1100
00 WSOVLY1  OVR 00000080 12131415161718191A1B1C1D1E1F2021
00 WSOVLY1  OVR 01000080 22232425262728292A2B2C2D2E2F3031
00 WSOVLY1  OVR 0200000A 323300
01 TIME     COM 0000000A 343500
02 SORT     BAS 00000009 363700
02 SORT     BAK 0000000F 383900
E5 SORTA    BAS 00000008 3A00
E5 PRIN     STR 0000000C 3B3C00
```

The first FCB, CPMIO.ASM, belongs to user 0. It contains 55 (85 decimal) records stored in blocks 02 to 0C. The block numbers refer to the actual regions on the disk where the file is stored. The next file, DUMP.COM, has seven records; they all fit into block 0D. Lines 6, 7, and 8 of the directory listing refer to the same file, WSOVLY1.OVR. This file

is so large that it requires three FCBs (called physical extents). The first and second FCBs (designated 0 and 1) contain 80 records. The third FCB is designated as extent 2; it has 0A records. The block numbers run from 12 to 33 hex. For this disk format, each FCB can reference a maximum of one extent of 80 records (16K bytes).

The block size for the previous example is 1K bytes. However, CP/M disks may have block sizes of 2K, 4K, 8K, or 16K bytes. The next disk format we will consider has 2K bytes per block. With this format each FCB can contain a maximum of two 16K logical extents. Let us see how the previous files are stored with this format:

```
00 CPMIO    ASM  00000055  01020304050600
00 DUMP     COM  00000007  0700
00 GO       COM  00000002  0800
00 LOAD     COM  0000000E  0900
00 CPMIO    HEX  00000007  0A00
00 WSOVLY1  OVR  01000080  0B0C0D0E0F101112131415161718191A
00 WSOVLY1  OVR  0200000A  1B00
01 TIME     COM  0000000A  1C00
02 SORT     BAS  00000009  1D00
02 SORT     BAK  0000000F  1E00
```

The files in this example are the same size as in the previous example. However, fewer blocks are needed for the longer files because the block size is twice as large. Notice that WSOVLY1.OVR needs only two FCBs rather than three. However, the first of these two FCBs shows an extent of 1, meaning that extents 0 and 1 are both contained in one FCB. Each of the two extents has 80 records. The third extent, extent 2, is contained in the second FCB. It has 0A records. The block numbers run from 0B to 1B.

As a third example, consider a high-density disk that also has a 2K-byte block size but uses two-byte block addresses. The same files are shown again. However, the block addresses now appear as 0200, 0300, 0400, and so on:

```
00 CPMIO    ASM  00000055  020003000400050006000700
00 DUMP     COM  00000007  0800
00 GO       COM  00000002  0900
00 LOAD     COM  0000000E  0A00
00 CPMIO    HEX  00000007  0B00
00 WSOVLY1  OVR  00000080  0C000D000E000F00100011001200 1300
00 WSOVLY1  OVR  01000080  140015001600170018001900 1A001B00
00 WSOVLY1  OVR  0200000A  1C00
```

THE BLOCK ALLOCATION MAP

When a warm start is performed by typing control-C, all disk drives are reset and the directory on drive A is read. If a drive other than A is currently the default drive, the disk directory for that drive is also read at this time. A block allocation map is constructed during this initialization step. This map uses a single bit to represent each block. Blocks that are currently in use are given a value of 1. Unused blocks have a value of 0. CP/M searches the map for unused blocks when a new file is to be created.

The third part of our last program constructs a block map not from the CP/M version, but by actually locating each block address in the disk directory. Our technique will show when there are multiple references to a particular block.

The map for a newly formatted disk might look like that in Figure 8.10. In this example, the first block, block 0, is located in the upper-left corner. The value of 1 means that the block is in use. The remaining positions have a value of 0, indicating that they are not in use. The first block of this disk is reserved for the directory. The directory itself will always occupy the first one or more blocks at the beginning of the data area. Thus there will always be values of 1 at the beginning of this map. The number of blocks given in the summary at the bottom of the display does not include those reserved for the directory.

Each time a file is saved on a disk, the corresponding blocks will be set to 1. As more and more files are saved, the block map gradually becomes filled in. If some files are erased, holes will open up in the table. After a while, the map might look like that in Figure 8.11.

The allocation map also can indicate whether there are multiple links to a file. For example, consider the allocation map in Figure 8.12. In this example, several blocks (41, 42, 47, and others) are marked with a value of 2. This

```
     01234567  89ABCDEF  01234567  89ABCDEF
00:  10000000  00000000  00000000  00000000
20:  00000000  00000000  00000000  00000000
40:  00000000  00000000  00000000  00000000
60:  00000000  00000000  00000000  00000000
80:  00000000  00000000  00000000  00000000
A0:  00000000  00
169 total blocks, 0 in use, 169 remaining
```

Figure 8.10: Block Allocation Map for a Newly Formatted Disk

```
     01234567  89ABCDEF  01234567  89ABCDEF
00:  11111111  11111111  11111111  11111111
20:  11111111  11111000  01111111  11111111
40:  11111111  11111111  10000001  11111111
60:  11111111  00000000  00000000  00000000
80:  00000000  00000000  00000000  11111111
A0:  11111000  00
169 total blocks, 63 in use, 106 remaining
```

Figure 8.11: Block Allocation Map for a Disk with Files

```
     01234567  89ABCDEF  01234567  89ABCDEF
00:  11111111  11111111  11111111  11111111
20:  11111111  11111111  11111111  11111111
40:  12211112  21111111  11111111  11111111
60:  22111111  11111110  11111111  11111121
80:  11101222  21111111  11111111  11111111
A0:  11111101  11
169 total blocks, 166 in use, 3 remaining
```

Figure 8.12: Block Allocation Map Indicating Multiple Links to a File

indicates that there are two different files that refer to these blocks. This can occur with a disk utility program such as BADLIM or RECLAIM. These programs read the entire disk looking for bad sectors. If bad sectors are found, they are collected into a special file so that they will not be used. Of course, if the original program is still present, it will also refer to these sections.

VIEWING THE DISK DIRECTORY BLOCKS AND THE BLOCK ALLOCATION MAP

In this section we will extend our directory program so it will display the actual directory entries. The user number and block addresses will be shown. A separate feature will construct a block allocation map for the disk. Before we develop this program, however, we need to add one more macro to our library.

A Macro to Fill Large Blocks

In Chapter 4 we wrote a macro to fill an area of memory with a particular byte. The area was limited to 256 bytes, because a single register was used to count the number of locations to fill. For the next program we will need to fill an area larger than 256 bytes. Consequently, we will alter our FILL macro so that a double register is used to count the number of locations.

Make a copy of macro FILL and give it the name FILLD (for double precision). Alter the new macro so it looks like Figure 8.13. Notice that the same flag, FLFLAG, is used for this version. This means that you should only use one of these two macros for any particular program.

We are now ready to add two new features to our directory program.

The DIREC Program, Version 2

Make a copy of the first version of the directory program shown in Figure 8.7 and alter it to look like the version shown in Figure 8.14. Macro FILLD is needed. Assemble the program and try it out. As with the previous version, if you execute this program without a parameter, the currently logged-in disk is used. However, if you want to select another disk, give the disk name followed by a colon.

This new version begins by printing out the disk parameters, just as the previous version did. But then if you press any console key, the program will continue. Each directory entry is shown on a separate line. The user number, extent, number of records, and block addresses are included in this listing. Pressing any console key a second time will display the third part of the program—a block allocation map for the disk. Blocks that are in use will be designated by a value of 1 in the map. Blocks that are free are shown by zeros.

Program DIREC begins like the previous one with macros ENTER, VERSN, and CPMVER. Subroutine CDISK is called to determine the default disk drive. BDOS function 25 is used for this task. A check is made to see if a disk drive was specified on the command line. If so, subroutine SETDSK is called as before. If no drive was specified, the default drive is coded in location FCB1. The drive name is also displayed on the console at that time.

The disk parameters are displayed, as they were with the previous version. The program then waits for any key to be pressed. This causes the complete disk directory with the block addresses to be shown. When any other key is pressed, the block allocation map is displayed.

```
FILLD       MACRO   ADDR, BYTES, CHAR
;;(Put current date here)
;;(double precision version)
;;Inline macro to fill BYTES memory
;;locations with CHAR starting at ADDR.
;;Usage:    FILL    FCB+1, 8, blank
;;          FILL    FCB+9, 3, '?'
;;
            LOCAL   AROUND,FILL3?
            PUSH    H
            PUSH    B
            IF      NOT NUL ADDR
            LXI     H,ADDR
            ENDIF
            IF      NOT NUL BYTES
            LXI     B,BYTES
            ENDIF
            MVI     A,CHAR
            CALL    FILL2?
            POP     B
            POP     H
            IF      NOT FLFLAG
            JMP     AROUND
FILL2?:
            PUSH    D
            MOV     D,A
FILL3?:
            MOV     M,D
            INX     H
            DCX     B
            MOV     A,C
            ORA     B
            JNZ     FILL3?
            POP     D
            RET
FLFLAG      SET     TRUE
            ENDIF
AROUND:                             ;;FILLD
            ENDM
```

Figure 8.13: Macro FILLD to Fill a Large Portion of Memory

```
TITLE   'DIREC, directory utility'
;
;(Put current date here)
;
;
FALSE   EQU     0
TRUE    EQU     NOT FALSE
;
BOOT    EQU     0
BDOS    EQU     5               ;BDOS entry point
TPA     EQU     100H
FCB     EQU     5CH             ;file control block
FCB1    EQU     5CH             ;first file name
FCB2    EQU     6CH             ;second file name
DBUFF   EQU     80H             ;default buffer
ABUFF   EQU     DBUFF           ;actual buffer
UNUSED  EQU     0E5H            ;dir entry
;
;Set flags in main program so only one
;copy of certain subroutines will be generated.
;Place set lines before MACLIB call.
;
BNFLAG  SET     FALSE           ;binary to ASCII bin
CIFLAG  SET     FALSE           ;input console char
COFLAG  SET     FALSE           ;output console char
CRFLAG  SET     FALSE           ;carr-ret/line-feed
CXFLAG  SET     FALSE           ;binary to hex
DEFLAG  SET     FALSE           ;binary to decimal
DVFLAG  SET     FALSE           ;16-bit divide
FLFLAG  SET     FALSE           ;fill characters
MLFLAG  SET     FALSE           ;16-bit multiply in HL
MVFLAG  SET     FALSE           ;block move
PRFLAG  SET     FALSE           ;print console
;end of flags
;
        MACLIB  CPMMAC
;
ORG     TPA
```

Figure 8.14: Program DIREC to Display Disk Parameters and the Block Allocation Map

```
;
START:
            ENTER
            VERSN   '(current date).DIREC.2'
            CPMVER                  ;check version
            CPI     20H
            JC      ERRVER          ;wrong version
            PRINT   'For disk drive '
            CALL    CDISK           ;get current disk
            LDA     FCB1
            ORA     A               ;drive specified?
            CNZ     SETDSK          ;yes
            LDA     CURD2           ;requested drive
            STA     FCB1            ;binary
            ADI     'A'             ;convert to ASCII
            PCHAR
            CALL    GETDPH          ;disk parameters
            CALL    XAMINE          ;show them
            PRINT   'Press any key to continue: '
            READCH                  ;wait for character
            CALL    REPEAT          ;reset parameters
            CALL    BLOCK           ;block map
            JMP     DONE
;
;block move disk parameters to end of program
;
GETDPH:
            MVI     C,31            ;disk param address
            CALL    BDOS
            MOVE    ,DPARM,15       ;copy to end
            LHLD    BLKMAX          ;maximum # blocks
            INX     H
            SHLD    BLKMAX          ;starts at zero
            LHLD    DIRENT          ;# of directory entries
            INX     H               ;starts at zero
            DIVIDE  4               ;convert to # sectors
;Save number of directory sectors as 16 bits
            SHLD    DIRMAX          ;and save
```

Figure 8.14 (continued)

```
        SHLD    DIRMX2          ;count.
;
;Directory block allocation is stored as
;1000 0000 for 1 block, 1100 0000 for 2, etc.
;But we want left byte in H.
;
        LHLD    ALLOC           ;reverse bytes
        MOV     A,L
        MOV     L,H
        MOV     H,A
        SHLD    ALLOC
;get number of directory blocks as ASCII
        XRA     A               ;zero A
XAM3:
        DAD     H               ;shift left
        JNC     XAM4
        INR     A
        JMP     XAM3
XAM4:
        MOV     E,A             ;# dir blocks
        MVI     D,0
        LHLD    BLKMAX          ;blocks
        SBC     HL,DE           ;deduct for directory
        SHLD    NETBL           ;net data blocks
        MOV     A,E
        ORI     '0'             ;ASCII bias
        STA     ALLOCA          ;save
;
;select disk and setup disk parameter header
;
        LDA     FCB
        MOV     C,A
        CALL    SELDSK          ;select drive
        MOV     A,H             ;HL has DPH
        ORA     L
        JZ      ILDISK          ;error, no disk
        MOV     E,M             ;get translate table
        INX     H               ;address
```

Figure 8.14 (continued)

```
            MOV     D,M
            XCHG
            SHLD    DPH             ;save address
            RET
;
;display disk parameters
;
XAMINE:
            PRINT   <CR,LF,'Sectors/track: '>
            LHLD    NUMSEC
            HLDEC                   ;decimal
            PCHAR   BLANK
            PCHAR   '('
            OUTHL
            PRINT   <' hex)',CR,LF,'Sectors/block: '>
            LDA     BLM
            INR     A
            MOV     L,A
            MVI     H,0
            HLDEC
            PCHAR   BLANK
            PCHAR   '('
            OUTHEX  L
            PRINT   ' hex)'
            PRINT   <CR,LF,'Block size: '>
            DIVIDE  8
            MOV     B,L             ;save block size
            HLDEC
            PRINT   'K bytes'
            LHLD    NETBL           ;# data blocks
            MOV     A,B             ;block size
            MULT
            PRINT   <CR,LF,'Disk size: '>
            HLDEC
            PRINT   'K bytes'
            PRINT   <CR,LF,'Extents/entry: '>
            LDA     EMASK
            INR     A
```

Figure 8.14 (continued)

```
        MOV     L,A
        MVI     H,0
        HLDEC
        PRINT   <CR,LF,'Number of blocks: '>
        LHLD    BLKMAX
        HLDEC
        PCHAR   BLANK
        PCHAR   '('
        OUTHL
        PRINT   <' hex)',CR,LF,'Max directory entries: '>
        LHLD    DIRENT
        INX     H
        HLDEC
        PCHAR   BLANK
        PCHAR   '('
        OUTHL
        PRINT   <' hex)',CR,LF,'Directory blocks: '>
        LDA     ALLOCA
        PCHAR
        PCHAR   BLANK
        PCHAR   '('
        LDA     ALLOC+1
        BINBIN                  ;alloc in binary
        LDA     ALLOC
        ORA     A
        JZ      XAM2
        BINBIN                  ;2nd if needed
XAM2:
        PRINT   <')',CR,LF,'Track offset: '>
        LHLD    TRKOFF
        HLDEC
        MOV     A,H
        ORA     A
        JZ      XAM5            ;skip hex
        PCHAR   BLANK
        PCHAR   '('
        OUTHL
        PRINT   ' hex)'
```

Figure 8.14 (continued)

```
XAM5:
        CRLF
        RET
SETDSK:                                 ;set disk drive
        DCR     A                       ;0=a, 1=b
        STA     CURD2                   ;save
        MOV     E,A
        MVI     C,14                    ;select new drive
        JMP     BDOS
;
REPEAT:                                 ;reset parameters
        FILLD   SECTOR, HERE-SECTOR,0
        LHLD    TRKOFF
        SHLD    TRACK                   ;reset track offset
        LHLD    DIRMAX                  ;# directory sectors
        SHLD    DIRMX2
        RET
;
;show block allocation map
;
BLOCK:
;
;Set reserved directory blocks in map
;by shifting alloc to left.
;
        PRINT   <CR,LF,LF,' disk allocation map',CR,LF>
        LHLD    BLKMAX                  ;number of blocks
        MOV     B,H
        MOV     C,L                     ;put in BC
        FILLD   BMAP, , 0               ;zero map area
        LHLD    ALLOC                   ;both bytes
        LXI     D,BMAP                  ;location
C14A:
        XCHG
        INR     M                       ;set bit
        INX     H
        XCHG
        DAD     H                       ;shift left
```

Figure 8.14 (continued)

```
        MOV     A,L
        ORA     H               ;zero
        JNZ     C14A            ;no
BLOCK3:
        CALL    NXTSEC
        JZ      BLOCK4          ;finished
        CALL    BPROG
        JMP     BLOCK3
;
;display disk allocation map
;
BLOCK4:
        PRINT   'Press any key to continue: '
        READCH                  ;wait for character
        PRINT   <CR,LF,LF,'     01234567 89ABCDEF'>
        PRINT   ' 01234567 89ABCDEF'
        LHLD    BLKMAX
        MOV     B,H
        MOV     C,L
        LXI     H,BMAP          ;start of map
BMAP2:
        MOV     A,L
        ANI     0FH             ;mask upper 4 bits
        JNZ     BMAP6
        MOV     A,L
        ANI     1FH
        JZ      BMAP7           ;mask 4 bits
        PCHAR   BLANK           ;even
        JMP     BMAP5
BMAP7:
        ABORT   ESC
        CRLF                    ;start new line
        OUTHEX  L               ;show address
        PCHAR   ':'
        PCHAR   BLANK
        JMP     BMAP5
BMAP6:
        CPI     8
```

Figure 8.14 (continued)

```
        JNZ     BMAP5
        PCHAR   BLANK
BMAP5:
        MOV     A,M             ;get entry
        ORA     A               ;zero?
        JZ      BMAP8           ;yes
        XCHG                    ;save HL in DE
        LHLD    BLKCNT
        INX     H               ;use count
        SHLD    BLKCNT
        XCHG                    ;restore HL
BMAP8:
        CPI     10
        JNC     BMAP3           ;<9
        ORI     '0'             ;make ASCII
        JMP     BMAP4
BMAP3:
        ADI     'A'-10          ;make hex
BMAP4:
        PCHAR                   ;print
        INX     H
        DCX     B               ;count
        MOV     A,C             ;done?
        ORA     B
        JNZ     BMAP2           ;no
;
;show total number of blocks and number in use
;
        CRLF
        LHLD    NETBL           ;net # blocks
        HLDEC
        PUSH    H               ;save HL
        PRINT   ' total blocks, '
        LHLD    BLKCNT
        LDA     ALLOCA          ;dir blocks
        SUI     '0'             ;make binary
        MOV     E,A
```

Figure 8.14 (continued)

```
        MVI     D,0
        SBC     HL,DE
        HLDEC
        PRINT   ' in use, '
        XCHG
        POP     H
        SBC     HL,DE           ;difference
        HLDEC
        PRINT   <' remaining',CR,LF>
        RET
;
;code for setting up block allocation map
;
BPROG:
        CALL    E5AREA          ;found E5?
        MOV     A,M             ;first byte
        CPI     17              ;user > 16?
        JNC     BKINCD          ;yes
        PUSH    H               ;save pointer
;
        OUTHEX                  ;user number
        PCHAR   BLANK
        INX     H               ;file name
        PRINT   ,11             ;display file name
        PCHAR   BLANK
        LXI     D,11            ;move past file name
        DAD     D               ;first entry
        MVI     C,4             ;next 4 bytes
;
;next 4 bytes have extent and number of sectors
;
LOOP2:
        OUTHEX  M
        INX     H
        DCR     C
        JNZ     LOOP2
        PCHAR   BLANK
```

Figure 8.14 (continued)

```
            MVI     C,16            ;16 blocks/extent
;
;See if there are more than 255 blocks.
;A 16-bit block address is used if so.
;
            LDA     BLKMAX+1        ;high half
            ORA     A               ;zero?
            JNZ     BNEXT6          ;no, 16 bits
;
;code for 8-bit block addresses
BNEXT8:
            MOV     A,M             ;get byte
            OUTHEX                  ;display block number
            ORA     A               ;zero?
            JZ      BPRT2           ;last block
            PUSH    H
            LXI     H,BMAP          ;start
            MOV     E,A
            MVI     D,0
            DAD     D               ;offset
            INR     M               ;show use
            POP     H
            INX     H
            MOV     A,L
            ANI     0FH             ;end of line
            JNZ     BNEXT8          ;no
            JMP     BPRT2
;
;16-bit block addresses
;
BNEXT6:
            MOV     E,M             ;low byte
            OUTHEX  E               ;block number, low
            INX     H
            MOV     A,M             ;high
            OUTHEX                  ;block number, high
            ORA     E               ;zero?
```

Figure 8.14 (continued)

