marcus b

0.
THE PROGRAMMER'S
CP/M® HANDBOOK
Several years ago I was told that “Perfection is an English education, an American salary, and a Japanese wife.”

Accordingly, I wish to thank the members of Staff at Culford School in England, who gave me the English education, the people who work with me at Johnson-Laird Inc. and Control-C Software and our clients, who give me my American salary, and Mr. and Mrs. Kitagawa, who gave me Kay Kitagawa (who not only married me but took over where my English grammar left off).

A.J-L.
Acknowledgments

Although this book is not authorized or endorsed by Digital Research, I would like to express my thanks to Gary Kildall and Kathy Strutynski of Digital Research, and to Phil Nelson (formerl of Digital Research, now of Victor Technology) for their help in keeping me on the path to truth in this book. I would also like to thank Denise Penrose, Marty McNiff, Mary Borchers, and Ralph Baumgartner at Osborne/McGraw-Hill for their apparently inexhaustible patience.

A.J-L.
Contents

1 Introduction 1
2 The Structure of CP/M 5
3 The CP/M File System 17
4 The Console Command Processor (CCP) 45
5 The BASIC Disk Operating System 67
6 The BASIC Input/Output System 147
7 Building a New CP/M System 183
8 Writing an Enhanced BIOS 209
9 Dealing with Hardware Errors 295
10 Debugging a New CP/M System 319
11 Additional Utility Programs 371
12 Error Messages 449
A ASCII Character Set 465
B CP/M Command Summary 469
C Summary of BDOS Calls 479
D Summary of BIOS Calls 485
Index 487
This book is a sequel to the *Osborne CP/M® User Guide* by Thom Hogan. It is a technical book written mainly for programmers who require a thorough knowledge of the internal structure of CP/M—how the various pieces of CP/M work, how to use CP/M as an operating system, and finally, how to implement CP/M on different computer systems. This book is written for people who

- Have been working with microcomputers that run Digital Research's CP/M operating system.
- Understand the internals of the microprocessor world—bits, bytes, ports, RAM, ROM, and other jargon of the programmer.
- Know how to write in assembly language for the Intel 8080 or Zilog Z80 Central Processing Unit (CPU) chips.

If you don't have this kind of background, start by getting practical experience on a system running CP/M and by reading the following books from Osborne/McGraw-Hill:

- *An Introduction to Microcomputers: Volume I—Basic Concepts*

This book describes the fundamental concepts and facts that you need to
know about microprocessors in order to program them. If you really need basics, there is a Volume 0 called *The Beginner's Book.*

- *8080A/8085 Assembly Language Programming*
  This book covers all aspects of writing programs in 8080 assembly language, giving many examples.

- *Osborne CP/M® User Guide (2nd Edition)*
  This book introduces the CP/M operating system. It tells you how to use CP/M as a tool to get things done on a computer.

The book you are reading now deals only with CP/M Version 2.2 for the 8080 or Z80 chips. At the time of writing, new versions of CP/M and MP/M (the multi-user, multi-tasking successor to CP/M) were becoming available. CP/M-86 and MP/M-86 for the Intel 8086 CPU chip and MP/M-II for the 8080 or Z80 chips had been released, with CP/M 3.0 (8080 or Z80) in the wings. The 8086, although related architecturally to the 8080, is different enough to make it impossible to cover in detail in this book; and while MP/M-II and MP/M-86 are similar to CP/M, they have many aspects that cannot be adequately discussed within the scope of this book.

### Outline of Contents

This book explains topics as if you were starting from the top of a pyramid. Successive “slices” down the pyramid cover the same material but give more detail.

The first chapter includes a brief outline of the notation used in this book for example programs written in Intel 8080 assembly language and in the C programming language.

Chapter 2 deals with the structure of CP/M, describing its major parts, their positions in memory, and their functions.

Chapter 3 discusses CP/M's file system in as much detail as possible, given its proprietary nature. The directory entry, disk parameter block, and file organization are described.