```
        JZ      BPRT2       ;yes, quit
        MOV     D,M
        PUSH    H           ;save pointer
        LXI     H,BMAP      ;map start
        DAD     D           ;add address
        INR     M           ;show use
        POP     H           ;restore pointer
        INX     H           ;next location
        MOV     A,L
        ANI     0FH         ;end of line?
        JNZ     BNEXT6      ;no
BPRT2:
        POP     H           ;beginning of FCB
        CRLF                ;new line
BKINCD:
        CALL    DECCNT
        JZ      CKDONE
        LXI     D,32        ;FCB length
        DAD     D           ;next entry
        JMP     BPROG
;
;increment count, decrement sector number
;
CKDONE:
        LHLD    DIRMX2
        DCX     H           ;sector count
        SHLD    DIRMX2
        LHLD    SECTOR      ;16 bits
        INX     H
        SHLD    SECTOR
        XCHG                ;save in DE
        LHLD    NUMSEC      ;sectors/track
;see if we need another track
;
        SBC     HL,DE       ;difference
        MOV     A,L
        ORA     H           ;difference zero?
```

Figure 8.14 (continued)

```
            RNZ
            SHLD    SECTOR          ;set to zero
            LHLD    TRACK
            INX     H               ;incr track
            SHLD    TRACK
            RET
;
;Read next sector (4 directory entries).
;Return with zero flag set if no more.
;
NXTSEC:
            LDA     E5FLAG          ;uninitialized found?
            CPI     1
            RZ                      ;yes
NXTSF:
            LHLD    DIRMX2          ;more sectors?
            MOV     A,L
            ORA     H               ;set flags
            RZ                      ;no
            CALL    SETTRK          ;set track
            LHLD    SECTOR          ;16 bits
            MOV     B,H
            MOV     C,L
            CALL    TRANSL
            CALL    SETSEC          ;set sector
            CALL    READ
            MVI     A,4             ;entries/sector
            STA     ECOUNT          ;reset
            LXI     H,ABUFF         ;DMA address
            ANI     1
            XRI     1               ;invert error flag
            RET                     ;zero if bad flag
;
;Decrement number of remaining entries in sector
;(4 maximum). Zero flag set if no more.
DECCNT:
            LDA     ECOUNT          ;entries/sector
            DCR     A
```

Figure 8.14 (continued)

```
        STA     ECOUNT
        RET
;
;look for E5 uninitialized area, set E5FLAG = 1 if so
;
E5AREA:
        INX     H               ;1st char
        INX     H               ;2nd char
        MOV     A,M
        CPI     UNUSED
        DCX     H
        DCX     H
        RNZ                     ;not found
        MVI     A,1
        STA     E5FLAG          ;set flag
        RET
;
;find currently logged-in disk
;
CDISK:
        MVI     C,25
        CALL    BDOS
        STA     CURD2           ;A=0, B=1
        ADI     'A'             ;convert to ASCII
        STA     CURDSK
        RET
;
;translate BC from logical to physical
;sector number BC => HL => BC
;
TRANSL:
        LHLD    DPH             ;translate table
        XCHG
        CALL    SECTRN
        MOV     B,H
        MOV     C,L
        RET
;
```

Figure 8.14 (continued)

```
;set track to 16-bit value in BC
;
SETTRK:
        LHLD    TRACK       ;16 bits
        MOV     B,H         ;may be zero
        MOV     C,L
        LHLD    BOOT+1      ;warm boot
        PUSH    D
        LXI     D,3*9       ;offset
        DAD     D
        POP     D
        PCHL
SETSEC:                     ;select sector in BC
        LHLD    BOOT+1      ;warm boot
        PUSH    D
        LXI     D,3*10      ;offset
        DAD     D
        POP     D
        PCHL
SELDSK:                     ;select disk in C
        LHLD    BOOT+1      ;warm boot
        PUSH    D
        LXI     D,3*8       ;offset
        DAD     D
        POP     D
        PCHL
;read sector, A=0 if successful
READ:
        LHLD    BOOT+1      ;warm boot
        PUSH    D
        LXI     D,3*12      ;offset
        DAD     D
        POP     D
        PCHL
;write sector, A=0 if successful
WRITE:
        LHLD    BOOT+1      ;warm boot
        PUSH    D
```

Figure 8.14 (continued)

```
            LXI     D,3*13          ;offset
            DAD     D
            POP     D
            PCHL
;
;Sector translation from logical sector in BC
;to physical sector in HL, DE has translate table.
SECTRN:
            LHLD    BOOT+1          ;warm boot
            PUSH    D
            LXI     D,3*15          ;offset
            DAD     D
            POP     D
            PCHL
;
ERRVER:
            ERRORM  '?CP/M version must be 2 or greater'
ILDISK:
            ERRORM  '?Illegal disk drive'
DONE:
            EXIT
;
DPARM:                              ;copy of disk parameters
NUMSEC:     DS      2               ;sectors per track
BSHIFT:     DS      1               ;block shift
BLM:        DS      1               ;block mask
EMASK:      DS      1               ;extent mask
BLKMAX:     DS      2               ;max # blocks on disk
DIRENT:     DS      2               ;max # dir entries
ALLOC:      DS      2               ;A11,A10 reversed
CKS:        DS      2               ;check size
TRKOFF:     DS      2               ;track offset
DPH:        DS      2               ;disk parameter header
;
DIRMAX:     DS      2               ;max # directory sectors
NETBL:      DS      2               ;number of data blocks
ALLOCA:     DS      1               ;directory blocks (ASCII)
```

Figure 8.14 (continued)

```
DIRMX2:  DS      2                ;remaining direct sectors
CURDSK:  DS      1                ;current disk (ASCII)
CURD2:   DS      1                ;current disk (binary)
ECOUNT:  DS      1                ;entries in sector (0-3)
;
SECTOR:  DS      2                ;current sector
TRACK:   DS      2                ;current track
E5FLAG:  DS      1                ;found uninitialized if 1
BLKCNT:  DS      2                ;blocks in use
;
HERE:                             ;replace with ASEG:
ORG      (here and 0FF00H) + 100H ;ORG 0A00H for Microsoft
BMAP:    DS      1                ;block allocation map
;
         END     START
```

Figure 8.14 (continued)

This program will perform several disk operations by directly calling the BIOS rather than the BDOS. Recall that the BIOS begins with a sequence of jump vectors. In this case we are interested in the vectors that are 8, 9, 10, 12, 13, and 15 positions from the warm-start vector. These are the following vectors:

```
JMP   SELDSK   Select disk drive from register C
JMP   SETTRK   Set track number to value in BC
JMP   SETSEC   Set sector number to value in BC
JMP   READ     Read a sector into memory at DMA address
JMP   WRITE    Write a sector from memory at DMA address
JMP   SECTRN   Translate from logical to physical sector
```

The location of BIOS changes with each different size of CP/M. Consequently, the exact addresses cannot be coded directly into our program.

The address of the warm-start vector is stored at address 1. We can retrieve this address, add the offset to the desired vector, then branch to it. For example, the vector to select the disk is eight vectors past the warm-start vector, and each vector is three bytes long. Consequently, we need to add 3 times 8 to the warm-start vector. The instructions are as follows:

```
SELDSK:                          ;select disk in C
        LHLD    BOOT+1           ;warm boot to HL
```

```
PUSH    D       ;save DE
LXI     D,3*8   ;put offset in DE
DAD     D       ;add to HL
POP     D       ;restore DE
PCHL            ;branch to HL address
```

The other five routines operate in a similar way.

Each time a disk is logged in, CP/M constructs a bit map of the sectors in use. However, we will not use this map in our program. Rather, we will construct a separate table. A block of memory starting at BMAP at the end of the program is set aside for this purpose. We saw previously that CP/M allocates one bit for each block. However, in this case we will use one byte for each block. The map area is zeroed initially. Then each time a block number is encountered, the corresponding location in the block is incremented.

We start with the blocks allocated to the directory. Then each directory entry is scanned for blocks that are in use. When a block is found to be in use, the corresponding entry in the table is incremented.

SUMMARY

In this chapter we studied the CP/M disk directory in detail. We developed a disk program to list the disk parameters, show an expanded directory with the block addresses, and generate a block allocation map. We can add more features to this program to further increase its usefulness. For example, an accidentally deleted file can be recovered if the value at the beginning of the FCB can be changed from E5 to 0.

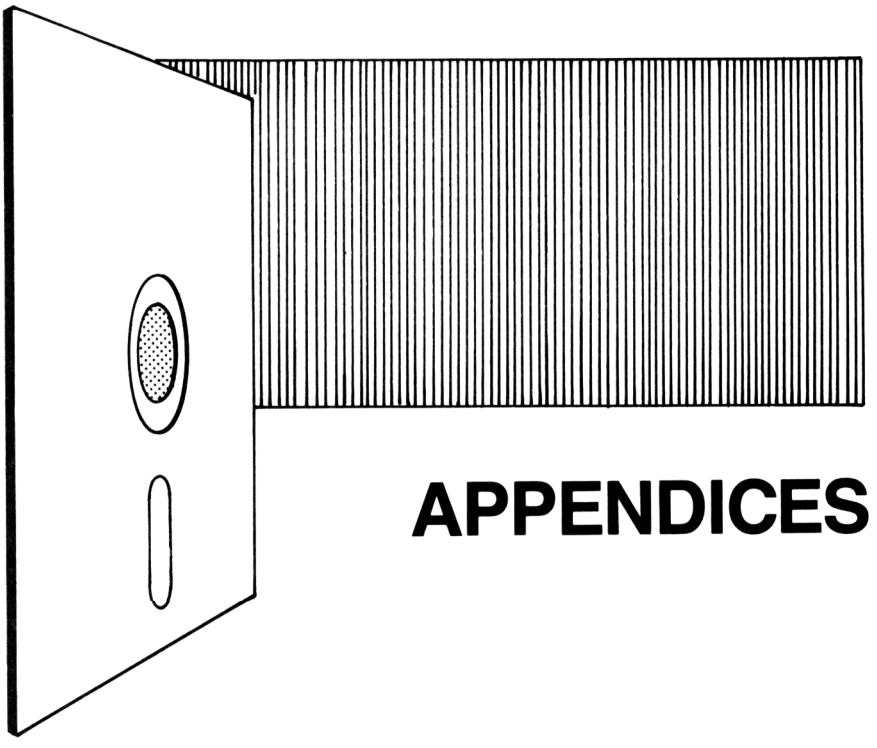
Creating multiple links to a single file is another useful feature if more than one user area is active. If a particular program is needed in more than one user area, it is usually necessary to save a separate copy of the program for each user. But this requires additional disk space. On the other hand, disk space can be saved by creating multiple FCBs to the same file. One FCB is designated for each user according to the initial byte of the FCB. However, the remainder of each FCB is the same. Thus all of these FCBs refer to the same file. (All of these features are incorporated into a disk utility program called FILEFIX, available commercially.)

The directory of your macro library should now look like this:

```
;;Macros in this library                 Flags
;;ABORT      MACRO   CHAR                CIFLAG, COFLAG
;;AMBIG      MACRO   OLD, NEW            (none)
```

```
;;BINBIN     MACRO                       BNFLAG
;;CLOSE      MACRO     POINTR            CLFLAG, COFLAG, CRFLAG
;;                                       PRFLAG, OPFLAG, MVFLAG,
;;                                       DEFLAG, CIFLAG, UNFLAG,
;;                                       RNFLAG, S2FLAG
;;COMPAR     MACRO     FIRST, SECOND, BYTES CMFLAG
;;COMPRA     MACRO     FIRST, SECOND, BYTES CMFLAG
;;CPMVER     MACRO                       (none)
;;CRLF       MACRO                       CRFLAG, COFLAG
;;DELETE     MACRO     POINTR, WHERE     DEFLAG, CIFLAG
;;                                       COFLAG, PRFLAG, UNFLAG
;;DIVIDE     MACRO     DENOM             DVFLAG
;;ENTER      MACRO                       (none)
;;ERRORM     MACRO     TEXT, WHERE       COFLAG, CRFLAG, PRFLAG
;;EXIT       MACRO     SPACE?            (none)
;;FILL       MACRO     ADDR, BYTES, CHAR FLFLAG
;;FILLD      MACRO     ADDR, BYTES, CHAR FLFLAG
;;GFNAME     MACRO     FCB               FNFLAG, FLFLAG, RCFLAG
;;                                       COFLAG, CRFLAG, PRFLAG
;;HEXHL      MACRO     POINTR            HXFLAG, RCFLAG
;;HLDEC      MACRO                       DEFLAG, COFLAG
;;LCHAR      MACRO     PAR               LOFLAG
;;LDFILE     MACRO     FCB, POINTR, CHAR COFLAG, DMFLAG
;;                                       RDFLAG
;;MAKE       MACRO     POINTR            MKFLAG, COFLAG, CRFLAG,
;;                                       PRFLAG
;;MOVE       MACRO     FROM, TO, BYTES   MVFLAG
;;MULT       MACRO     TIMES             MLFLAG
;;OPEN       MACRO     POINTR, WHERE     OPFLAG, COFLAG, PRFLAG
;;                                       CRFLAG
;;OUTHEX     MACRO     REG               CXFLAG, COFLAG
;;OUTHL      MACRO                       CXFLAG, COFLAG
;;PCHAR      MACRO     PAR               COFLAG
;;PFNAME     MACRO     FCB               COFLAG, PRFLAG
;;PRINT      MACRO     TEXT, BYTES       PRFLAG, COFLAG
;;PROTEC     MACRO     POINTR            (none)
;;READB      MACRO     BUFFR             RCFLAG
;;READCH     MACRO     REG               CIFLAG, COFLAG
;;READS      MACRO     POINTR, STAR      RDFLAG, COFLAG
;;RENAME     MACRO     POINTR            RNFLAG, COFLAG
;;                                       PRFLAG, CRFLAG
```

```
;;SBC        MACRO                    (none)
;;SETDMA     MACRO    POINTR          DMFLAG
;;SETUP2     MACRO                    S2FLAG, CIFLAG, COFLAG,
;;                                    CRFLAG, CMFLAG, DEFLAG,
;;                                    MKFLAG, MVFLAG, OPFLAG,
;;                                    PRFLAG, UNFLAG
;;SYSF       MACRO    FUNC, AE        (none)
;;UCASE      MACRO    REG             (none)
;;UNPROT     MACRO    POINTR          UNFLAG
;;UPPER      MACRO    REG             (none)
;;VERSN      MACRO    NUM             (none)
;;WRITES     MACRO    POINTR, STAR    WRFLAG, COFLAG
;;                                    PRFLAG
;;WRFILE     MACRO    FCB, POINTR     COFLAG, CRFLAG
;;                                    DMFLAG, WRFLAG
```



APPENDICES

APPENDIX A

The ASCII Character Set

The ASCII character set is listed here in numeric order with the corresponding decimal, hexadecimal, and octal values. The control characters are indicated with a caret (^). For example, the horizontal tab (HT) is formed with control-I (^I).

ASCII symbol	Decimal value	Hex value	Octal value	Control character	Meaning
NUL	0	00	000	^@	Null
SOH	1	01	001	^A	Start of heading
STX	2	02	002	^B	Start of text
ETX	3	03	003	^C	End of text
EOT	4	04	004	^D	End of transmission
ENQ	5	05	005	^E	Inquiry
ACK	6	06	006	^F	Acknowledge
BEL	7	07	007	^G	Bell
BS	8	08	010	^H	Backspace
HT	9	09	011	^I	Horizontal tab
LF	10	0A	012	^J	Line feed
VT	11	0B	013	^K	Vertical tab
FF	12	0C	014	^L	Form feed
CR	13	0D	015	^M	Carriage return

ASCII symbol	Decimal value	Hex value	Octal value	Control character	Meaning
SO	14	0E	016	^N	Shift out
SI	15	0F	017	^O	Shift in
DLE	16	10	020	^P	Data link escape
DC1	17	11	021	^Q	Device control 1
DC2	18	12	022	^R	Device control 2
DC3	19	13	023	^S	Device control 3
DC4	20	14	024	^T	Device control 4
NAK	21	15	025	^U	Negative acknowledge
SYN	22	16	026	^V	Synchronous idle
ETB	23	17	027	^W	End of transmission block
CAN	24	18	030	^X	Cancel
EM	25	19	031	^Y	End of medium
SUB	26	1A	032	^Z	Substitute
ESC	27	1B	033	^[	Escape
FS	28	1C	034	^\	File separator
GS	29	1D	035	^]	Group separator
RS	30	1E	036	^^	Record separator
US	31	1F	037	^—	Unit separator
SP	32	20	040		Space
!	33	21	041		
"	34	22	042		
#	35	23	043		
$	36	24	044		
%	37	25	045		
&	38	26	046		
'	39	27	047		Apostrophe
(	40	28	050		
)	41	29	051		
*	42	2A	052		
+	43	2B	053		
,	44	2C	054		Comma
-	45	2D	055		Minus
.	46	2E	056		Period
/	47	2F	057		
0	48	30	060		
1	49	31	061		
2	50	32	062		
3	51	33	063		

ASCII symbol	Decimal value	Hex value	Octal value	Control character	Meaning
4	52	34	064		
5	53	35	065		
6	54	36	066		
7	55	37	067		
8	56	38	070		
9	57	39	071		
:	58	3A	072		
;	59	3B	073		
<	60	3C	074		
=	61	3D	075		
>	62	3E	076		
?	63	3F	077		
@	64	40	100		
A	65	41	101		
B	66	42	102		
C	67	43	103		
D	68	44	104		
E	69	45	105		
F	70	46	106		
G	71	47	107		
H	72	48	110		
I	73	49	111		
J	74	4A	112		
K	75	4B	113		
L	76	4C	114		
M	77	4D	115		
N	78	4E	116		
O	79	4F	117		
P	80	50	120		
Q	81	51	121		
R	82	52	122		
S	83	53	123		
T	84	54	124		
U	85	55	125		
V	86	56	126		
W	87	57	127		
X	88	58	130		
Y	89	59	131		

ASCII symbol	Decimal value	Hex value	Octal value	Control character	Meaning
Z	90	5A	132		
[	91	5B	133		
\	92	5C	134		
]	93	5D	135		
^	94	5E	136		
_	95	5F	137		Underline
`	96	60	140		
a	97	61	141		
b	98	62	142		
c	99	63	143		
d	100	64	144		
e	101	65	145		
f	102	66	146		
g	103	67	147		
h	104	68	150		
i	105	69	151		
j	106	6A	152		
k	107	6B	153		
l	108	6C	154		
m	109	6D	155		
n	110	6E	156		
o	111	6F	157		
p	112	70	160		
q	113	71	161		
r	114	72	162		
s	115	73	163		
t	116	74	164		
u	117	75	165		
v	118	76	166		
w	119	77	167		
x	120	78	170		
y	121	79	171		
z	122	7A	172		
{	123	7B	173		
|	124	7C	174		
}	125	7D	175		
~	126	7E	176		
DEL	127	7F	177		Delete

APPENDIX B

A 64K Memory Map

The 8080 and Z80 microprocessors can directly address 64K bytes of memory. The memory area is mapped out in the chart that follows. Each entry represents a 256-byte block. The high-order byte of the address is given first as a hexadecimal value, then as an octal value. For example, the entry

Hex	*Oct*	*K*	*Bl*
20	040		32

represents an address range of 2000 to 2FFF hex, or 040-000 to 040-777 octal. The third column gives the decimal number of 1K blocks. The fourth column is the decimal number of 256-byte blocks starting at address 100 hex. As an example, suppose that a CP/M program runs from 100 hex to 3035 hex. The 30 hex entry in the table shows that the program contains 48 decimal blocks of 256 bytes. The program can be saved with the CP/M command

```
A>SAVE 48 (file name)
```

Hex	Oct	K	Bl
00	000		0
01	001		1
02	002		2
03	003	1	3
04	004		4
05	005		5
06	006		6
07	007	2	7
08	010		8
09	011		9
0A	012		10
0B	013	3	11
0C	014		12
0D	015		13
0E	016		14
0F	017	4	15
10	020		16
11	021		17
12	022		18
13	023	5	19
14	024		20
15	025		21

Hex	Oct	K	Bl
16	026		22
17	027	6	23
18	030		24
19	031		25
1A	032		26
1B	033	7	27
1C	034		28
1D	035		29
1E	036		30
1F	037	8	31
20	040		32
21	041		33
22	042		34
23	043	9	35
24	044		36
25	045		37
26	046		38
27	047	10	39
28	050		40
29	051		41
2A	052		42
2B	053	11	43
2C	054		44
2D	055		45
2E	056		46
2F	057	12	47
30	060		48
31	061		49
32	062		50
33	063	13	51
34	064		52
35	065		53
36	066		54
37	067	14	55
38	070		56
39	071		57
3A	072		58
3B	073	15	59
3C	074		60

Hex	Oct	K	Bl
3D	075		61
3E	076		62
3F	077	16	63
40	100		64
41	101		65
42	102		66
43	103	17	67
44	104		68
45	105		69
46	106		70
47	107	18	71
48	110		72
49	111		73
4A	112		74
4B	113	19	75
4C	114		76
4D	115		77
4E	116		78
4F	117	20	79
50	120		80
51	121		81
52	122		82
53	123	21	83
54	124		84
55	125		85
56	126		86
57	127	22	87
58	130		88
59	131		89
5A	132		90
5B	133	23	91
5C	134		92
5D	135		93
5E	136		94
5F	137	24	95
60	140		96
61	141		97
62	142		98
63	143	25	99

Hex	Oct	K	Bl
64	144		100
65	145		101
66	146		102
67	147	26	103
68	150		104
69	151		105
6A	152		106
6B	153	27	107
6C	154		108
6D	155		109
6E	156		110
6F	157	28	111
70	160		112
71	161		113
72	162		114
73	163	29	115
74	164		116
75	165		117
76	166		118
77	167	30	119
78	170		120
79	171		121
7A	172		122
7B	173	31	123
7C	174		124
7D	175		125
7E	176		126
7F	177	32	127
80	200		128
81	201		129
82	202		130
83	203	33	131
84	204		132
85	205		133
86	206		134
87	207	34	135
88	210		136
89	211		137
8A	212		138
8B	213	35	139
8C	214		140
8D	215		141
8E	216		142
8F	217	36	143
90	220		144
91	221		145
92	222		146
93	223	37	147
94	224		148
95	225		149
96	226		150
97	227	38	151
98	230		152
99	231		153
9A	232		154
9B	233	39	155
9C	234		156
9D	235		157
9E	236		158
9F	237	40	159
A0	240		160
A1	241		161
A2	242		162
A3	243	41	163
A4	244		164
A5	245		165
A6	246		166
A7	247	42	167
A8	250		168
A9	251		169
AA	252		170
AB	253	43	171
AC	254		172
AD	255		173
AE	256		174
AF	257	44	175
B0	260		176
B1	261		177

Hex	Oct	K	Bl
B2	262		178
B3	263	45	179
B4	264		180
B5	265		181
B6	266		182
B7	267	46	183
B8	270		184
B9	271		185
BA	272		186
BB	273	47	187
BC	274		188
BD	275		189
BE	276		190
BF	277	48	191
C0	300		192
C1	301		193
C2	302		194
C3	303	49	195
C4	304		196
C5	305		197
C6	306		198
C7	307	50	199
C8	310		200
C9	311		201
CA	312		202
CB	313	51	203
CC	314		204
CD	315		205
CE	316		206
CF	317	52	207
D0	320		208
D1	321		209
D2	322		210
D3	323	53	211
D4	324		212
D5	325		213
D6	326		214
D7	327	54	215
D8	330		216

Hex	Oct	K	Bl
D9	331		217
DA	332		218
DB	333	55	219
DC	334		220
DD	335		221
DE	336		222
DF	337	56	223
E0	340		224
E1	341		225
E2	342		226
E3	343	57	227
E4	344		228
E5	345		229
E6	346		230
E7	347	58	231
E8	350		232
E9	351		233
EA	352		234
EB	353	59	235
EC	354		236
ED	355		237
EE	356		238
EF	357	60	239
F0	360		240
F1	361		241
F2	362		242
F3	363	61	243
F4	364		244
F5	365		245
F6	366		246
F7	367	62	247
F8	370		248
F9	371		249
FA	372		250
FB	373	63	251
FC	374		252
FD	375		253
FE	376		254
FF	377	64	255

APPENDIX C

The 8080 Instruction Set *Alphabetic*

The 8080 instruction set is listed alphabetically with the corresponding hexadecimal code. The following representations apply:

nn	8-bit parameter
nnnn	16-bit parameter

Hex		Mnemonic		Hex		Mnemonic	
CE	nn	ACI	nn	86		ADD	M
8F		ADC	A	C6	nn	ADI	nn
88		ADC	B	A7		ANA	A
89		ADC	C	A0		ANA	B
8A		ADC	D	A1		ANA	C
8B		ADC	E	A2		ANA	D
8C		ADC	H	A3		ANA	E
8D		ADC	L	A4		ANA	H
8E		ADC	M	A5		ANA	L
87		ADD	A	A6		ANA	M
80		ADD	B	E6	nn	ANI	nn
81		ADD	C	CD	nnnn	CALL	nnnn
82		ADD	D	DC	nnnn	CC	nnnn
83		ADD	E	FC	nnnn	CM	nnnn
84		ADD	H	2F		CMA	
85		ADD	L	3F		CMC	

Hex		Mnemonic		Hex		Mnemonic	
BF		CMP	A	24		INR	H
B8		CMP	B	2C		INR	L
B9		CMP	C	34		INR	M
BA		CMP	D	03		INX	B
BB		CMP	E	13		INX	D
BC		CMP	H	23		INX	H
BD		CMP	L	33		INX	SP
BE		CMP	M	DA	nnnn	JC	nnnn
D4	nnnn	CNC	nnnn	FA	nnnn	JM	nnnn
C4	nnnn	CNZ	nnnn	C3	nnnn	JMP	nnnn
F4	nnnn	CP	nnnn	D2	nnnn	JNC	nnnn
EC	nnnn	CPE	nnnn	C2	nnnn	JNZ	nnnn
FE	nn	CPI	nn	F2	nnnn	JP	nnnn
E4	nnnn	CPO	nnnn	EA	nnnn	JPE	nnnn
CC	nnnn	CZ	nnnn	E2	nnnn	JPO	nnnn
27		DAA		CA	nnnn	JZ	nnnn
09		DAD	B	3A	nnnn	LDA	nnnn
19		DAD	D	0A		LDAX	B
29		DAD	H	1A		LDAX	D
39		DAD	SP	2A	nnnn	LHLD	nnnn
3D		DCR	A	01	nnnn	LXI	B,nnnn
05		DCR	B	11	nnnn	LXI	D,nnnn
0D		DCR	C	21	nnnn	LXI	H,nnnn
15		DCR	D	31	nnnn	LXI	SP,nnnn
1D		DCR	E	7F		MOV	A,A
25		DCR	H	78		MOV	A,B
2D		DCR	L	79		MOV	A,C
35		DCR	M	7A		MOV	A,D
0B		DCX	B	7B		MOV	A,E
1B		DCX	D	7C		MOV	A,H
2B		DCX	H	7D		MOV	A,L
3B		DCX	SP	7E		MOV	A,M
F3		DI		47		MOV	B,A
FB		EI		40		MOV	B,B
76		HLT		41		MOV	B,C
DB	nn	IN	nn	42		MOV	B,D
3C		INR	A	43		MOV	B,E
04		INR	B	44		MOV	B,H
0C		INR	C	45		MOV	B,L
14		INR	D	46		MOV	B,M
1C		INR	E	4F		MOV	C,A

Hex	Mnemonic		Hex		Mnemonic	
48	MOV	C,B	71		MOV	M,C
49	MOV	C,C	72		MOV	M,D
4A	MOV	C,D	73		MOV	M,E
4B	MOV	C,E	74		MOV	M,H
4C	MOV	C,H	75		MOV	M,L
4D	MOV	C,L	3E	nn	MVI	A,nn
4E	MOV	C,M	06	nn	MVI	B,nn
57	MOV	D,A	0E	nn	MVI	C,nn
50	MOV	D,B	16	nn	MVI	D,nn
51	MOV	D,C	1E	nn	MVI	E,nn
52	MOV	D,D	26	nn	MVI	H,nn
53	MOV	D,E	2E	nn	MVI	L,nn
54	MOV	D,H	36	nn	MVI	M,nn
55	MOV	D,L	00		NOP	
56	MOV	D,M	B7		ORA	A
5F	MOV	E,A	B0		ORA	B
58	MOV	E,B	B1		ORA	C
59	MOV	E,C	B2		ORA	D
5A	MOV	E,D	B3		ORA	E
5B	MOV	E,E	B4		ORA	H
5C	MOV	E,H	B5		ORA	L
5D	MOV	E,L	B6		ORA	M
5E	MOV	E,M	F6	nn	ORI	nn
67	MOV	H,A	D3	nn	OUT	nn
60	MOV	H,B	E9		PCHL	
61	MOV	H,C	C1		POP	B
62	MOV	H,D	D1		POP	D
63	MOV	H,E	E1		POP	H
64	MOV	H,H	F1		POP	PSW
65	MOV	H,L	C5		PUSH	B
66	MOV	H,M	D5		PUSH	D
6F	MOV	L,A	E5		PUSH	H
68	MOV	L,B	F5		PUSH	PSW
69	MOV	L,C	17		RAL	
6A	MOV	L,D	1F		RAR	
6B	MOV	L,E	D8		RC	
6C	MOV	L,H	C9		RET	
6D	MOV	L,L	07		RLC	
6E	MOV	L,M	F8		RM	
77	MOV	M,A	D0		RNC	
70	MOV	M,B	C0		RNZ	

Hex		Mnemonic		Hex		Mnemonic	
F0		RP		32	nnnn	STA	nnnn
E8		RPE		02		STAX	B
E0		RPO		12		STAX	D
0F		RRC		37		STC	
C7		RST	0	97		SUB	A
CF		RST	1	90		SUB	B
D7		RST	2	91		SUB	C
DF		RST	3	92		SUB	D
E7		RST	4	93		SUB	E
EF		RST	5	94		SUB	H
F7		RST	6	95		SUB	L
FF		RST	7	96		SUB	M
C8		RZ		D6	nn	SUI	nn
9F		SBB	A	EB		XCHG	
98		SBB	B	AF		XRA	A
99		SBB	C	A8		XRA	B
9A		SBB	D	A9		XRA	C
9B		SBB	E	AA		XRA	D
9C		SBB	H	AB		XRA	E
9D		SBB	L	AC		XRA	H
9E		SBB	M	AD		XRA	L
DE	nn	SBI	nn	AE		XRA	M
22	nnnn	SHLD	nnnn	EE	nn	XRI	nn
F9		SPHL		E3		XTHL	

APPENDIX D

The 8080 Instruction Set *Numeric*

The 8080 instruction set is listed numerically with the corresponding hexadecimal code. The following representations apply:

nn	8-bit parameter
nnnn	16-bit parameter

Hex		Mnemonic		Hex		Mnemonic	
00		NOP		16	nn	MVI	D,nn
01	nnnn	LXI	B,nnnn	17		RAL	
02		STAX	B	18		<not used>	
03		INX	B	19		DAD	D
04		INR	B	1A		LDAX	D
05		DCR	B	1B		DCX	D
06	nn	MVI	B,nn	1C		INR	E
07		RLC		1D		DCR	E
08		<not used>		1E	nn	MVI	E,nn
09		DAD	B	1F		RAR	
0A		LDAX	B	20		<not used>	
0B		DCX	B	21	nnnn	LXI	H,nnnn
0C		INR	C	22	nnnn	SHLD	nnnn
0D		DCR	C	23		INX	H
0E	nn	MVI	C,nn	24		INR	H
0F		RRC		25		DCR	H
10		<not used>		26	nn	MVI	H,nn
11	nnnn	LXI	D,nnnn	27		DAA	
12		STAX	D	28		<not used>	
13		INX	D	29		DAD	H
14		INR	D	2A	nnnn	LHLD	nnnn
15		DCR	D	2B		DCX	H

Hex		Mnemonic		Hex	Mnemonic	
2C		INR	L	55	MOV	D,L
2D		DCR	L	56	MOV	D,M
2E	nn	MVI	L,nn	57	MOV	D,A
2F		CMA		58	MOV	E,B
30		<not used>		59	MOV	E,C
31	nnnn	LXI	SP,nnnn	5A	MOV	E,D
32	nnnn	STA	nnnn	5B	MOV	E,E
33		INX	SP	5C	MOV	E,H
34		INR	M	5D	MOV	E,L
35		DCR	M	5E	MOV	E,M
36	nn	MVI	M,nn	5F	MOV	E,A
37		STC		60	MOV	H,B
38		<not used>		61	MOV	H,C
39		DAD	SP	62	MOV	H,D
3A	nnnn	LDA	nnnn	63	MOV	H,E
3B		DCX	SP	64	MOV	H,H
3C		INR	A	65	MOV	H,L
3D		DCR	A	66	MOV	H,M
3E	nn	MVI	A,nn	67	MOV	H,A
3F		CMC		68	MOV	L,B
40		MOV	B,B	69	MOV	L,C
41		MOV	B,C	6A	MOV	L,D
42		MOV	B,D	6B	MOV	L,E
43		MOV	B,E	6C	MOV	L,H
44		MOV	B,H	6D	MOV	L,L
45		MOV	B,L	6E	MOV	L,M
46		MOV	B,M	6F	MOV	L,A
47		MOV	B,A	70	MOV	M,B
48		MOV	C,B	71	MOV	M,C
49		MOV	C,C	72	MOV	M,D
4A		MOV	C,D	73	MOV	M,E
4B		MOV	C,E	74	MOV	M,H
4C		MOV	C,H	75	MOV	M,L
4D		MOV	C,L	76	HLT	
4E		MOV	C,M	77	MOV	M,A
4F		MOV	C,A	78	MOV	A,B
50		MOV	D,B	79	MOV	A,C
51		MOV	D,C	7A	MOV	A,D
52		MOV	D,D	7B	MOV	A,E
53		MOV	D,E	7C	MOV	A,H
54		MOV	D,H	7D	MOV	A,L

Hex	Mnemonic		Hex	Mnemonic	
7E	MOV	A,M	A7	ANA	A
7F	MOV	A,A	A8	XRA	B
80	ADD	B	A9	XRA	C
81	ADD	C	AA	XRA	D
82	ADD	D	AB	XRA	E
83	ADD	E	AC	XRA	H
84	ADD	H	AD	XRA	L
85	ADD	L	AE	XRA	M
86	ADD	M	AF	XRA	A
87	ADD	A	B0	ORA	B
88	ADC	B	B1	ORA	C
89	ADC	C	B2	ORA	D
8A	ADC	D	B3	ORA	E
8B	ADC	E	B4	ORA	H
8C	ADC	H	B5	ORA	L
8D	ADC	L	B6	ORA	M
8E	ADC	M	B7	ORA	A
8F	ADC	A	B8	CMP	B
90	SUB	B	B9	CMP	C
91	SUB	C	BA	CMP	D
92	SUB	D	BB	CMP	E
93	SUB	E	BC	CMP	H
94	SUB	H	BD	CMP	L
95	SUB	L	BE	CMP	M
96	SUB	M	BF	CMP	A
97	SUB	A	C0	RNZ	
98	SBB	B	C1	POP	B
99	SBB	C	C2 nnnn	JNZ	nnnn
9A	SBB	D	C3 nnnn	JMP	nnnn
9B	SBB	E	C4 nnnn	CNZ	nnnn
9C	SBB	H	C5	PUSH	B
9D	SBB	L	C6 nn	ADI	nn
9E	SBB	M	C7	RST	0
9F	SBB	A	C8	RZ	
A0	ANA	B	C9	RET	
A1	ANA	C	CA nnnn	JZ	nnnn
A2	ANA	D	CB	<not used>	
A3	ANA	E	CC nnnn	CZ	nnnn
A4	ANA	H	CD nnnn	CALL	nnnn
A5	ANA	L	CE nn	ACI	nn
A6	ANA	M	CF	RST	1

Hex		Mnemonic		Hex		Mnemonic	
D0		RNC		E8		RPE	
D1		POP	D	E9		PCHL	
D2	nnnn	JNC	nnnn	EA	nnnn	JPE	nnnn
D3	nn	OUT	nn	EB		XCHG	
D4	nnnn	CNC	nnnn	EC	nnnn	CPE	nnnn
D5		PUSH	D	ED		<not used>	
D6	nn	SUI	nn	EE	nn	XRI	nn
D7		RST	2	EF		RST	5
D8		RC		F0		RP	
D9		<not used>		F1		POP	PSW
DA	nnnn	JC	nnnn	F2	nnnn	JP	nnnn
DB	nn	IN	nn	F3		DI	
DC	nnnn	CC	nnnn	F4	nnnn	CP	nnnn
DD		<not used>		F5		PUSH	PSW
DE	nn	SBI	nn	F6	nn	ORI	nn
DF		RST	3	F7		RST	6
E0		RPO		F8		RM	
E1		POP	H	F9		SPHL	
E2	nnnn	JPO	nnnn	FA	nnnn	JM	nnnn
E3		XTHL		FB		EI	
E4	nnnn	CPO	nnnn	FC	nnnn	CM	nnnn
E5		PUSH	H	FD		<not used>	
E6	nn	ANI	nn	FE	nn	CPI	nn
E7		RST	4	FF		RST	7

APPENDIX E

The Z80 Instruction Set *Alphabetic*

The Zilog Z80 instruction set is listed alphabetically with the corresponding hexadecimal values. The following representations apply:

nn	8-bit parameters
nnnn	16-bit parameters
dd	8-bit signed displacement
*	Instructions common to the 8080

Hex		Mnemonic			Hex		Mnemonic		
8E		*	ADC	A,(HL)	81		*	ADD	A,C
DD	8Edd		ADC	A,(IX+dd)	82		*	ADD	A,D
FD	8Edd		ADC	A,(IY+dd)	83		*	ADD	A,E
8F		*	ADC	A,A	84		*	ADD	A,H
88		*	ADC	A,B	85		*	ADD	A,L
89		*	ADC	A,C	C6	nn	*	ADD	A,nn
8A		*	ADC	A,D	09		*	ADD	HL,BC
8B		*	ADC	A,E	19		*	ADD	HL,DE
8C		*	ADC	A,H	29		*	ADD	HL,HL
8D		*	ADC	A,L	39		*	ADD	HL,SP
CE	nn	*	ADC	A,nn	DD	09		ADD	IX,BC
ED	4A		ADC	HL,BC	DD	19		ADD	IX,DE
ED	5A		ADC	HL,DE	DD	29		ADD	IX,IX
ED	6A		ADC	HL,HL	DD	39		ADD	IX,SP
ED	7A		ADC	HL,SP	FD	09		ADD	IY,BC
86		*	ADD	A,(HL)	FD	19		ADD	IY,DE
DD	86dd		ADD	A,(IX+dd)	FD	29		ADD	IY,IY
FD	86dd		ADD	A,(IY+dd)	FD	39		ADD	IY,SP
87		*	ADD	A,A	A6		*	AND	(HL)
80		*	ADD	A,B	DD	A6dd		AND	(IX+dd)

Hex		Mnemonic		Hex		Mnemonic	
FD	A6dd		AND (IY+dd)	CB	5F	BIT	3,A
A7		*	AND A	CB	58	BIT	3,B
A0		*	AND B	CB	59	BIT	3,C
A1		*	AND C	CB	5A	BIT	3,D
A2		*	AND D	CB	5B	BIT	3,E
A3		*	AND E	CB	5C	BIT	3,H
A4		*	AND H	CB	5D	BIT	3,L
A5		*	AND L	CB	66	BIT	4,(HL)
E6	nn	*	AND nn	DD	CBdd66	BIT	4,(IX+dd)
CB	46		BIT 0,(HL)	FD	CBdd66	BIT	4,(IY+dd)
DD	CBdd46		BIT 0,(IX+dd)	CB	67	BIT	4,A
FD	CBdd46		BIT 0,(IY+dd)	CB	60	BIT	4,B
CB	47		BIT 0,A	CB	61	BIT	4,C
CB	40		BIT 0,B	CB	62	BIT	4,D
CB	41		BIT 0,C	CB	63	BIT	4,E
CB	42		BIT 0,D	CB	64	BIT	4,H
CB	43		BIT 0,E	CB	65	BIT	4,L
CB	44		BIT 0,H	CB	6E	BIT	5,(HL)
CB	45		BIT 0,L	DD	CBdd6E	BIT	5,(IX+dd)
CB	4E		BIT 1,(HL)	FD	CBdd6E	BIT	5,(IY+dd)
DD	CBdd4E		BIT 1,(IX+dd)	CB	6F	BIT	5,A
FD	CBdd4E		BIT 1,(IY+dd)	CB	68	BIT	5,B
CB	4F		BIT 1,A	CB	69	BIT	5,C
CB	48		BIT 1,B	CB	6A	BIT	5,D
CB	49		BIT 1,C	CB	6B	BIT	5,E
CB	4A		BIT 1,D	CB	6C	BIT	5,H
CB	4B		BIT 1,E	CB	6D	BIT	5,L
CB	4C		BIT 1,H	CB	76	BIT	6,(HL)
CB	4D		BIT 1,L	DD	CBdd76	BIT	6,(IX+dd)
CB	56		BIT 2,(HL)	FD	CBdd76	BIT	6,(IY+dd)
DD	CBdd56		BIT 2,(IX+dd)	CB	77	BIT	6,A
FD	CBdd56		BIT 2,(IY+dd)	CB	70	BIT	6,B
CB	57		BIT 2,A	CB	71	BIT	6,C
CB	50		BIT 2,B	CB	72	BIT	6,D
CB	51		BIT 2,C	CB	73	BIT	6,E
CB	52		BIT 2,D	CB	74	BIT	6,H
CB	53		BIT 2,E	CB	75	BIT	6,L
CB	54		BIT 2,H	CB	7E	BIT	7,(HL)
CB	55		BIT 2,L	DD	CBdd7E	BIT	7,(IX+dd)
CB	5E		BIT 3,(HL)	FD	CBdd7E	BIT	7,(IY+dd)
DD	CBdd5E		BIT 3,(IX+dd)	CB	7F	BIT	7,A
FD	CBdd5E		BIT 3,(IY+dd)	CB	78	BIT	7,B

Hex		Mnemonic		
CB	79		BIT	7,C
CB	7A		BIT	7,D
CB	7B		BIT	7,E
CB	7C		BIT	7,H
CB	7D		BIT	7,L
DC	nnnn	*	CALL	C,nnnn
FC	nnnn	*	CALL	M,nnnn
D4	nnnn	*	CALL	NC,nnnn
CD	nnnn	*	CALL	nnnn
C4	nnnn	*	CALL	NZ,nnnn
F4	nnnn	*	CALL	P,nnnn
EC	nnnn	*	CALL	PE,nnnn
E4	nnnn	*	CALL	PO,nnnn
CC	nnnn	*	CALL	Z,nnnn
3F		*	CCF	
BE		*	CP	(HL)
DD	BEdd		CP	(IX+dd)
FD	BEdd		CP	(IY+dd)
BF		*	CP	A
B8		*	CP	B
B9		*	CP	C
BA		*	CP	D
BB		*	CP	E
BC		*	CP	H
BD		*	CP	L
FE	nn	*	CP	nn
ED	A9		CPD	
ED	B9		CPDR	
ED	A1		CPI	
ED	B1		CPIR	
2F		*	CPL	
27		*	DAA	
35		*	DEC	(HL)
DD	35dd		DEC	(IX+dd)
FD	35dd		DEC	(IY+dd)
3D		*	DEC	A
05		*	DEC	B
0B		*	DEC	BC
0D		*	DEC	C