Chapter 4 covers the Console Command Processor (CCP), examining the way in which you enter command lines, the CP/M commands built into the CCP, how the CCP loads programs, and how it transfers control to these programs.

Chapter 5 begins the programming section. It deals with the system calls your programs can make to the high-level part of CP/M, the Basic Disk Operating System (BDOS).

Chapters 6 through 10 deal with the Basic Input/Output System (BIOS). This is the part of CP/M that is unique to each computer system. It is the part that you as a programmer will write and implement for your own computer system.

Chapter 6 describes a standard implementation of the BIOS.
Chapter 7 describes the mechanism for rebuilding CP/M for a different configuration.

Chapter 8 tells you how to write an enhanced BIOS.

Chapter 9 takes a close look at how to handle hardware errors—how to detect and deal with them, and how to make this task easier for the person using the computer.

Chapter 10 discusses the problems you may face when you try to debug your BIOS code. It includes debugging subroutines and describes techniques that will save you time and suffering.

Chapter 11 describes several utility programs, some that work with the features of the enhanced BIOS in Chapter 8 and some that will work with all CP/M 2 implementations.

Chapter 12 concerns error messages and some oddities that you will discover, sometimes painfully, in CP/M. Messages are explained and some probable causes for strange results are documented.

The appendixes contain "ready-reference" information and summaries of information that you need at your side when designing, coding, and testing programs to run under CP/M or your own BIOS routines.

---

**Notation**

When you program your computer, you will be sitting in front of your terminal interacting with CP/M and the utility programs that run under it. The sections that follow describe the notation used to represent the dialog that will appear on your terminal and the output that will appear on your printer.

**Console Dialog**

This book follows the conventions used in the *Osborne CP/M User Guide*, extended slightly to handle more complex dialogs. In this book

- `<name>` means the ASCII character named between the angle brackets, `< and >`. For example, `<BEL>` is the ASCII Bell character, and `<HT>` is the ASCII Horizontal Tab Character. (Refer to Appendix A for the complete ASCII character set.)
- `<cr>` means to press the CARRIAGE RETURN key.
- 123 or a number without a suffix means a decimal number.
- 100B or a number followed by B means a binary number.
- 0A5H or a number followed by H means a hexadecimal number. A hexadecimal number starting with a letter is usually shown with a leading 0 to avoid confusion.
4 The CP/M Programmer's Handbook

- \(^x\) means to hold the CONTROL (CTRL) key down while pressing the x key.
- Underline is keyboard input you type. Output from the computer is shown without underlining.

**Assembly Language Program Examples**

This book uses Intel 8080 mnemonics throughout as a "lowest common denominator"—the Z80 CPU contains features absent in the 8080, but not vice versa. Output from Digital Research's ASM Assembler is shown so that you can see the generated object code as well as the source.

**High-Level Language Examples**

The utility programs described in Chapter 11 are written in C, a programming language which lends itself to describing algorithms clearly without becoming entangled in linguistic bureaucracy. Cryptic expressions have been avoided in favor of those that most clearly show how to solve the problem. Ample comments explain the code.


**Example Programs on Diskette**

Example programs in this book have been assembled with ASM and tested with DDT, Digital Research's Dynamic Debugging Tool. C examples were compiled using Leor Zolman's BDS C Compiler (Version 1.50) and tested using the enhanced BIOS described in Chapter 8.

All of the source code shown in this book is available on a single-sided, single-density, 8-inch diskette (IBM 3740 format). Please do not contact Osborne/McGraw-Hill to order this diskette. Call or write

Johnson-Laird, Inc.
Attn: The CP/M Programmer's Handbook Diskette
6441 SW Canyon Court
Portland, OR 97221
Tel: (503) 292-6330

The diskette is available for $50 plus shipping costs.
This chapter introduces the pieces that make up CP/M — what they are and what they do. This bird’s-eye view of CP/M will establish a framework to which later chapters will add more detailed information.

You may have purchased the standard version of CP/M directly from Digital Research, but it is more likely you received CP/M when you bought your microprocessor system or its disk drive system. Or, you may have purchased CP/M separately from a software distributor. In any case, this distributor or the company that made the system or disk drive will have already modified the standard version of CP/M to work on your specific hardware. Most manufacturers’ versions of CP/M have more files on their system diskette than are described here for the standard Digital Research release.