15		*	DEC	D
1B		*	DEC	DE
1D		*	DEC	E
25		*	DEC	H
2B		*	DEC	HL
DD	2B		DEC	IX
FD	2B		DEC	IY
2D		*	DEC	L
3B		*	DEC	SP
F3		*	DI	
10	dd		DJNZ	dd
FB		*	EI	
E3		*	EX	(SP),HL
DD	E3		EX	(SP),IX
FD	E3		EX	(SP),IY
08			EX	AF,AF'
EB		*	EX	DE,HL
D9			EXX	
76		*	HALT	
ED	46		IM	0
ED	56		IM	1
ED	5E		IM	2
ED	78		IN	A,(C)
DB	nn	*	IN	A,(nn)
ED	40		IN	B,(C)
ED	48		IN	C,(C)
ED	50		IN	D,(C)
ED	58		IN	E,(C)
ED	60		IN	H,(C)
ED	68		IN	L,(C)
34		*	INC	(HL)
DD	34dd		INC	(IX+dd)
FD	34dd		INC	(IY+dd)
3C		*	INC	A
04		*	INC	B
03		*	INC	BC
0C		*	INC	C
14		*	INC	D
13		*	INC	DE
1C		*	INC	E
24		*	INC	H
23		*	INC	HL
DD	23		INC	IX
FD	23		INC	IY
2C		*	INC	L

Hex			Mnemonic		Hex			Mnemonic	
33		*	INC	SP	FD	71dd		LD	(IY+dd),C
ED	AA		IND		FD	72dd		LD	(IY+dd),D
ED	BA		INDR		FD	73dd		LD	(IY+dd),E
ED	A2		INI		FD	74dd		LD	(IY+dd),H
ED	B2		INIR		FD	75dd		LD	(IY+dd),L
E9		*	JP	(HL)	FD	36ddnn		LD	(IY+dd),nn
DD	E9		JP	(IX)	32	nnnn	*	LD	(nnnn),A
FD	E9		JP	(IY)	ED	43nnnn		LD	(nnnn),BC
DA	nnnn	*	JP	C,nnnn	ED	53nnnn		LD	(nnnn),DE
FA	nnnn	*	JP	M,nnnn	22	nnnn	*	LD	(nnnn),HL
D2	nnnn	*	JP	NC,nnnn	DD	22nnnn		LD	(nnnn),IX
C3	nnnn	*	JP	nnnn	FD	22nnnn		LD	(nnnn),IY
C2	nnnn	*	JP	NZ,nnnn	ED	73nnnn		LD	(nnnn),SP
F2	nnnn	*	JP	P,nnnn	0A		*	LD	A,(BC)
EA	nnnn	*	JP	PE,nnnn	1A		*	LD	A,(DE)
E2	nnnn	*	JP	PO,nnnn	7E		*	LD	A,(HL)
CA	nnnn	*	JP	Z,nnnn	DD	7Edd		LD	A,(IX+dd)
38	dd		JR	C,dd	FD	7Edd		LD	A,(IY+dd)
18	dd		JR	dd	3A	nnnn	*	LD	A,(nnnn)
30	dd		JR	NC,dd	7F		*	LD	A,A
20	dd		JR	NZ,dd	78		*	LD	A,B
28	dd		JR	Z,dd	79		*	LD	A,C
02		*	LD	(BC),A	7A		*	LD	A,D
12		*	LD	(DE),A	7B		*	LD	A,E
77		*	LD	(HL),A	7C		*	LD	A,H
70		*	LD	(HL),B	ED	57		LD	A,I
71		*	LD	(HL),C	7D		*	LD	A,L
72		*	LD	(HL),D	3E	nn	*	LD	A,nn
73		*	LD	(HL),E	ED	5F		LD	A,R
74		*	LD	(HL),H	46		*	LD	B,(HL)
75		*	LD	(HL),L	DD	46dd		LD	B,(IX+dd)
36	nn	*	LD	(HL),nn	FD	46dd		LD	B,(IY+dd)
DD	77dd		LD	(IX+dd),A	47		*	LD	B,A
DD	70dd		LD	(IX+dd),B	40		*	LD	B,B
DD	71dd		LD	(IX+dd),C	41		*	LD	B,C
DD	72dd		LD	(IX+dd),D	42		*	LD	B,D
DD	73dd		LD	(IX+dd),E	43		*	LD	B,E
DD	74dd		LD	(IX+dd),H	44		*	LD	B,H
DD	75dd		LD	(IX+dd),L	45		*	LD	B,L
DD	36ddnn		LD	(IX+dd),nn	06	nn	*	LD	B,nn
FD	77dd		LD	(IY+dd),A	ED	4Bnnnn		LD	BC,(nnnn)
FD	70dd		LD	(IY+dd),B	01	nnnn	*	LD	BC,nnnn

Hex			Mnemonic		Hex			Mnemonic	
4E		*	LD	C,(HL)	63		*	LD	H,E
DD	4Edd		LD	C,(IX+dd)	64		*	LD	H,H
FD	4Edd		LD	C,(IY+dd)	65		*	LD	H,L
4F		*	LD	C,A	26	nn	*	LD	H,nn
48		*	LD	C,B	2A	nnnn	*	LD	HL,(nnnn)
49		*	LD	C,C	21	nnnn	*	LD	HL,nnnn
4A		*	LD	C,D	ED	47		LD	I,A
4B		*	LD	C,E	DD	2Annnn		LD	IX,(nnnn)
4C		*	LD	C,H	DD	21nnnn		LD	IX,nnnn
4D		*	LD	C,L	FD	2Annnn		LD	IY,(nnnn)
0E	nn	*	LD	C,nn	FD	21nnnn		LD	IY,nnnn
56		*	LD	D,(HL)	6E		*	LD	L,(HL)
DD	56dd		LD	D,(IX+dd)	DD	6Edd		LD	L,(IX+dd)
FD	56dd		LD	D,(IY+dd)	FD	6Edd		LD	L,(IY+dd)
57		*	LD	D,A	6F		*	LD	L,A
50		*	LD	D,B	68		*	LD	L,B
51		*	LD	D,C	69		*	LD	L,C
52		*	LD	D,D	6A		*	LD	L,D
53		*	LD	D,E	6B		*	LD	L,E
54		*	LD	D,H	6C		*	LD	L,H
55		*	LD	D,L	6D		*	LD	L,L
16	nn	*	LD	D,nn	2E	nn	*	LD	L,nn
ED	5Bnnnn		LD	DE,(nnnn)	ED	4F		LD	R,A
11	nnnn	*	LD	DE,nnnn	ED	7Bnnnn		LD	SP,(nnnn)
5E		*	LD	E,(HL)	F9		*	LD	SP,HL
DD	5Edd		LD	E,(IX+dd)	DD	F9		LD	SP,IX
FD	5Edd		LD	E,(IY+dd)	FD	F9		LD	SP,IY
5F		*	LD	E,A	31	nnnn	*	LD	SP,nnnn
58		*	LD	E,B	ED	A8		LDD	
59		*	LD	E,C	ED	B8		LDDR	
5A		*	LD	E,D	ED	A0		LDI	
5B		*	LD	E,E	ED	B0		LDIR	
5C		*	LD	E,H	ED	44		NEG	
5D		*	LD	E,L	00		*	NOP	
1E	nn	*	LD	E,nn	B6		*	OR	(HL)
66		*	LD	H,(HL)	DD	B6dd		OR	(IX+dd)
DD	66dd		LD	H,(IX+dd)	FD	B6dd		OR	(IY+dd)
FD	66dd		LD	H,(IY+dd)	B7		*	OR	A
67		*	LD	H,A	B0		*	OR	B
60		*	LD	H,B	B1		*	OR	C
61		*	LD	H,C	B2		*	OR	D
62		*	LD	H,D	B3		*	OR	E

Hex		Mnemonic			Hex		Mnemonic	
B4		*	OR	H	CB	89	RES	1,C
B5		*	OR	L	CB	8A	RES	1,D
F6	nn	*	OR	nn	CB	8B	RES	1,E
ED	BB		OTDR		CB	8C	RES	1,H
ED	B3		OTIR		CB	8D	RES	1,L
ED	79		OUT	(C),A	CB	96	RES	2,(HL)
ED	41		OUT	(C),B	DD	CBdd96	RES	2,(IX+dd)
ED	49		OUT	(C),C	FD	CBdd96	RES	2,(IY+dd)
ED	51		OUT	(C),D	CB	97	RES	2,A
ED	59		OUT	(C),E	CB	90	RES	2,B
ED	61		OUT	(C),H	CB	91	RES	2,C
ED	69		OUT	(C),L	CB	92	RES	2,D
D3	nn	*	OUT	(nn),A	CB	93	RES	2,E
ED	AB		OUTD		CB	94	RES	2,H
ED	A3		OUTI		CB	95	RES	2,L
F1		*	POP	AF	CB	9E	RES	3,(HL)
C1		*	POP	BC	DD	CBdd9E	RES	3,(IX+dd)
D1		*	POP	DE	FD	CBdd9E	RES	3,(IY+dd)
E1		*	POP	HL	CB	9F	RES	3,A
DD	E1		POP	IX	CB	98	RES	3,B
FD	E1		POP	IY	CB	99	RES	3,C
F5		*	PUSH	AF	CB	9A	RES	3,D
C5		*	PUSH	BC	CB	9B	RES	3,E
D5		*	PUSH	DE	CB	9C	RES	3,H
E5		*	PUSH	HL	CB	9D	RES	3,L
DD	E5		PUSH	IX	CB	A6	RES	4,(HL)
FD	E5		PUSH	IY	DD	CBddA6	RES	4,(IX+dd)
CB	86		RES	0,(HL)	FD	CBddA6	RES	4,(IY+dd)
DD	CBdd86		RES	0,(IX+dd)	CB	A7	RES	4,A
FD	CBdd86		RES	0,(IY+dd)	CB	A0	RES	4,B
CB	87		RES	0,A	CB	A1	RES	4,C
CB	80		RES	0,B	CB	A2	RES	4,D
CB	81		RES	0,C	CB	A3	RES	4,E
CB	82		RES	0,D	CB	A4	RES	4,H
CB	83		RES	0,E	CB	A5	RES	4,L
CB	84		RES	0,H	CB	AE	RES	5,(HL)
CB	85		RES	0,L	DD	CBddAE	RES	5,(IX+dd)
CB	8E		RES	1,(HL)	FD	CBddAE	RES	5,(IY+dd)
DD	CBdd8E		RES	1,(IX+dd)	CB	AF	RES	5,A
FD	CBdd8E		RES	1,(IY+dd)	CB	A8	RES	5,B
CB	8F		RES	1,A	CB	A9	RES	5,C
CB	88		RES	1,B	CB	AA	RES	5,D

Hex	Mnemonic	Hex	Mnemonic
CB AB	RES 5,E	CB 14	RL H
CB AC	RES 5,H	CB 15	RL L
CB AD	RES 5,L	17	* RLA
CB B6	RES 6,(HL)	CB 06	RLC (HL)
DD CBddB6	RES 6,(IX+dd)	DD CBdd06	RLC (IX+dd)
FD CBddB6	RES 6,(IY+dd)	FD CBdd06	RLC (IY+dd)
CB B7	RES 6,A .	CB 07	RLC A
CB B0	RES 6,B	CB 00	RLC B
CB B1	RES 6,C	CB 01	RLC C
CB B2	RES 6,D	CB 02	RLC D
CB B3	RES 6,E	CB 03	RLC E
CB B4	RES 6,H	CB 04	RLC H
CB B5	RES 6,L	CB 05	RLC L
CB BE	RES 7,(HL)	07	* RLCA
DD CBddBE	RES 7,(IX+dd)	ED 6F	RLD
FD CBddBE	RES 7,(IY+dd)	CB 1E	RR (HL)
CB BF	RES 7,A	DD CBdd1E	RR (IX+dd)
CB B8	RES 7,B	FD CBdd1E	RR (IY+dd)
CB B9	RES 7,C	CB 1F	RR A
CB BA	RES 7,D	CB 18	RR B
CB BB	RES 7,E	CB 19	RR C
CB BC	RES 7,H	CB 1A	RR D
CB BD	RES 7,L	CB 1B	RR E
C9	* RET	CB 1C	RR H
D8	* RET C	CB 1D	RR L
F8	* RET M	1F	* RRA
D0	* RET NC	CB 0E	RRC (HL)
C0	* RET NZ	DD CBdd0E	RRC (IX+dd)
F0	* RET P	FD CBdd0E	RRC (IY+dd)
E8	* RET PE	CB 0F	RRC A
E0	* RET PO	CB 08	RRC B
C8	* RET Z	CB 09	RRC C
ED 4D	RETI	CB 0A	RRC D
ED 45	RETN	CB 0B	RRC E
CB 16	RL (HL)	CB 0C	RRC H
DD CBdd16	RL (IX+dd)	CB 0D	RRC L
FD CBdd16	RL (IY+dd)	0F	* RRCA
CB 17	RL A	ED 67	RRD
CB 10	RL B	C7	* RST 0
CB 11	RL C	CF	* RST 8
CB 12	RL D	D7	* RST 10H
CB 13	RL E	DF	* RST 18H

Hex		Mnemonic		Hex		Mnemonic	
E7		* RST	20H	FD	CBddD6	SET	2,(IY+dd)
EF		* RST	28H	CB	D7	SET	2,A
F7		* RST	30H	CB	D0	SET	2,B
FF		* RST	38H	CB	D1	SET	2,C
9E		* SBC	A,(HL)	CB	D2	SET	2,D
DD	9Edd	SBC	A,(IX+dd)	CB	D3	SET	2,E
FD	9Edd	SBC	A,(IY+dd)	CB	D4	SET	2,H
9F		* SBC	A,A	CB	D5	SET	2,L
98		* SBC	A,B	CB	DE	SET	3,(HL)
99		* SBC	A,C	DD	CBddDE	SET	3,(IX+dd)
9A		* SBC	A,D	FD	CBddDE	SET	3,(IY+dd)
9B		* SBC	A,E	CB	DF	SET	3,A
9C		* SBC	A,H	CB	D8	SET	3,B
9D		* SBC	A,L	CB	D9	SET	3,C
DE	nn	* SBC	A,nn	CB	DA	SET	3,D
ED	42	SBC	HL,BC	CB	DB	SET	3,E
ED	52	SBC	HL,DE	CB	DC	SET	3,H
ED	62	SBC	HL,HL	CB	DD	SET	3,L
ED	72	SBC	HL,SP	CB	E6	SET	4,(HL)
37		* SCF		DD	CBddE6	SET	4,(IX+dd)
CB	C6	SET	0,(HL)	FD	CBddE6	SET	4,(IY+dd)
DD	CBddC6	SET	0,(IX+dd)	CB	E7	SET	4,A
FD	CBddC6	SET	0,(IY+dd)	CB	E0	SET	4,B
CB	C7	SET	0,A	CB	E1	SET	4,C
CB	C0	SET	0,B	CB	E2	SET	4,D
CB	C1	SET	0,C	CB	E3	SET	4,E
CB	C2	SET	0,D	CB	E4	SET	4,H
CB	C3	SET	0,E	CB	E5	SET	4,L
CB	C4	SET	0,H	CB	EE	SET	5,(HL)
CB	C5	SET	0,L	DD	CBddEE	SET	5,(IX+dd)
CB	CE	SET	1,(HL)	FD	CBddEE	SET	5,(IY+dd)
DD	CBddCE	SET	1,(IX+dd)	CB	EF	SET	5,A
FD	CBddCE	SET	1,(IY+dd)	CB	E8	SET	5,B
CB	CF	SET	1,A	CB	E9	SET	5,C
CB	C8	SET	1,B	CB	EA	SET	5,D
CB	C9	SET	1,C	CB	EB	SET	5,E
CB	CA	SET	1,D	CB	EC	SET	5,H
CB	CB	SET	1,E	CB	ED	SET	5,L
CB	CC	SET	1,H	CB	F6	SET	6,(HL)
CB	CD	SET	1,L	DD	CBddF6	SET	6,(IX+dd)
CB	D6	SET	2,(HL)	FD	CBddF6	SET	6,(IY+dd)
DD	CBddD6	SET	2,(IX+dd)	CB	F7	SET	6,A

Hex	Mnemonic		Hex	Mnemonic	
CB F0	SET	6,B	CB 2C	SRA	H
CB F1	SET	6,C	CB 2D	SRA	L
CB F2	SET	6,D	CB 3E	SRL	(HL)
CB F3	SET	6,E	DD CBdd3E	SRL	(IX+dd)
CB F4	SET	6,H	FD CBdd3E	SRL	(IY+dd)
CB F5	SET	6,L	CB 3F	SRL	A
CB FE	SET	7,(HL)	CB 38	SRL	B
DD CBddFE	SET	7,(IX+dd)	CB 39	SRL	C
FD CBddFE	SET	7,(IY+dd)	CB 3A	SRL	D
CB FF	SET	7,A	CB 3B	SRL	E
CB F8	SET	7,B	CB 3C	SRL	H
CB F9	SET	7,C	CB 3D	SRL	L
CB FA	SET	7,D	96	* SUB	(HL)
CB FB	SET	7,E	DD 96dd	SUB	(IX+dd)
CB FC	SET	7,H	FD 96dd	SUB	(IY+dd)
CB FD	SET	7,L	97	* SUB	A
CB 26	SLA	(HL)	90	* SUB	B
DD CBdd26	SLA	(IX+dd)	91	* SUB	C
FD CBdd26	SLA	(IY+dd)	92	* SUB	D
CB 27	SLA	A	93	* SUB	E
CB 20	SLA	B	94	* SUB	H
CB 21	SLA	C	95	* SUB	L
CB 22	SLA	D	D6 nn	* SUB	nn
CB 23	SLA	E	AE	* XOR	(HL)
CB 24	SLA	H	DD AEdd	XOR	(IX+dd)
CB 25	SLA	L	FD AEdd	XOR	(IY+dd)
CB 2E	SRA	(HL)	AF	* XOR	A
DD CBdd2E	SRA	(IX+dd)	A8	* XOR	B
FD CBdd2E	SRA	(IY+dd)	A9	* XOR	C
CB 2F	SRA	A	AA	* XOR	D
CB 28	SRA	B	AB	* XOR	E
CB 29	SRA	C	AC	* XOR	H
CB 2A	SRA	D	AD	* XOR	L
CB 2B	SRA	E	E nn	* XOR	nn

APPENDIX F

The Z80 Instruction Set *Numeric*

The Z80 instruction set is listed numerically with the corresponding hexadecimal values. The following representations apply:

nn	8-bit parameter
nnnn	16-bit parameter
dd	8-bit signed displacement
*	Instructions common to the 8080

Hex		Mnemonic		Hex		Mnemonic	
00		* NOP		13		* INC	DE
01	nnnn	* LD	BC,nnnn	14		* INC	D
02		* LD	(BC),A	15		* DEC	D
03		* INC	BC	16	nn	* LD	D,nn
04		* INC	B	17		* RLA	
05		* DEC	B	18	dd	JR	dd
06	nn	* LD	B,nn	19		* ADD	HL,DE
07		* RLCA		1A		* LD	A,(DE)
08		EX	AF,AF'	1B		* DEC	DE
09		* ADD	HL,BC	1C		* INC	E
0A		* LD	A,(BC)	1D		* DEC	E
0B		* DEC	BC	1E	nn	* LD	E,nn
0C		* INC	C	1F		* RRA	
0D		* DEC	C	20	dd	JR	NZ,dd
0E	nn	* LD	C,nn	21	nnnn	* LD	HL,nnnn
0F		* RRCA		22	nnnn	* LD	(nnnn),HL
10	dd	DJNZ	dd	23		* INC	HL
11	nnnn	* LD	DE,nnnn	24		* INC	H
12		* LD	(DE),A	25		* DEC	H

Hex		Mnemonic		Hex	Mnemonic	
26	nn	* LD	H,nn	50	* LD	D,B
27		* DAA		51	* LD	D,C
28	dd	JR	Z,dd	52	* LD	D,D
29		* ADD	HL,HL	53	* LD	D,E
2A	nnnn	* LD	HL,(nnnn)	54	* LD	D,H
2B		* DEC	HL	55	* LD	D,L
2C		* INC	L	56	* LD	D,(HL)
2D		* DEC	L	57	* LD	D,A
2E	nn	* LD	L,nn	58	* LD	E,B
2F		* CPL		59	* LD	E,C
30	dd	JR	NC,dd	5A	* LD	E,D
31	nnnn	* LD	SP,nnnn	5B	* LD	E,E
32	nnnn	* LD	(nnnn),A	5C	* LD	E,H
33		* INC	SP	5D	* LD	E,L
34		* INC	(HL)	5E	* LD	E,(HL)
35		* DEC	(HL)	5F	* LD	E,A
36	nn	* LD	(HL),nn	60	* LD	H,B
37		* SCF		61	* LD	H,C
38	dd	JR	C,dd	62	* LD	H,D
39		* ADD	HL,SP	63	* LD	H,E
3A	nnnn	* LD	A,(nnnn)	64	* LD	H,H
3B		* DEC	SP	65	* LD	H,L
3C		* INC	A	66	* LD	H,(HL)
3D		* DEC	A	67	* LD	H,A
3E	nn	* LD	A,nn	68	* LD	L,B
3F		* CCF		69	* LD	L,C
40		* LD	B,B	6A	* LD	L,D
41		* LD	B,C	6B	* LD	L,E
42		* LD	B,D	6C	* LD	L,H
43		* LD	B,E	6D	* LD	L,L
44		* LD	B,H	6E	* LD	L,(HL)
45		* LD	B,L	6F	* LD	L,A
46		* LD	B,(HL)	70	* LD	(HL),B
47		* LD	B,A	71	* LD	(HL),C
48		* LD	C,B	72	* LD	(HL),D
49		* LD	C,C	73	* LD	(HL),E
4A		* LD	C,D	74	* LD	(HL),H
4B		* LD	C,E	75	* LD	(HL),L
4C		* LD	C,H	76	* HALT	
4D		* LD	C,L	77	* LD	(HL),A
4E		* LD	C,(HL)	78	* LD	A,B
4F		* LD	C,A	79	* LD	A,C

Hex	Mnemonic		Hex		Mnemonic	
7A	* LD	A,D	A4		* AND	H
7B	* LD	A,E	A5		* AND	L
7C	* LD	A,H	A6		* AND	(HL)
7D	* LD	A,L	A7		* AND	A
7E	* LD	A,(HL)	A8		* XOR	B
7F	* LD	A,A	A9		* XOR	C
80	* ADD	A,B	AA		* XOR	D
81	* ADD	A,C	AB		* XOR	E
82	* ADD	A,D	AC		* XOR	H
83	* ADD	A,E	AD		* XOR	L
84	* ADD	A,H	AE		* XOR	(HL)
85	* ADD	A,L	AF		* XOR	A
86	* ADD	A,(HL)	B0		* OR	B
87	* ADD	A,A	B1		* OR	C
88	* ADC	A,B	B2		* OR	D
89	* ADC	A,C	B3		* OR	E
8A	* ADC	A,D	B4		* OR	H
8B	* ADC	A,E	B5		* OR	L
8C	* ADC	A,H	B6		* OR	(HL)
8D	* ADC	A,L	B7		* OR	A
8E	* ADC	A,(HL)	B8		* CP	B
8F	* ADC	A,A	B9		* CP	C
90	* SUB	B	BA		* CP	D
91	* SUB	C	BB		* CP	E
92	* SUB	D	BC		* CP	H
93	* SUB	E	BD		* CP	L
94	* SUB	H	BE		* CP	(HL)
95	* SUB	L	BF		* CP	A
96	* SUB	(HL)	C0		* RET	NZ
97	* SUB	A	C1		* POP	BC
98	* SBC	A,B	C2	nnnn	* JP	NZ,nnnn
99	* SBC	A,C	C3	nnnn	* JP	nnnn
9A	* SBC	A,D	C4	nnnn	* CALL	NZ,nnnn
9B	* SBC	A,E	C5		* PUSH	BC
9C	* SBC	A,H	C6	nn	* ADD	A,nn
9D	* SBC	A,L	C7		* RST	0
9E	* SBC	A,(HL)	C8		* RET	Z
9F	* SBC	A,A	C9		* RET	
A0	* AND	B	CA	nnnn	* JP	Z,nnnn
A1	* AND	C	CB	00	RLC	B
A2	* AND	D	CB	01	RLC	C
A3	* AND	E	CB	02	RLC	D

Hex	Mnemonic	Hex	Mnemonic
CB 03	RLC E	CB 2D	SRA L
CB 04	RLC H	CB 2E	SRA (HL)
CB 05	RLC L	CB 2F	SRA A
CB 06	RLC (HL)	CB 38	SRL B
CB 07	RLC A	CB 39	SRL C
CB 08	RRC B	CB 3A	SRL D
CB 09	RRC C	CB 3B	SRL E
CB 0A	RRC D	CB 3C	SRL H
CB 0B	RRC E	CB 3D	SRL L
CB 0C	RRC H	CB 3E	SRL (HL)
CB 0D	RRC L	CB 3F	SRL A
CB 0E	RRC (HL)	CB 40	BIT 0,B
CB 0F	RRC A	CB 41	BIT 0,C
CB 10	RL B	CB 42	BIT 0,D
CB 11	RL C	CB 43	BIT 0,E
CB 12	RL D	CB 44	BIT 0,H
CB 13	RL E	CB 45	BIT 0,L
CB 14	RL H	CB 46	BIT 0,(HL)
CB 15	RL L	CB 47	BIT 0,A
CB 16	RL (HL)	CB 48	BIT 1,B
CB 17	RL A	CB 49	BIT 1,C
CB 18	RR B	CB 4A	BIT 1,D
CB 19	RR C	CB 4B	BIT 1,E
CB 1A	RR D	CB 4C	BIT 1,H
CB 1B	RR E	CB 4D	BIT 1,L
CB 1C	RR H	CB 4E	BIT 1,(HL)
CB 1D	RR L	CB 4F	BIT 1,A
CB 1E	RR (HL)	CB 50	BIT 2,B
CB 1F	RR A	CB 51	BIT 2,C
CB 20	SLA B	CB 52	BIT 2,D
CB 21	SLA C	CB 53	BIT 2,E
CB 22	SLA D	CB 54	BIT 2,H
CB 23	SLA E	CB 55	BIT 2,L
CB 24	SLA H	CB 56	BIT 2,(HL)
CB 25	SLA L	CB 57	BIT 2,A
CB 26	SLA (HL)	CB 58	BIT 3,B
CB 27	SLA A	CB 59	BIT 3,C
CB 28	SRA B	CB 5A	BIT 3,D
CB 29	SRA C	CB 5B	BIT 3,E
CB 2A	SRA D	CB 5C	BIT 3,H
CB 2B	SRA E	CB 5D	BIT 3,L
CB 2C	SRA H	CB 5E	BIT 3,(HL)

Hex	Mnemonic	Hex	Mnemonic
CB 5F	BIT 3,A	CB 89	RES 1,C
CB 60	BIT 4,B	CB 8A	RES 1,D
CB 61	BIT 4,C	CB 8B	RES 1,E
CB 62	BIT 4,D	CB 8C	RES 1,H
CB 63	BIT 4,E	CB 8D	RES 1,L
CB 64	BIT 4,H	CB 8E	RES 1,(HL)
CB 65	BIT 4,L	CB 8F	RES 1,A
CB 66	BIT 4,(HL)	CB 90	RES 2,B
CB 67	BIT 4,A	CB 91	RES 2,C
CB 68	BIT 5,B	CB 92	RES 2,D
CB 69	BIT 5,C	CB 93	RES 2,E
CB 6A	BIT 5,D	CB 94	RES 2,H
CB 6B	BIT 5,E	CB 95	RES 2,L
CB 6C	BIT 5,H	CB 96	RES 2,(HL)
CB 6D	BIT 5,L	CB 97	RES 2,A
CB 6E	BIT 5,(HL)	CB 98	RES 3,B
CB 6F	BIT 5,A	CB 99	RES 3,C
CB 70	BIT 6,B	CB 9A	RES 3,D
CB 71	BIT 6,C	CB 9B	RES 3,E
CB 72	BIT 6,D	CB 9C	RES 3,H
CB 73	BIT 6,E	CB 9D	RES 3,L
CB 74	BIT 6,H	CB 9E	RES 3,(HL)
CB 75	BIT 6,L	CB 9F	RES 3,A
CB 76	BIT 6,(HL)	CB A0	RES 4,B
CB 77	BIT 6,A	CB A1	RES 4,C
CB 78	BIT 7,B	CB A2	RES 4,D
CB 79	BIT 7,C	CB A3	RES 4,E
CB 7A	BIT 7,D	CB A4	RES 4,H
CB 7B	BIT 7,E	CB A5	RES 4,L
CB 7C	BIT 7,H	CB A6	RES 4,(HL)
CB 7D	BIT 7,L	CB A7	RES 4,A
CB 7E	BIT 7,(HL)	CB A8	RES 5,B
CB 7F	BIT 7,A	CB A9	RES 5,C
CB 80	RES 0,B	CB AA	RES 5,D
CB 81	RES 0,C	CB AB	RES 5,E
CB 82	RES 0,D	CB AC	RES 5,H
CB 83	RES 0,E	CB AD	RES 5,L
CB 84	RES 0,H	CB AE	RES 5,(HL)
CB 85	RES 0,L	CB AF	RES 5,A
CB 86	RES 0,(HL)	CB B0	RES 6,B
CB 87	RES 0,A	CB B1	RES 6,C
CB 88	RES 1,B	CB B2	RES 6,D

Hex	Mnemonic	Hex	Mnemonic
CB B3	RES 6,E	CB DD	SET 3,L
CB B4	RES 6,H	CB DE	SET 3,(HL)
CB B5	RES 6,L	CB DF	SET 3,A
CB B6	RES 6,(HL)	CB E0	SET 4,B
CB B7	RES 6,A	CB E1	SET 4,C
CB B8	RES 7,B	CB E2	SET 4,D
CB B9	RES 7,C	CB E3	SET 4,E
CB BA	RES 7,D	CB E4	SET 4,H
CB BB	RES 7,E	CB E5	SET 4,L
CB BC	RES 7,H	CB E6	SET 4,(HL)
CB BD	RES 7,L	CB E7	SET 4,A
CB BE	RES 7,(HL)	CB E8	SET 5,B
CB BF	RES 7,A	CB E9	SET 5,C
CB C0	SET 0,B	CB EA	SET 5,D
CB C1	SET 0,C	CB EB	SET 5,E
CB C2	SET 0,D	CB EC	SET 5,H
CB C3	SET 0,E	CB ED	SET 5,L
CB C4	SET 0,H	CB EE	SET 5,(HL)
CB C5	SET 0,L	CB EF	SET 5,A
CB C6	SET 0,(HL)	CB F0	SET 6,B
CB C7	SET 0,A	CB F1	SET 6,C
CB C8	SET 1,B	CB F2	SET 6,D
CB C9	SET 1,C	CB F3	SET 6,E
CB CA	SET 1,D	CB F4	SET 6,H
CB CB	SET 1,E	CB F5	SET 6,L
CB CC	SET 1,H	CB F6	SET 6,(HL)
CB CD	SET 1,L	CB F7	SET 6,A
CB CE	SET 1,(HL)	CB F8	SET 7,B
CB CF	SET 1,A	CB F9	SET 7,C
CB D0	SET 2,B	CB FA	SET 7,D
CB D1	SET 2,C	CB FB	SET 7,E
CB D2	SET 2,D	CB FC	SET 7,H
CB D3	SET 2,E	CB FD	SET 7,L
CB D4	SET 2,H	CB FE	SET 7,(HL)
CB D5	SET 2,L	CB FF	SET 7,A
CB D6	SET 2,(HL)	CC nnnn	* CALL Z,nnnn
CB D7	SET 2,A	CD nnnn	* CALL nnnn
CB D8	SET 3,B	CE nn	* ADC A,nn
CB D9	SET 3,C	CF	* RST 8
CB DA	SET 3,D	D0	* RET NC
CB DB	SET 3,E	D1	* POP DE
CB DC	SET 3,H	D2 nnnn	* JP NC,nnnn

Hex		Mnemonic			Hex		Mnemonic	
D3	nn	*	OUT	(nn),A	DD	B6dd	OR	(IX+dd)
D4	nnnn	*	CALL	NC,nnnn	DD	BEdd	CP	(IX+dd)
D5		*	PUSH	DE	DD	CBdd06	RLC	(IX+dd)
D6	nn	*	SUB	nn	DD	CBdd0E	RRC	(IX+dd)
D7		*	RST	10H	DD	CBdd16	RL	(IX+dd)
D8		*	RET	C	DD	CBdd1E	RR	(IX+dd)
D9			EXX		DD	CBdd26	SLA	(IX+dd)
DA	nnnn	*	JP	C,nnnn	DD	CBdd2E	SRA	(IX+dd)
DB	nn	*	IN	A,(nn)	DD	CBdd3E	SRL	(IX+dd)
DC	nnnn	*	CALL	C,nnnn	DD	CBdd46	BIT	0,(IX+dd)
DD	09		ADD	IX,BC	DD	CBdd4E	BIT	1,(IX+dd)
DD	19		ADD	IX,DE	DD	CBdd56	BIT	2,(IX+dd)
DD	21nnnn		LD	IX,nnnn	DD	CBdd5E	BIT	3,(IX+dd)
DD	22nnnn		LD	(nnnn),IX	DD	CBdd66	BIT	4,(IX+dd)
DD	23		INC	IX	DD	CBdd6E	BIT	5,(IX+dd)
DD	29		ADD	IX,IX	DD	CBdd76	BIT	6,(IX+dd)
DD	2Annnn		LD	IX,(nnnn)	DD	CBdd7E	BIT	7,(IX+dd)
DD	2B		DEC	IX	DD	CBdd86	RES	0,(IX+dd)
DD	34dd		INC	(IX+dd)	DD	CBdd8E	RES	1,(IX+dd)
DD	35dd		DEC	(IX+dd)	DD	CBdd96	RES	2,(IX+dd)
DD	36ddnn		LD	(IX+dd),nn	DD	CBdd9E	RES	3,(IX+dd)
DD	39		ADD	IX,SP	DD	CBddA6	RES	4,(IX+dd)
DD	46dd		LD	B,(IX+dd)	DD	CBddAE	RES	5,(IX+dd)
DD	4Edd		LD	C,(IX+dd)	DD	CBddB6	RES	6,(IX+dd)
DD	56dd		LD	D,(IX+dd)	DD	CBddBE	RES	7,(IX+dd)
DD	5Edd		LD	E,(IX+dd)	DD	CBddC6	SET	0,(IX+dd)
DD	66dd		LD	H,(IX+dd)	DD	CBddCE	SET	1,(IX+dd)
DD	6Edd		LD	L,(IX+dd)	DD	CBddD6	SET	2,(IX+dd)
DD	70dd		LD	(IX+dd),B	DD	CBddDE	SET	3,(IX+dd)
DD	71dd		LD	(IX+dd),C	DD	CBddE6	SET	4,(IX+dd)
DD	72dd		LD	(IX+dd),D	DD	CBddEE	SET	5,(IX+dd)
DD	73dd		LD	(IX+dd),E	DD	CBddF6	SET	6,(IX+dd)
DD	74dd		LD	(IX+dd),H	DD	CBddFE	SET	7,(IX+dd)
DD	75dd		LD	(IX+dd),L	DD	E1	POP	IX
DD	77dd		LD	(IX+dd),A	DD	E3	EX	(SP),IX
DD	7Edd		LD	A,(IX+dd)	DD	E5	PUSH	IX
DD	86dd		ADD	A,(IX+dd)	DD	E9	JP	(IX)
DD	8Edd		ADC	A,(IX+dd)	DD	F9	LD	SP,IX
DD	96dd		SUB	(IX+dd)	DE	nn	* SBC	A,nn
DD	9Edd		SBC	A,(IX+dd)	DF		* RST	18H
DD	A6dd		AND	(IX+dd)	E0		* RET	PO
DD	AEdd		XOR	(IX+dd)	E1		* POP	HL

Hex		Mnemonic			Hex		Mnemonic		
E2	nnnn	*	JP	PO,nnnn	ED	69		OUT	(C),L
E3		*	EX	(SP),HL	ED	6A		ADC	HL,HL
E4	nnnn	*	CALL	PO,nnnn	ED	6F		RLD	
E5		*	PUSH	HL	ED	72		SBC	HL,SP
E6	nn	*	AND	nn	ED	73nnnn		LD	(nnnn),SP
E7		*	RST	20H	ED	78		IN	A,(C)
E8		*	RET	PE	ED	79		OUT	(C),A
E9		*	JP	(HL)	ED	7A		ADC	HL,SP
EA	nnnn	*	JP	PE,nnnn	ED	7Bnnnn		LD	SP,(nnnn)
EB		*	EX	DE,HL	ED	A0		LDI	
EC	nnnn	*	CALL	PE,nnnn	ED	A1		CPI	
ED	40		IN	B,(C)	ED	A2		INI	
ED	41		OUT	(C),B	ED	A3		OUTI	
ED	42		SBC	HL,BC	ED	A8		LDD	
ED	43nnnn		LD	(nnnn),BC	ED	A9		CPD	
ED	44		NEG		ED	AA		IND	
ED	45		RETN		ED	AB		OUTD	
ED	46		IM	0	ED	B0		LDIR	
ED	47		LD	I,A	ED	B1		CPIR	
ED	48		IN	C,(C)	ED	B2		INIR	
ED	49		OUT	(C),C	ED	B3		OTIR	
ED	4A		ADC	HL,BC	ED	B8		LDDR	
ED	4Bnnnn		LD	BC,(nnnn)	ED	B9		CPDR	
ED	4D		RETI		ED	BA		INDR	
ED	4F		LD	R,A	ED	BB		OTDR	
ED	50		IN	D,(C)	EE	nn	*	XOR	N
ED	51		OUT	(C),D	EF		*	RST	28H
ED	52		SBC	HL,DE	F0		*	RET	P
ED	53nnnn		LD	(nnnn),DE	F1		*	POP	AF
ED	56		IM	1	F2	nnnn	*	JP	P,nnnn
ED	57		LD	A,I	F3		*	DI	
ED	58		IN	E,(C)	F4	nnnn	*	CALL	P,nnnn
ED	59		OUT	(C),E	F5		*	PUSH	AF
ED	5A		ADC	HL,DE	F6	nn	*	OR	nn
ED	5Bnnnn		LD	DE,(nnnn)	F7		*	RST	30H
ED	5E		IM	2	F8		*	RET	M
ED	5F		LD	A,R	F9		*	LD	SP,HL
ED	60		IN	H,(C)	FA	nnnn	*	JP	M,nnnn
ED	61		OUT	(C),H	FB		*	EI	
ED	62		SBC	HL,HL	FC	nnnn	*	CALL	M,nnnn
ED	67		RRD		FD	09		ADD	IY,BC
ED	68		IN	L,(C)	FD	19		ADD	IY,DE

Hex		Mnemonic		Hex		Mnemonic	
FD	21nnnn	LD	IY,nnnn	FD	CBdd1E	RR	(IY+dd)
FD	22nnnn	LD	(nnnn),IY	FD	CBdd26	SLA	(IY+dd)
FD	23	INC	IY	FD	CBdd2E	SRA	(IY+dd)
FD	29	ADD	IY,IY	FD	CBdd3E	SRL	(IY+dd)
FD	2Annnn	LD	IY,(nnnn)	FD	CBdd46	BIT	0,(IY+dd)
FD	2B	DEC	IY	FD	CBdd4E	BIT	1,(IY+dd)
FD	34dd	INC	(IY+dd)	FD	CBdd56	BIT	2,(IY+dd)
FD	35dd	DEC	(IY+dd)	FD	CBdd5E	BIT	3,(IY+dd)
FD	36ddnn	LD	(IY+dd),nn	FD	CBdd66	BIT	4,(IY+dd)
FD	39	ADD	IY,SP	FD	CBdd6E	BIT	5,(IY+dd)
FD	46dd	LD	B,(IY+dd)	FD	CBdd76	BIT	6,(IY+dd)
FD	4Edd	LD	C,(IY+dd)	FD	CBdd7E	BIT	7,(IY+dd)
FD	56dd	LD	D,(IY+dd)	FD	CBdd86	RES	0,(IY+dd)
FD	5Edd	LD	E,(IY+dd)	FD	CBdd8E	RES	1,(IY+dd)
FD	66dd	LD	H,(IY+dd)	FD	CBdd96	RES	2,(IY+dd)
FD	6Edd	LD	L,(IY+dd)	FD	CBdd9E	RES	3,(IY+dd)
FD	70dd	LD	(IY+dd),B	FD	CBddA6	RES	4,(IY+dd)
FD	71dd	LD	(IY+dd),C	FD	CBddAE	RES	5,(IY+dd)
FD	72dd	LD	(IY+dd),D	FD	CBddB6	RES	6,(IY+dd)
FD	73dd	LD	(IY+dd),E	FD	CBddBE	RES	7,(IY+dd)
FD	74dd	LD	(IY+dd),H	FD	CBddC6	SET	0,(IY+dd)
FD	75dd	LD	(IY+dd),L	FD	CBddCE	SET	1,(IY+dd)
FD	77dd	LD	(IY+dd),A	FD	CBddD6	SET	2,(IY+dd)
FD	7Edd	LD	A,(IY+dd)	FD	CBddDE	SET	3,(IY+dd)
FD	86dd	ADD	A,(IY+dd)	FD	CBddE6	SET	4,(IY+dd)
FD	8Edd	ADC	A,(IY+dd)	FD	CBddEE	SET	5,(IY+dd)
FD	96dd	SUB	(IY+dd)	FD	CBddF6	SET	6,(IY+dd)
FD	9Edd	SBC	A,(IY+dd)	FD	CBddFE	SET	7,(IY+dd)
FD	A6dd	AND	(IY+dd)	FD	E1	POP	IY
FD	AEdd	XOR	(IY+dd)	FD	E3	EX	(SP),IY
FD	B6dd	OR	(IY+dd)	FD	E5	PUSH	IY
FD	BEdd	CP	(IY+dd)	FD	E9	JP	(IY)
FD	CBdd06	RLC	(IY+dd)	FD	F9	LD	SP,IY
FD	CBdd0E	RRC	(IY+dd)	FE	nn	* CP	nn
FD	CBdd16	RL	(IY+dd)	FF		* RST	38H

APPENDIX G

Details of the 8080 Instruction Set

A summary of the 8080 instruction set is given in this appendix. The instructions are listed alphabetically by the official Intel mnemonic. The Zilog (Z80) version of the mnemonic is shown in angle brackets.

The letters A, B, C, D, E, H, L, and SP are used for the standard 8080 register names. In addition, the symbols BC, DE, and HL are used for the register pairs. The following symbols are used for general parameters:

r, r2	8-bit CPU register
nn	General 8-bit constant
nnnn	16-bit constant

The flag bits are represented by the following symbols:

C	Carry
H	Half carry
N	Add/subtract
P	Parity
S	Sign
Z	Zero

For the Zilog mnemonic, pointers to memory or input/output addresses are enclosed in parentheses.

ACI nn <ADC A,nn>

Add the constant nn to the accumulator and the carry flag. The result is placed in the accumulator.

Flags affected: C, H, O, S, Z
Flag reset: N

ADC M <ADC A,(HL)>

Add the memory byte referenced by the HL register to the accumulator and the carry flag. The result is placed in the accumulator.

Flags affected: C, H, O, S, Z
Flag reset: N

ADC r <ADC A,r>

Add the value in register r to the accumulator and the carry flag. The result is placed in the accumulator.

Flags affected: C, H, O, S, Z
Flag reset: N

ADD M <ADD A,(HL)>

Add the memory byte referenced by the HL register to the accumulator. The result is placed in the accumulator.

Flags affected: C, H, O, S, Z
Flag reset: N

ADD r <ADD A,r>

Add the value in register r to the accumulator. The result is placed in the accumulator.

Flags affected: C, H, O, S, Z
Flag reset: N

ADI nn <ADD A,nn>

Add the constant nn to the accumulator. The result is placed in A.

Flags affected: C, H, O, S, Z
Flag reset: N

ANA M <AND (HL)>

Perform a logical AND with the accumulator and the memory location referenced by the HL register. The result is placed in the accumulator.

Flags affected: P, S, Z
Flags reset: C, N
Flag set: H

ANA r <AND r>

Perform a logical AND with the accumulator and register r. The result is placed in the accumulator. The instruction ANA A is an effective way to test the parity, sign, and zero flags, because this instruction does not alter the value in A.

Flags affected: P, S, Z
Flags reset: C, N
Flag set: H

ANI nn <AND nn>

Perform a logical AND with the accumulator and the constant given as the parameter. The result is placed in the accumulator. This instruction can be used to selectively reset bits of the accumulator. For example, the instruction ANI 7FH will reset bit 7.

Flags affected: P, S, Z
Flags reset: C, N
Flag set: H

CALL nnnn <CALL nnnn>

Unconditional subroutine call to address nnnn. The address of the following instruction is pushed onto the stack.

Flags affected: none

CC	nnnn	<CALL	C,nnnn>
CM	nnnn	<CALL	M,nnnn>
CNC	nnnn	<CALL	NC,nnnn>
CNZ	nnnn	<CALL	NZ,nnnn>
CP	nnnn	<CALL	P,nnnn>
CPE	nnnn	<CALL	PE,nnnn>
CPO	nnnn	<CALL	PO,nnnn>
CZ	nnnn	<CALL	Z,nnnn>

Conditional subroutine call to address nnnn. The address of the following instruction is pushed onto the stack. The conditions are as follows:

C	Means carry flag set	(Carry)
M	Means sign flag set	(Minus)
NC	Means carry flag reset	(Not carry)
NZ	Means zero flag reset	(Not zero)
P	Means sign flag reset	(Plus)
PE	Means parity flag set	(Parity even)
PO	Means parity flag reset	(Parity odd)
Z	Means zero flag set	(Zero)

CMA <CPL>

Complement the accumulator. This instruction performs a one's complement on the accumulator; that is, each bit that has a value of 0 is changed to 1, and each bit that has a value of 1 is changed to 0.

Flags set: H, N

CMC <CCF>

Complement the carry flag. This instruction can be given after an STC command to reset the carry flag.

Flag affected: C
Flag reset: N

CMP M <CP (HL)>

Compare the byte in memory referenced by the HL register to the accumulator, which is an implied operand. The zero flag is set if the accumulator is equal to the operand. The carry flag is set if the accumulator

is smaller than the operand.

Flags affected: C, H, O, S, Z
Flag set: N

CMP r <CP r>

Compare register r to the accumulator, which is an implied operand. The zero flag is set if the accumulator is equal to the operand. The carry flag is set if the accumulator is smaller than the operand.

Flags affected: C, H, O, S, Z
Flag set: N

CPI nn <CP nn>

Compare the constant given in the operand to the accumulator, which is an implied operand. The zero flag is set if the accumulator is equal to the operand. The carry flag is set if the accumulator is smaller than the operand.

Flags affected: C, H, O, S, Z
Flag set: N

DAA <DAA>

Decimal adjust the accumulator. This instruction is used after each addition with BCD numbers. The Z80 performs this operation properly for both addition and subtraction. The 8080, however, gives an incorrect result for subtraction.

Flags affected: C, H, O, S, Z

DAD B <ADD HL,BC>
DAD D <ADD HL,DE>
DAD H <ADD HL,HL>
DAD SP <ADD HL,SP>

Add the specified double register to the HL register. The result is placed in HL. This is a double-precision addition. The carry flag is set if the result is greater than 16 bits (if overflow occurs). The instruction DAD H performs a 16-bit arithmetic shift left, effectively doubling the HL value. The

DAD SP instruction can be used to save an incoming stack pointer:

```
LXI     H,0
DAD     SP
SHLD    OLDSTK
```

Flags affected: C, H, O, S, Z
Flag reset: N

```
DCR     M       <DEC    (HL)>
```

Decrement the memory byte referenced by the HL register.

Flags affected: H, O, S, Z
Flag set: N
Flag not affected: C

```
DCR     r       <DEC    r>
```

Decrement register r. Do not try to decrement a register past zero while executing a JNC loop. The carry flag is not affected by this operation.

Flags affected: H, O, S, Z
Flag set: N
Flag not affected: C

```
DCX     B       <DEC    BC>
DCX     D       <DEC    DE>
DCX     H       <DEC    HL>
DCX     SP      <DEC    SP>
```

Decrement the indicated double register. Do not try to decrement a double register to zero in a JNZ loop. It will not work because this operation does not affect any of the PSW flags. Instead, move one byte of the double register into the accumulator and perform a logical OR with the other byte:

```
REPEAT:
        . . .
        MOV     A,C
        ORA     B
        JNZ     REPEAT
```

Flags affected: none

DI <DI>

Disable interrupt request.

EI <EI>

Enable interrupt request.

HLT <HALT>

Suspend operation of the CPU until a reset or interrupt occurs.

IN nn <IN A,(nn)>

Input a byte to the accumulator from the port address nn.

Flags affected: none

INR M <INC (HL)>

Increment the memory byte referenced by the HL register.

Flags affected: H, O, S, Z
Flag set: N
Flag not affected: C

INR r <INC r>

Increment the 8-bit register. Do not try to increment a register past zero while executing a JNC loop. It will not work because the carry flag is unaffected by this instruction.

Flags affected: H, O, S, Z
Flag set: N
Flag not affected: C

INX B <INC BC>
INX D <INC DE>
INX H <INC HL>
INX SP <INC SP>

Increment the specified double register.

Flags affected: none

```
JMP    nnnn        <JP    nnnn>
```

Unconditional jump to address nnnn.

Flags affected: none

```
JC     nnnn        <JP    C,nnnn>
JM     nnnn        <JP    M,nnnn>
JNC    nnnn        <JP    NC,nnnn>
JNZ    nnnn        <JP    NZ,nnnn>
JP     nnnn        <JP    P,nnnn>
JPE    nnnn        <JP    PE,nnnn>
JPO    nnnn        <JP    PO,nnnn>
JZ     nnnn        <JP    Z,nnnn>
```

Conditional jump to address nnnn where:

C	Means carry flag set	(Carry)
M	Means sign flag set	(Minus)
NC	Means carry flag reset	(Not carry)
NZ	Means zero flag reset	(Not zero)
P	Means sign flag reset	(Plus)
PE	Means parity flag set	(Parity even)
PO	Means parity flag reset	(Parity odd)
Z	Means zero flag set	(Zero)

```
LDA    nnnn        <LD    A,(nnnn)>
```

Load the accumulator from the memory byte referenced by the 16-bit pointer nnnn.

```
LDAX   B           <LD    A,(BC)>
LDAX   D           <LD    A,(DE)>
```

Move the memory byte referenced by the specified double register BC or DE into the accumulator. (See STAX B.)

```
LHLD   nnnn        <LD    HL,(nnnn)>
```

Load register L from the address referenced by the 16-bit value nnnn.

Load register H from the address nnnn + 1.

LXI	B,nnnn	<LD	BC,nnnn>
LXI	D,nnnn	<LD	DE,nnnn>
LXI	H,nnnn	<LD	HL,nnnn>
LXI	SP,nnnn	<LD	SP,nnnn>

Load the specified double register with the 16-bit constant nnnn.

MOV M,r <LD (HL),r>

Move the byte in register r to the memory byte referenced by the HL register.

MOV r,M <LD r,(HL)>

Move the byte referenced by the HL register into register r.

MOV r,r2 <LD r,r2>

Move the byte from register r2 to r.

MVI M,nn <LD (HL),nn>

Move the data byte nn into the memory location referenced by the HL register.

MVI r,nn <LD r,nn>

Load register r with the 8-bit data byte nn.

NOP <NOP>

No operation is performed by the CPU.

Flags affected: none

ORA M <OR (HL)>

Perform a logical OR with the accumulator and the memory byte referenced by the HL register. The result is placed in the accumulator.

Flags affected: P, S, Z
Flags reset: C, H, N

```
ORA    r        <OR    r>
```

Perform a logical OR with the accumulator and register r. The result is placed in the accumulator. An instruction of ORA A is an efficient way to test the parity, sign, and zero flags, because this instruction does not alter the value in A.

Flags affected: P, S, Z
Flags reset: C, H, N

```
ORI    nn       <OR    nn>
```

Perform a logical OR with the accumulator and the data byte nn. The result is placed in the accumulator. This instruction can be used to set individual bits of the accumulator. For example, ORI 20H will set bit 5 to a logical 1.

Flags affected: P, S, Z
Flags reset: C, H, N

```
OUT    nn       <OUT   (nn),A>
```

Output the byte in the accumulator to the port address nn.

Flags affected: none

```
PCHL       <JP    (HL)>
```

Copy the HL register into the program counter, then retrieve the next instruction from the address referenced by HL. This instruction causes a branch to the address referenced by HL.

Flags affected: none

```
POP    B        <POP   BC>
POP    D        <POP   DE>
POP    H        <POP   HL>
```

Copy two bytes of memory into the appropriate double register as follows. The memory byte referenced by the stack pointer is moved into

the low-order byte (C, E, or L), then the stack pointer is incremented. The memory byte referenced by the new stack-pointer value is then moved into the high-order byte (B, D, or H). The stack pointer is incremented a second time.

Flags affected: none

```
POP   PSW        <POP   AF>
```

Move the byte at the memory location referenced by the stack pointer into the flag register (PSW), and increment the stack pointer. Then move the byte at the location referenced by the new stack-pointer value into the accumulator and increment the stack pointer a second time.

Flags affected: all

```
PUSH  B          <PUSH  BC>
PUSH  D          <PUSH  DE>
PUSH  H          <PUSH  HL>
```

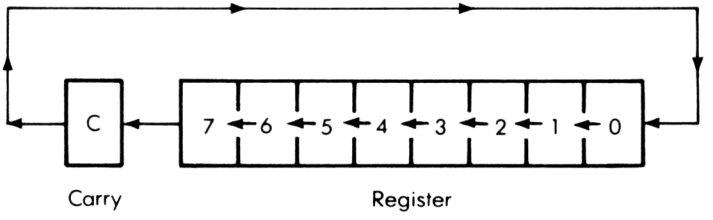
Store the referenced double register in memory as follows. The stack pointer is decremented, then the byte in the specified high-order register B, D, or H is copied to the memory location referenced by the stack pointer. The stack pointer is decremented a second time. The byte in the low-order register C, E, or L is moved to the byte referenced by the current value of the stack pointer.

Flags affected: none

```
PUSH  PSW        <PUSH  AF>
```

Store the accumulator and flag register in memory as follows. The stack pointer is decremented, then the value in the accumulator is moved to the memory byte referenced by the stack pointer. The stack pointer is decremented a second time. The flag register is copied to the byte at the memory address referenced by the current stack-pointer value.

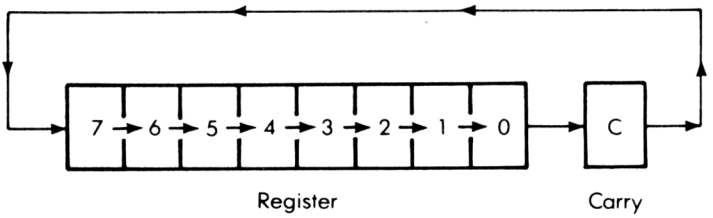
Flags affected: none



RAL <RLA>

This instruction rotates bits to the left through carry by one position. The byte in the accumulator is rotated left through carry. The carry flag moves to bit 0. Bit 7 of the accumulator moves to the carry flag.

Flags affected: C
Flags reset: H, N



RAR <RRA>

This instruction rotates bits to the right through carry by one position. The accumulator is rotated right through carry. The carry flag moves to bit 7. Bit 0 moves to the carry flag.

Flag affected: C
Flags reset: H, N

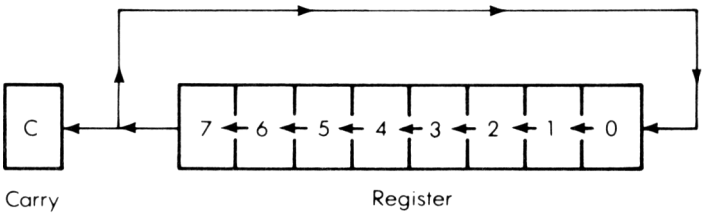
RET <RET>

Return from a subroutine. The top of the stack is moved into the program counter. The stack pointer is incremented twice.

RC	<RET	C>
RM	<RET	M>
RNC	<RET	NC>
RNZ	<RET	NZ>
RP	<RET	P>
RPE	<RET	PE>
RPO	<RET	PO>
RZ	<RET	Z>

Conditional return from a subroutine. If the condition is met, the top of the stack is moved into the program counter. The stack pointer is incremented twice.

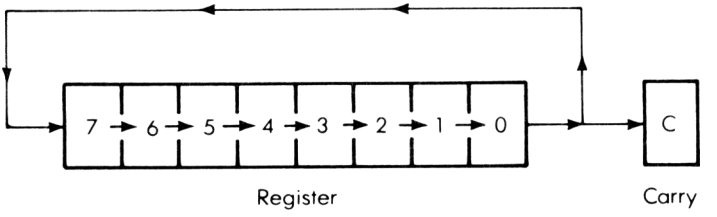
C	Means carry flag set	(Carry)
M	Means sign flag set	(Minus)
NC	Means carry flag reset	(Not carry)
NZ	Means zero flag reset	(Not zero)
P	Means sign flag reset	(Plus)
PE	Means parity flag set	(Parity even)
PO	Means parity flag reset	(Parity odd)
Z	Means zero flag set	(Zero)



RLC <RLCA>

This instruction rotates bits to the left by one position. The accumulator is rotated left circularly. Bit 7 moves to both the zero bit and the carry flag.

Flags affected: C
Flags reset: H, N



RRC <RRCA>

This instruction rotates bits to the right by one position. The accumulator is rotated right circularly. Bit 0 moves to both the carry flag and bit 7.

Flag affected: C
Flags reset: H, N

RST	0	<RST	00H>
RST	1	<RST	08H>
RST	2	<RST	10H>
RST	3	<RST	18H>
RST	4	<RST	20H>
RST	5	<RST	28H>
RST	6	<RST	30H>
RST	7	<RST	38H>

These restart instructions generate one-byte subroutine calls to the address given in the Z80 operand. For example, RST 7 calls address 38 hex.

SBB M <SBC A,(HL)>

Subtract the carry flag and the memory byte referenced by the HL register from the accumulator. The result is placed in the accumulator.

Flags affected: C, H, O, S, Z
Flag set: N

SBB r <SBC A,r>

Subtract the carry flag and the specified CPU register from the accumulator. The result is placed in the accumulator.

Flags affected: C, H, O, S, Z
Flag set: N

SBI nn <SBC A,nn>

Subtract the data byte nn and the carry flag from the accumulator. The result is placed in the accumulator.

Flags affected: C, H, O, S, Z
Flag set: N

SHLD nnnn <LD (nnnn),HL>

Store register L at the memory address nnnn. Store register H at the address nnnn + 1.

SPHL <LD SP,HL>

Load the stack pointer from the HL register. This instruction can be used to retrieve a previously saved stack pointer.

```
LHLD  nnnn
SPHL
```

STA nnnn <LD (nnnn),A>

Store the accumulator in the memory location referenced by nnnn.

STAX B <LD (BC),A>
STAX D <LD (DE),A>

Move the byte in the accumulator to the memory byte referenced by the specified register pair. (See LDAX B.)

STC <SCF>

Set the carry flag. There is no equivalent reset command. However, the carry flag can be reset with the XRA A instruction or with the pair of instructions STC and CMC.

Flag set: C
Flags reset: H, N

SUB M <SUB (HL)>

Subtract the memory byte referenced by the HL register from the accumulator. The result is placed in the accumulator.

Flags affected: C, H, O, S, Z
Flag set: N

SUB r <SUB r>

Subtract the specified CPU register from the accumulator. The result is placed in the accumulator.

Flags affected: C, H, O, S, Z
Flag set: N

SUI nn <SUB nn>

Subtract the data byte nn from the accumulator. The result is placed in the accumulator.

Flags affected: C, H, O, S, Z
Flag set: N

XCHG <EX DE,HL>

Exchange the double registers DE and HL.

Flags affected: none

XRA M <XOR (HL)>

Perform a logical exclusive OR with the accumulator and the byte referenced by the HL register. The result is placed in the accumulator.

Flags affected: P, S, Z
Flags reset: C, H, N

XRA r <XOR r>

Perform a logical exclusive OR with the accumulator and register r. The result is placed in the accumulator. The XRA A instruction is an efficient way to zero the accumulator, although all flags are then reset. XRA A is also frequently used to reset the carry flag, because there is no single

instruction for this operation.

Flags affected: P, S, Z
Flags reset: C, H, N

XRI nn <XOR nn>

Perform a logical exclusive OR with the accumulator and the data byte nn. The result is placed in the accumulator.

Flags affected: P, S, Z
Flags reset: C, H, N

XTHL <EX (SP),HL>

Exchange the byte in memory referenced by the stack pointer with register L. Exchange the byte referenced by the stack pointer + 1 with register H.

Flags affected: none

APPENDIX H

Details of the Z80 Instruction Set

A summary of the Z80 instruction set is given in this appendix.* The instructions are listed alphabetically by the official Zilog mnemonic. If there is a corresponding 8080 instruction, the Intel mnemonic is shown in angle brackets; refer to Appendix G for the details of this instruction. If there is no 8080 equivalent, "no 8080" is shown in angle brackets. The Z80 mnemonics are listed in numeric order in Appendix F. The Z80 equivalent of an 8080 mnemonic can be found from the cross reference given in Appendix G.

The letters A, B, C, D, E, H, I, L, IX, IY, R, and SP are used for the standard Z80 register names. In addition, the symbols BC, DE, and HL are used for the register pairs. The following symbols are used for general parameters:

r, r2	8-bit CPU register
dd	8-bit signed displacement
nn	General 8-bit constant
nnnn	16-bit constant

The flag bits are represented by the following symbols:

C	Carry
H	Half carry
N	Add/subtract
P/O	Parity/overflow
S	Sign
Z	Zero

Pointers to memory and input/output addresses are enclosed in parentheses.

---

*More details can be be obtained from the Zilog programmer's manual, *Z80 Assembly Language Programming Manual,* Zilog, Inc., 1977.

```
ADC    A,(HL)        <ADC    M>
```

```
ADC    A,(IX+dd)     <no 8080>
ADC    A,(IY+dd)     <no 8080>
```

Add the memory byte referenced by the sum of the specified index register and the displacement to the accumulator and the carry flag. The result is placed in the accumulator.

Flags affected: C, H, O, S, Z
Flag reset: N

```
ADC    A,r           <ADC    r>
```

```
ADC    A,nn          <ACI    nn>
```

```
ADC    HL,BC         <no 8080>
ADC    HL,DE         <no 8080>
ADC    HL,HL         <no 8080>
ADC    HL,SP         <no 8080>
```

Add the indicated double register to the HL register and the carry flag. The result is placed in HL.

Flags affected: C, H, O, S, Z
Flag reset: N

```
ADD    A,(HL)        <ADD    M>
```

```
ADD    A,(IX+dd)     <no 8080>
ADD    A,(IY+dd)     <no 8080>
```

Add the memory byte pointed to by the sum of the specified index register and the displacement to the accumulator. The result is placed in the accumulator.

Flags affected: C, H, O, S, Z
Flag reset: N

```
ADD    A,r           <ADD    r>
```

```
ADD   A,nn      <ADI   nn>
```

```
ADD   HL,BC     <DAD   B>
ADD   HL,DE     <DAD   D>
ADD   HL,HL     <DAD   H>
ADD   HL,SP     <DAD   SP>
```

```
ADD   IX,BC     <no 8080>
ADD   IX,DE     <no 8080>
ADD   IX,IX     <no 8080>
ADD   IX,SP     <no 8080>
ADD   IY,BC     <no 8080>
ADD   IY,DE     <no 8080>
ADD   IY,IY     <no 8080>
ADD   IY,SP     <no 8080>
```

Add the indicated double register to the specified index register. The result is placed in the index register. The HL register pair does not participate in this group of instructions. Notice that there is no equivalent series of ADC instructions.

Flags affected: C, O, S, Z
Flag reset: N

```
AND   (HL)      <ANA   M>
```

```
AND   (IX+dd)     <no 8080>
AND   (IY+dd)     <no 8080>
```

Perform a logical AND with the accumulator and the memory byte referenced by the sum of the index register and the displacement. The result is placed in the accumulator.

Flags affected: P, S, Z
Flags reset: C, N
Flag set: H

```
AND   r         <ANA   r>
```

```
AND   nn        <ANI   nn>
```

```
BIT     b,(HL)          <no 8080>
BIT     b,(IX+dd)       <no 8080>
BIT     b,(IY+dd)       <no 8080>
```

Test bit b of the memory byte referenced by the second operand. Bit b can range from 0 through 7. The zero flag is set if the referenced bit is a logical 1, otherwise it is reset. Thus the zero flag becomes the complement of the selected bit.

Flag affected: Z
Flag set: H
Flag reset: N

```
BIT     b,r         <no 8080>
```

Test bit b of register r, where b can range from 0 through 7. The zero flag is set if the referenced bit is a logical 1. It is reset otherwise.

Flag affected: Z
Flag set: H
Flag reset: N

```
CALL    nnnn        <CALL       nnnn>
```

```
CALL    C,nnnn          <CC     nnnn>
CALL    M,nnnn          <CM     nnnn>
CALL    NC,nnnn         <CNC    nnnn>
CALL    NZ,nnnn         <CNZ    nnnn>
CALL    P,nnnn          <CP     nnnn>
CALL    PE,nnnn         <CPE    nnnn>
CALL    PO,nnnn         <CPO    nnnn>
CALL    Z,nnnn          <CZ     nnnn>
```

```
CCF     <CMC>
```

```
CP      (HL)        <CMP  M>
```

```
CP      (IX+dd)     <no 8080>
CP      (IY+dd)     <no 8080>
```

Compare the memory location referenced by the sum of the index register

and the displacement to the accumulator, which is an implied operand. The zero flag is set if the accumulator is equal to the operand. The carry flag is set if the accumulator is smaller than the operand.

Flags affected: C, H, O, S, Z
Flag set: N

CP r <CMP r>

CP nn <CPI nn>

CPD <no 8080>
CPDR <no 8080>
CPI <no 8080>
CPIR <no 8080>

Compare the memory byte pointed to by HL to the accumulator. Decrement HL (if D) or increment HL (if I). Decrement the byte count in the BC register. Repeat the operation for CPDR and CPIR until a match is found or until the BC register pair has been decremented to zero. The zero flag is set if a match is found. The parity flag is set if BC is decremented to 0.

Flags affected: H, S
Flag set: N, Z if A = (HL), P if BC = 0

CPL <CMA>

DAA <DAA>

DEC (HL) <DCR M>

DEC (IX+dd) <no 8080>
DEC (IY+dd) <no 8080>

Decrement the memory byte pointed to by the sum of the index register and the displacement.

Flags affected: H, O, S, Z
Flag set: N
Flag not affected: C

```
DEC    r        <DCR    r>
```

```
DEC    BC       <DCX    B>
DEC    DE       <DCX    D>
DEC    HL       <DCX    H>
DEC    SP       <DCX    SP>
```

```
DEC    IX       <no 8080>
DEC    IY       <no 8080>
```

Decrement the index register.

Flags affected: none

```
DI     <DI>
```

```
DJNZ   dd       <no 8080>
```

Decrement register B and jump relative to displacement dd if B register is not 0.

Flags affected: none

```
EI     <EI>
```

```
EX     (SP),HL      <XTHL>
```

```
EX     (SP),IX      <no 8080>
EX     (SP),IY      <no 8080>
```

Exchange the 16 bits referenced by the stack pointer with the specified index register.

Flags affected: none

```
EX     AF,AF'       <no 8080>
```

Exchange the accumulator and flag register with the alternate set.

Flags affected: all

```
EX    DE,HL        <XCHG>
```

```
EXX        <no 8080>
```

Exchange BC, DE, and HL with the alternate set.

Flags affected: none

```
HALT       <HLT>
```

```
IM    0        <no 8080>
IM    1        <no 8080>
IM    2        <no 8080>
```

Sets interrupt mode 0, 1, or 2. Interrupt mode 0 is automatically selected when a Z80 reset occurs. The result is the same as the 8080 interrupt response. Interrupt mode 1 performs an RST 38H instruction. Interrupt mode 2 provides for many interrupt locations.

```
IN    r,(C)        <no 8080>
```

Input a byte from the port address in register C to register r.

Flags affected: P, S, Z
Flags reset: H, N

```
IN    A,(nn)       <IN    nn>
```

```
INC   (HL)         <INR   M>
```

```
INC   (IX+dd)      <no 8080>
INC   (IY+dd)      <no 8080>
```

Increment the memory byte pointed to by the sum of the index register and the displacement.

Flags affected: H, O, S, Z
Flag set: N
Flag not affected: C

```
INC    r        <INR    r>
```

```
INC    BC       <INX    B>
INC    DE       <INX    D>
INC    HL       <INX    H>
INC    SP       <INX    SP>
```

```
INC    IX       <no 8080>
INC    IY       <no 8080>
```

Increment the specified index register.

Flags affected: none

```
IND             <no 8080>
INDR            <no 8080>
INI             <no 8080>
INIR            <no 8080>
```

Input a byte from the port address in register C to the memory byte pointed to by HL. Decrement register B. The HL register is incremented (if I) or decremented (if D). For INDR and INIR the process is repeated until the 8-bit register B becomes 0.

Flag affected: Z (if B = 0)
Flag set: N

```
JP     (HL)     <PCHL>
```

```
JP     (IX)     <no 8080>
JP     (IY)     <no 8080>
```

Copy the contents of the specified index register into the program counter; then retrieve the next instruction from the address referenced by IX or IY.

Flags affected: none

```
JP     nnnn     <JMP    nnnn>
```

```
JP   C,nnnn      <JC    nnnn>
JP   M,nnnn      <JM    nnnn>
JP   NC,nnnn     <JNC   nnnn>
JP   NZ,nnnn     <JNZ   nnnn>
JP   P,nnnn      <JP    nnnn>
JP   PE,nnnn     <JPE   nnnn>
JP   PO,nnnn     <JPO   nnnn>
JP   Z,nnnn      <JZ    nnnn>
```

```
JR   nn      <no 8080>
```

Unconditional relative jump with a signed displacement nn. The jump is limited to 129 bytes forward and 126 bytes backward in memory.

Flags affected: none

```
JR   C,nn        <no 8080>
JR   NC,nn       <no 8080>
JR   NZ,nn       <no 8080>
JR   Z,nn        <no 8080>
```

Conditional relative jump to address nn where:

C	Means carry flag set	(Carry)
NC	Means carry flag reset	(Not carry)
NZ	Means zero flag reset	(Not zero)
Z	Means zero flag set	(Zero)

```
LD   (BC),A      <STAX  B>
LD   (DE),A      <STAX  D>
```

```
LD   (HL),r      <MOV   M,r>
```

```
LD   (HL),nn     <MVI   M,nn>
```

```
LD   (IX+dd),r       <no 8080>
LD   (IX+dd),nn      <no 8080>
LD   (IY+dd),r       <no 8080>
LD   (IY+dd),nn      <no 8080>
```

Move the byte in register r or the immediate byte nn into the memory byte referenced by the sum of the index register plus the displacement. These instructions can be used to load relocatable binary code.

```
LD   (nnnn),A      <STA   nnnn>
```

```
LD   (nnnn),BC     <no 8080>
LD   (nnnn),DE     <no 8080>
```

Store the low-order byte (C or E) of the specified double register at the memory location nnnn. Store the high-order byte (B or D) at nnnn + 1.

```
LD   (nnnn),HL     <SHLD   nnnn>
```

```
LD   (nnnn),IX     <no 8080>
LD   (nnnn),IY     <no 8080>
LD   (nnnn),SP     <no 8080>
```

Store the low-order byte of the specified register IX, IY, or SP at the location nnnn. Store the high-order byte at nnnn + 1. The instruction LD (nnnn),SP can be used to temporarily save an incoming stack pointer. It can later be restored by an LD SP,(nnnn) operation.

```
LD   A,(BC)        <LDAX   B>
LD   A,(DE)        <LDAX   D>
```

```
LD   A,I       <no 8080>
```

Load the accumulator from the interrupt-vector register. The parity flag reflects the state of the interrupt-enable flip-flop.

Flags affected: P, S, Z
Flags reset: H, N

```
LD   A,R       <no 8080>
```

Load the accumulator from the memory-refresh register. The parity flag reflects the state of the interrupt-enable flip-flop. This is an easy way to obtain a fairly decent random number.

Flags affected: P, S, Z
Flags reset: H, N

```
LD   I,A        <no 8080>
```

Copy the accumulator into the interrupt-vector register.

Flags affected: none

```
LD   R,A        <no 8080>
```

Copy the accumulator into the memory-refresh register.

Flags affected: none

```
LD   r,(HL)        <MOV   r,M>
```

```
LD   r,(IX+dd)        <no 8080>
LD   r,(IY+dd)        <no 8080>
```

Move the byte at the memory location referenced by the sum of the index register and the displacement into register r.

```
LD   r,r2        <MOV   r,r2>
```

```
LD   r,nn        <MVI   r,nn>
```

```
LD   A,(nnnn)        <LDA   nnnn>
```

```
LD   BC,(nnnn)        <no 8080>
LD   DE,(nnnn)        <no 8080>
```

Load the low-order byte (C or E) from the location referenced by the 16-bit pointer nnnn. Load the high-order byte (B or D) from nnnn + 1.

```
LD   HL,(nnnn)        <LHLD   nnnn>
```

```
LD   BC,nnnn        <LXI   B,nnnn>
LD   DE,nnnn        <LXI   D,nnnn>
LD   HL,nnnn        <LXI   H,nnnn>
LD   SP,nnnn        <LXI   SP,nnnn>
LD   IX,nnnn        <no 8080>
LD   IY,nnnn        <no 8080>
```

Load the specified double register with the 16-bit constant nnnn. Be careful not to confuse LD HL,(nnnn) with LD HL,nnnn.

```
LD   IX,(nnnn)      <no 8080>
LD   IY,(nnnn)      <no 8080>
LD   SP,(nnnn)      <no 8080>
```

Load the low byte of IX, IY, or SP from the memory location nnnn. Load the high byte from nnnn + 1. The LD SP,(nnnn) instruction can be used to retrieve a previously saved stack pointer.

```
LD   SP,HL      <SPHL>
LD   SP,IX      <no 8080>
LD   SP,IY      <no 8080>
```

Load the stack pointer from the specified 16-bit register. The SPHL instruction can be used to retrieve a previously saved stack pointer when the 8080 CPU is used.

```
LHLD   nnnn
SPHL
```

```
LDD     <no 8080>
LDDR    <no 8080>
LDI     <no 8080>
LDIR    <no 8080>
```

Move the byte referenced by the HL pair into the location pointed to by the DE register pair. Decrement the 16-bit byte counter in BC. Increment (if I) or decrement (if D) both HL and DE. Repeat the operation for LDDR and LDIR until the BC register has been decremented to zero.

```
NEG     <no 8080>
```

This instruction performs a two's complement on the accumulator. It effectively subtracts the accumulator from zero. To perform this task on an 8080 use a CMA command followed by an INR A command.

Flags affected: all

```
NOP     <NOP>
```

OR (HL) <ORA M>

OR (IX+dd) <no 8080>
OR (IY+dd) <no 8080>

Perform a logical OR with the accumulator and the byte referenced by the specified index register plus the displacement. The result is placed in the accumulator.

Flags affected: P, S, Z
Flags reset: C, H, N

OR r <ORA r>

OR nn <ORI nn>

OTDR <no 8080>
OTIR <no 8080>

Output a byte from the memory location pointed to by the HL pair. The port address is contained in register C. Register B is decremented. The HL register pair is incremented (if I) or decremented (if D). The process is repeated until register B has become zero.

Flags set: N, Z

OUT (C),r <no 8080>

Output the byte in register r to the port address contained in register C.

Flags affected: none

OUT (nn),A <OUT nn>

OUTD <no 8080>
OUTI <no 8080>

Output a byte from the memory location pointed to by the HL pair. The port address is contained in register C. Register B is decremented. The HL register pair is incremented (if I) or decremented (if D).

Flag affected: Z
Flag set: N

```
POP   AF      <POP   PSW>
```

```
POP   BC      <POP   B>
POP   DE      <POP   D>
POP   HL      <POP   H>
```

```
POP   IX      <no 8080>
POP   IY      <no 8080>
```

Copy the top of the stack into the specified index register. Increment the stack pointer twice.

Flags affected: none

```
PUSH  AF      <PUSH  PSW>
```

```
PUSH  BC      <PUSH  B>
PUSH  DE      <PUSH  D>
PUSH  HL      <PUSH  H>
```

```
PUSH  IX      <no 8080>
PUSH  IY      <no 8080>
```

The indicated index register is copied to the top of the stack. The stack pointer is decremented twice.

Flags affected: none

```
RES   b,(HL)          <no 8080>
RES   b,(IX+dd)       <no 8080>
RES   b,(IY+dd)       <no 8080>
```

Reset bit b of the memory byte referenced by the second operand. Bit b can range from 0 through 7.

Flags affected: none
Flag reset: N

RES b,r <no 8080>

Reset bit b of register r to a value of 0. Bit b can range from 0 through 7.

Flags affected: none
Flag reset: N

RET <RET>

RET C <RC>
RET M <RM>
RET NC <RNC>
RET NZ <RNZ>
RET P <RP>
RET PE <RPE>
RET PO <RPO>
RET Z <RZ>

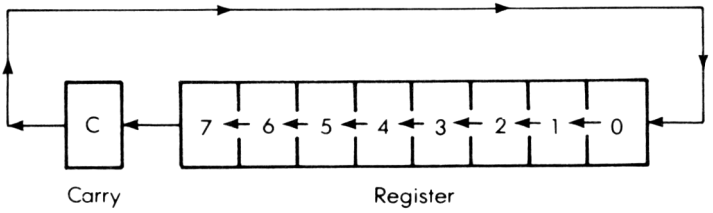
RETI <no 8080>

Return from maskable interrupt.

RETN <no 8080>

Return from nonmaskable interrupt.

The following RL and RLA instructions rotate bits to the left through carry.



RL (HL) <no 8080>

The memory byte referenced by the HL pair is rotated left through carry. The carry flag moves into bit 0. Bit 7 moves to the carry flag.

Flags affected: C, P, S, Z
Flags reset: H, N

RL (IX+dd) <no 8080>
RL (IY+dd) <no 8080>

The memory byte referenced by the sum of the index register and the displacement is rotated left through carry. The carry flag moves into bit 0. Bit 7 moves to the carry flag.

Flags affected: C, P, S, Z
Flags reset: H, N

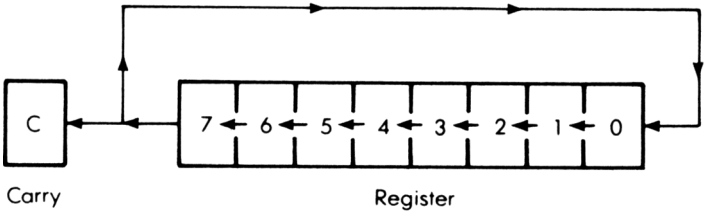
RL r <no 8080>

The byte in register r is rotated left through carry. The carry flag moves into bit 0. Bit 7 moves to the carry flag. Note: the instruction RL A performs the same task that instruction RLA does, but instruction RLA is twice as fast.

Flags affected: C, P, S, Z
Flags reset: H, N

RLA <RAL>

The following RLC and RLCA instructions rotate bits to the left.



RLC (HL) <no 8080>

The byte referenced by the HL pair is rotated left circularly. Bit 7 moves to both the zero bit and the carry flag.

Flags affected: C, P, S, Z
Flags reset: H, N

RLC (IX+dd) <no 8080>
RLC (IY+dd) <no 8080>

The byte referenced by the specified index register plus the displacement is rotated left circularly. Bit 7 moves to both the zero bit and the carry flag.

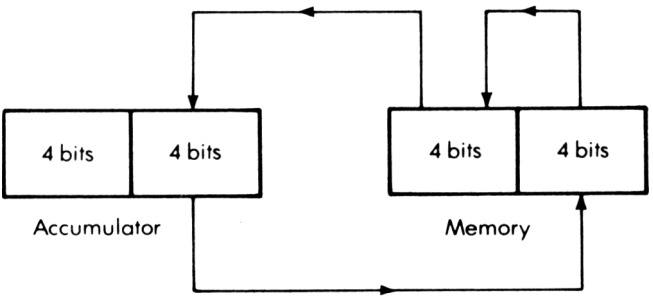
Flags affected: C, P, S, Z
Flags reset: H, N

RLC r <no 8080>

The byte in register r is rotated left circularly. Bit 7 moves to both the zero bit and the carry flag. Note: RLC A performs the same task that instruction RLCA does, but instruction RLCA is twice as fast.

Flags affected: C, P, S, Z
Flags reset: H, N

RLCA <RLC>



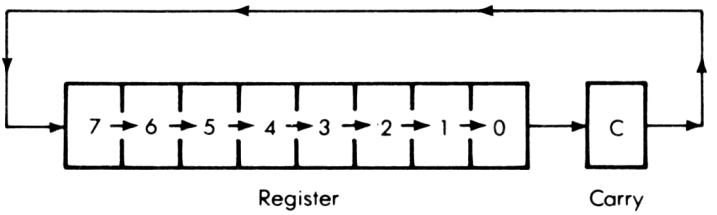
RLD <no 8080>

A four-bit rotation over 12 bits. The low four bits of A move to the low

four bits of the memory location referenced by the HL pair. The original low four bits of memory move to the high four bits. The original high four bits move to the low four bits of A. This instruction is used for BCD operations.

Flags affected: P, S, Z
Flags reset: H, N

The following RR and RRA instructions rotate bits to the right through carry.



RR (HL) <no 8080>

The memory byte pointed to by the HL pair is rotated right through carry. Carry moves to bit 7. Bit 0 moves to the carry flag.

Flags affected: C, P, S, Z
Flags reset: H, N

RR (IX+dd) <no 8080>
RR (IY+dd) <no 8080>

The memory byte pointed to by the specified index register plus the offset is rotated right through carry. The carry flag moves to bit 7. Bit 0 moves to the carry flag.

Flags affected: C, P, S, Z
Flags reset: H, N

RR r <no 8080>

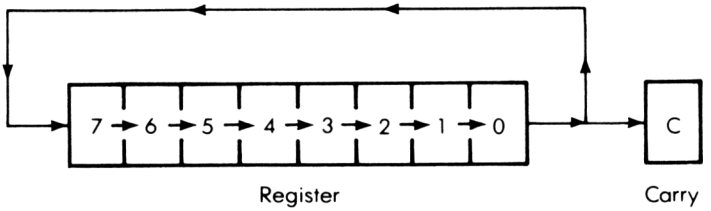
The byte in register r is rotated right through carry. Carry moves to bit 7. Bit 0 moves to the carry flag. Note: RR A performs the same task that

instruction RRA does, but instruction RRA is twice as fast.

Flags affected: C, P, S, Z
Flags reset: H, N

RRA <RAR>

The following RRC and RRCA instructions rotate bits to the right.



RRC (HL) <no 8080>

The memory byte pointed to by the HL pair is rotated right circularly. Bit 0 moves to both the carry flag and bit 7.

Flags affected: C, P, S, Z
Flags reset: H, N

RRC (IX+dd) <no 8080>
RRC (IY+dd) <no 8080>

The memory byte pointed to by the index register plus the offset is rotated right circularly. Bit 0 moves to both the carry flag and bit 7.

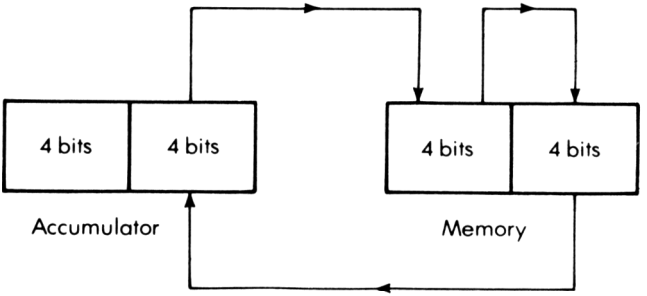
Flags affected: C, P, S, Z
Flags reset: H, N

RRC r <no 8080>

The byte in register r is rotated right circularly. Bit 0 moves to both the carry flag and bit 7. Note: RRC A performs the same task that instruction RRCA does, but instruction RRCA is twice as fast.

Flags affected: C, P, S, Z
Flags reset: H, N

RRCA <RRC>



RRD <no 8080>

A four-bit rotation over 12 bits. The low four bits of A move to the high four bits of the memory location referenced by the HL pair. The original high four bits of memory move to the low four bits. The original low four bits move to the low four bits of A. This instruction is used for BCD operations.

Flags affected: P, S, Z
Flags reset: H, N

RST 00H <RST 0>
RST 08H <RST 1>
RST 10H <RST 2>
RST 18H <RST 3>
RST 20H <RST 4>
RST 28H <RST 5>
RST 30H <RST 6>
RST 38H <RST 7>

SBC A,(HL) <SBB M>

SBC A,(IX+dd) <no 8080>
SBC A,(IY+dd) <no 8080>

Subtract the carry flag and the memory byte pointed to by the sum of the index register and the displacement from the accumulator. The result is placed in the accumulator.

Flags affected: C, H, O, S, Z
Flag set: N

```
SBC   A,r        <SBB   r>
```

```
SBC   A,nn       <SBI   nn>
```

```
SBC   HL,BC      <no 8080>
SBC   HL,DE      <no 8080>
SBC   HL,HL      <no 8080>
SBC   HL,SP      <no 8080>
```

Subtract the specified CPU double register and the carry flag from the HL register pair. The result is placed in HL. You may need to reset the carry flag with an OR A operation before using these instructions.

Flags affected: C, H, O, S, Z
Flag set: N

```
SCF      <STC>
```

```
SET   b,(HL)          <no 8080>
SET   b,(IX+dd)       <no 8080>
SET   b,(IY+dd)       <no 8080>
```

Set bit b of the memory byte referenced by the second operand. Bit b can range from 0 through 7.

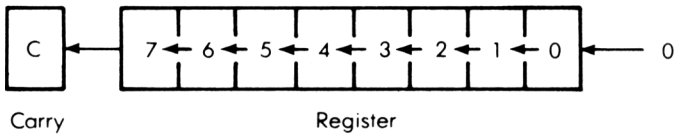
Flags affected: none
Flag reset: N

```
SET   b,r        <no 8080>
```

Set bit b of register r. Bit b can range from 0 through 7.

Flags affected: none
Flag reset: N

The following SLA instructions shift bits to the left.



SLA (HL) <no 8080>

Perform an arithmetic shift left on the memory byte pointed to by the HL pair. Bit 7 is moved to the carry flag. A zero is moved into bit 0. This operation doubles the original value.

Flags affected: C, P, S, Z
Flags reset: H, N

SLA (IX+dd) <no 8080>
SLA (IY+dd) <no 8080>

Perform an arithmetic shift left on the memory byte pointed to by the index register plus the displacement. Bit 7 is moved to the carry flag. A zero is moved into bit 0. This operation doubles the original value.

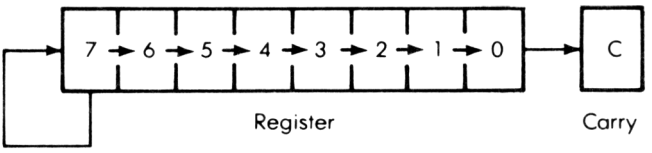
Flags affected: C, P, S, Z
Flags reset: H, N

SLA r <no 8080>

Perform an arithmetic shift left on register r. Bit 7 is moved to the carry flag. A zero is moved into bit 0. This operation doubles the original value. Note: SLA A performs the same task that instruction ADD A,A does, but instruction ADD A,A is twice as fast.

Flags affected: C, P, S, Z
Flags reset: H, N

The following SRA instructions shift bits to the right.



SRA (HL) <no 8080>

Perform an arithmetic shift right on the memory byte pointed to by the HL pair. Bit 0 moves to the carry flag. Bit 7 is copied into bit 6.

Flags affected: C, P, S, Z
Flags reset: H, N

SRA (IX+dd) <no 8080>
SRA (IY+dd) <no 8080>

Perform an arithmetic shift right on the byte pointed to by the index register plus the displacement. Bit 0 moves to carry and bit 7 is copied into bit 6.

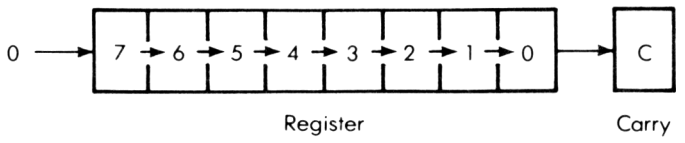
Flags affected: C, P, S, Z
Flags reset: H, N

SRA r <no 8080>

Perform an arithmetic shift right on register r. Bit 0 moves to carry and bit 7 is copied into bit 6. The operation effectively halves the register value. The carry flag represents the remainder. The carry flag is set if the original number was odd.

Flags affected: C, P, S, Z
Flags reset: H, N

The following SRL instructions shift bits to the right.



SRL (HL) <no 8080>

Perform a logical shift right on the byte pointed to by the HL register pair. A zero bit is moved into bit 7. Bit 0 moves to the carry flag.

Flags affected: C, P, Z
Flags reset: H, N, S

SRL (IX+dd) <no 8080>
SRL (IY+dd) <no 8080>

Perform a logical shift right on the byte pointed to by the index register plus the displacement. A zero bit is moved into bit 7. Bit 0 moves to the carry flag.

Flags affected: C, P, Z
Flags reset: H, N, S

SRL r <no 8080>

Perform a logical shift right on register r. A zero bit is moved into bit 7. Bit 0 moves to the carry flag.

Flags affected: C, P, S, Z
Flags reset: H, N

SUB (HL) <SUB M>

SUB (IX+dd) <no 8080>
SUB (IY+dd) <no 8080>

Subtract the memory byte referenced by the index register plus the displacement from the value in the accumulator. The result is placed in A.

Flags affected: C, H, O, S, Z
Flag set: N

SUB r <SUB r>

SUB nn <SUI nn>

XOR (HL) <XRA M>

XOR (IX+dd) <no 8080>
XOR (IY+dd) <no 8080>

Perform a logical exclusive OR with the accumulator and the byte referenced by the sum of specified index register and the displacement. The result is placed in the accumulator.

Flags affected: P, S, Z
Flags reset: C, H, N

XOR r <XRA r>

XOR nn <XRI nn>

APPENDIX I

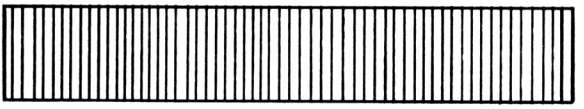
The CP/M BDOS Functions

The Nondisk BDOS Functions

Function number (in C)	Operation	Value sent	Value returned
1	Read console		character in A
2	Write console	character in E	
3	Read reader		character in A
4	Write punch	character in E	
5	Write list	character in E	
6	Direct console I/O	FF (input) character (output)	0 = not ready or character in A
7	Determine IOBYTE		byte in A
8	Set IOBYTE	in E	
9	Print buffer	address in DE	
10	Read buffer	address in DE	
11	Return console status		byte in A
12	Return CP/M version		byte in A and L

The Disk-Related BDOS Functions

Function number (in C)	Operation	Value sent	Value returned
13	Reset disks		
14	Select disk	E = disk	
15	Open file	DE = FCB	A = error code
16	Close file	DE = FCB	A = error code
17	Search for first	DE = FCB	A = error code
18	Search for next		A = error code
19	Delete file	DE = FCB	A = error code
20	Read sequential	DE = FCB	A = error code
21	Write sequential	DE = FCB	A = error code
22	Make new file	DE = FCB	A = error code
23	Rename file	DE = original FCB	A = error code
24	Determine logged-in drives		HL = vector
25	Find default drive		A = drive
26	Set DMA address	DE = address	
27	Get allocation vector		HL = vector
28	Write protect disk		
29	Find R/O drives		HL = vector
30	Set file attributes	DE = FCB	
31	Get disk parameter block		HL = block
32	Get or set user number	E = FF E = new user number	A = user number
33	Read randomly	DE = FCB	A = error code
34	Write randomly	DE = FCB	A = error code
35	Get file size	DE = FCB	
36	Set random record	DE = FCB	



INDEX

Selections from The SYBEX Library

Personal Computers

YOUR KAYPRO II/4/10™

by Andrea Reid and Gary Deidrichs

250 pp., illustr., Ref. 0-166

This book is a non-technical introduction to the KAYPRO family of computers. You will find all you need to know about operating your KAYPRO within this one complete guide.

Languages

C

UNDERSTANDING C

by Bruce Hunter

200 pp., Ref 0-123

Explains how to use the powerful C language for a variety of applications. Some programming experience assumed.

BASIC

YOUR FIRST BASIC PROGRAM

by Rodnay Zaks

150pp. illustr. in color, Ref. 0-129

A "how-to-program" book for the first time computer user, aged 8 to 88.

FIFTY BASIC EXERCISES

by J. P. Lamoitier

232 pp., 90 illustr., Ref. 0-056

Teaches BASIC by actual practice, using graduated exercises drawn from everyday applications. All programs written in Microsoft BASIC.

BASIC FOR BUSINESS

by Douglas Hergert

224 pp., 15 illustr., Ref. 0-080

A logically organized, no-nonsense introduction to BASIC programming for business applications. Includes many fully-explained accounting programs, and shows you how to write them.

EXECUTIVE PLANNING WITH BASIC

by X. T. Bui

196 pp., 19 illustr., Ref. 0-083

An important collection of business management decision models in BASIC, including Inventory Management (EOQ), Critical Path Analysis and PERT, Financial Ratio Analysis, Portfolio Management, and much more.

BASIC PROGRAMS FOR SCIENTISTS AND ENGINEERS

by Alan R. Miller

318 pp., 120 illustr., Ref. 0-073

This book from the "Programs for Scientists and Engineers" series provides a library of problem-solving programs while developing proficiency in BASIC.

CELESTIAL BASIC

by Eric Burgess

300 pp., 65 illustr., Ref. 0-087

A collection of BASIC programs that rapidly complete the chores of typical astronomical computations. It's like having a planetarium in your own home! Displays apparent movement of stars, planets and meteor showers.

YOUR SECOND BASIC PROGRAM

by Gary Lippman

250 pp., illustr., Ref. 0-152

A sequel to *Your First BASIC Program*, this book follows the same patient, detailed approach and brings you to the next level of programming skill.

Pascal

INTRODUCTION TO PASCAL (Including UCSD Pascal™)

by Rodnay Zaks

420 pp., 130 illustr., Ref. 0-066

A step-by-step introduction for anyone wanting to learn the Pascal language. Describes UCSD and Standard Pascals. No technical background is assumed.

THE PASCAL HANDBOOK

by Jacques Tiberghien

486 pp., 270 illustr., Ref. 0-053

A dictionary of the Pascal language, defining every reserved word, operator, procedure and function found in all major versions of Pascal.

INTRODUCTION TO THE UCSD p-SYSTEM™

by Charles W. Grant and Jon Butah

300 pp., 10 illustr., Ref. 0-061

A simple, clear introduction to the UCSD Pascal Operating System; for beginners through experienced programmers.

Software and Applications

Operating Systems

THE CP/M® HANDBOOK

by Rodnay Zaks

320 pp., 100 illustr., Ref 0-048

An indispensable reference and guide to CP/M—the most widely-used operating system for small computers.

THE BEST OF CP/M® SOFTWARE

by John D. Halamka

250 pp., illustr., Ref. 0-100

This book reviews tried-and-tested, commercially available software for your CP/M system.

REAL WORLD UNIX™

by John D. Halamka

250 pp., illustr., Ref. 0-093

This book is written for the beginning and intermediate UNIX user in a practical, straightforward manner, with specific instructions given for many special applications.

THE CP/M PLUS™ HANDBOOK

by Alan R. Miller

250 pp., illustr., Ref. 0-158

This guide is easy for the beginner to understand, yet contains valuable information for advanced users of CP/M Plus (Version 3).

Business Software

INTRODUCTION TO WORDSTAR™

by Arthur Naiman

202 pp., 30 illustr., Ref. 0-077

Makes it easy to learn how to use WordStar, a powerful word processing program for personal computers.

PRACTICAL WORDSTAR™ USES

by Julie Anne Arca

200 pp., illustr., Ref. 0-107

Pick your most time-consuming office tasks and this book will show you how to streamline them with WordStar.

DOING BUSINESS WITH SUPERCALC™

by Stanley R. Trost

248 pp., illustr., Ref. 0-095

Presents accounting and management planning applications—from financial statements to master budgets; from pricing models to investment strategies.

UNDERSTANDING dBASE II™

by Alan Simpson

220 pp., illustr., Ref. 0-147

Learn programming techniques for mailing label systems, bookkeeping and data base management, as well as ways to interface dBASE II with other software systems.

DOING BUSINESS WITH dBASE II™

by Stanley R. Trost

250 pp., illustr., Ref. 0-160

Learn to use dBASE II for accounts receivable, recording business income and expenses, keeping personal records and mailing lists, and much more.

DOING BUSINESS WITH PFS®

by Stanley R. Trost

250 pp., illustr., Ref. 0-161

This practical guide describes specific business and personal applications in detail. Learn to use PFS for accounting, data analysis, mailing lists and more.

INFOPOWER: PRACTICAL INFOSTAR™ USES

by Jule Anne Arca and Charles F. Pirro

275 pp., illustr., Ref. 0-108

This book gives you an overview of InfoStar, including DataStar and ReportStar, WordStar, MailMerge, and SuperSort. Hands on exercises take you step-by-step through real life business applications.

COMPUTER POWER FOR YOUR LAW OFFICE

by Daniel Remer

225 pp., Ref. 0-109

How to use computers to reach peak productivity in your law office, simply and inexpensively.

COMPUTER POWER FOR YOUR ACCOUNTING OFFICE

by James Morgan

250 pp., illustr., Ref. 0-164

This book is a convenient source of information about computerizing you accounting office, with an emphasis on hardware and software options.

Assembly Language Programming

PROGRAMMING THE 6502

by Rodnay Zaks

386 pp., 160 illustr., Ref. 0-046

Assembly language programming for the 6502, from basic concepts to advanced data structures.

PROGRAMMING THE Z80

by Rodnay Zaks

624 pp., 200 illustr., Ref. 0-069

A complete course in programming the Z80 microprocessor and a thorough introduction to assembly language.

PROGRAMMING THE 8086/8088

by James W. Coffron

300 pp., illustr., Ref. 0-120

This book explains how to program the 8086 and 8088 in assembly language. No prior programming knowledge required.

Hardware and Peripherals

MICROPROCESSOR INTERFACING TECHNIQUES

by Rodnay Zaks and Austin Lesea

456 pp., 400 illustr., Ref. 0-029

Complete hardware and software interconnect techniques, including D to A conversion, peripherals, standard buses and troubleshooting.

THE RS-232 SOLUTION

by Joe Campbell

225 pp., illustr., Ref. 0-140

Finally, a book that will show you how to correctly interface your computer to any RS-232-C peripheral.

are different.

Here is why . . .

At SYBEX, each book is designed with you in mind. Every manuscript is carefully selected and supervised by our editors, who are themselves computer experts. We publish the best authors, whose technical expertise is matched by an ability to write clearly and to communicate effectively. Programs are thoroughly tested for accuracy by our technical staff. Our computerized production department goes to great lengths to make sure that each book is well-designed.

In the pursuit of timeliness, SYBEX has achieved many publishing firsts. SYBEX was among the first to integrate personal computers used by authors and staff into the publishing process. SYBEX was the first to publish books on the CP/M operating system, microprocessor interfacing techniques, word processing, and many more topics.

Expertise in computers and dedication to the highest quality product have made SYBEX a world leader in computer book publishing. Translated into fourteen languages, SYBEX books have helped millions of people around the world to get the most from their computers. We hope we have helped you, too.

For a complete catalog of our publications please contact:

U.S.A.
SYBEX, Inc.
2344 Sixth Street
Berkeley,
California 94710
Tel: (800) 227-2346
(415) 848-8233
Telex: 336311

FRANCE
SYBEX
4 Place Félix-Eboué
75583 Paris Cedex 12
France
Tel: 1/347-30-20
Telex: 211801

GERMANY
SYBEX-VERLAG
Heyestr. 22
4000 Düsseldorf 12
West Germany
Tel: (0211) 287066
Telex: 08 588 163

MASTERING CP/M

This book explains techniques for using, altering, and adding features to the CP/M operating system. It will give you a complete understanding of the CP/M modules, particularly the BIOS and BDOS. Macros, the powerful tools for developing and organizing assembly language programs, are clearly presented.

For advanced CP/M users and systems programmers, this book will enable you to explore the subtleties of CP/M, and to enhance the power of CP/M commands.

Included is a comprehensive set of reference appendices.

ABOUT THE AUTHOR:

Alan R. Miller is a professor at the New Mexico Institute of Mining and Technology and a contributing software editor to Interface Age Magazine.

He received his Ph.D in Engineering from the University of California at Berkeley, and has been teaching programming methods to engineers since 1967. He has worked with CP/M since its inception and has developed a number of techniques that have improved its usefulness. He is the author of three other SYBEX books.

". . . good, comprehensive programmer's guide to CP/M."
—Microsystems

". . . crammed full of useful chunks of software which can be incorporated into other programs." *—Computing Now*

". . . a vast array of information . . . No book I've seen to date goes into the depth of technical information nor offers the technical expertise this one does . . . If you enjoy crawling around on the bits and bytes level in your computer, then you'll love this book." *—Computing Canada*

Miller

MASTERING CP/M

0-068