Some manufacturers have rewritten all the documentation so that you may not have received any Digital Research CP/M manuals. If this is the case, you should order the complete set from Digital Research, because as a programmer, you will need to have them for reference.
Digital Research provides a standard "vanilla-flavored" version of CP/M that will run only on the Intel Microcomputer Development System (MDS). The CP/M package from Digital Research contains seven manuals and an 8-inch, single-sided, single-density standard IBM 3740 format diskette.

The following manuals come with this CP/M system:

- *An Introduction to CP/M Features and Facilities.* This is a brief description of CP/M and the utility programs you will find on the diskette. It describes only CP/M version 1.4.

- *CP/M 2.0 User's Guide.* Digital Research wrote this manual to describe the new features of CP/M 2.0 and the extensions made to existing CP/M 1.4 features.

- *ED: A Context Editor for the CP/M Disk System.* By today's standards, ED is a primitive line editor, but you can still use it to make changes to files containing ASCII text, such as the BIOS source code.

- *CP/M Assembler (ASM).* ASM is a simple but fast assembler that can be used to translate the BIOS source code on the diskette into machine code. Since ASM is only a bare-bones assembler, many programmers now use its successor, MAC (also from Digital Research).

- *CP/M Dynamic Debugging Tool (DDT).* DDT is an extremely useful program that allows you to load programs in machine code form and then test them, executing the program either one machine instruction at a time or stopping only when the CPU reaches a specific point in the program.

- *CP/M Alteration Guide.* There are two manuals with this title, one for CP/M version 1.4 and the other for 2.0. Both manuals describe, somewhat cryptically, how to modify CP/M.

- *CP/M Interface Guide.* Again, there are two versions, 1.4 and 2.0. These manuals tell you how to write programs that communicate directly with CP/M.

The diskette supplied by Digital Research has the following files:

*ASM.COM*

The CP/M assembler.

*BIOS.ASM*

A source code file containing a sample BIOS for the Intel Microcomputer Development System (MDS). Unless you have the MDS, this file is useful only as an example of a BIOS.
Chapter 2: The Structure of CP/M

CBIOS.ASM
Another source code file for a BIOS. This one is skeletal: There are gaps so that you can insert code for your computer.

DDT.COM
The Dynamic Debugging Tool program.

DEBLOCK.ASM
A source code file that you will need to use in the BIOS if your computer uses sector sizes other than 128 bytes. It is an example of how to block and deblock 128-byte sectors to and from the sector size you need.

DISKDEF.LIB
A library of source text that you will use if you have a copy of Digital Research's advanced assembler, MAC.

DUMP.ASM
The source for an example program. DUMP reads a CP/M disk file and displays it in hexadecimal form on the console.

DUMP.COM
The actual executable program derived from DUMP.ASM.

ED.COM
The source file editor.

LOAD.COM
A program that takes the machine code file output by the assembler, ASM, and creates another file with the data rearranged so that you can execute the program by just typing its name on the keyboard.

MOVCPM.COM
A program that creates versions of CP/M for different memory sizes.

PIP.COM
A program for copying information from one place to another (PIP is short for Peripheral Interchange Program).

STAT.COM
A program that displays statistics about the CP/M and other information that you have stored on disks.

SUBMIT.COM
A program that you use to enter CP/M commands automatically. It helps you avoid repeated typing of long command sequences.

SYSGEN.COM
A program that writes CP/M onto diskettes.

XSUB.COM
An extended version of the SUBMIT program. The files named previously
The pieces of CP/M

CP/M is composed of the Basic Disk Operating System (BDOS), the Console Command Processor (CCP), and the Basic Input/Output System (BIOS).

On occasion you will see references in CP/M manuals to something called the FDOS, which stands for “Floppy Disk Operating System.” This name is given to the portion of CP/M consisting of both the BDOS and BIOS and is a relic passed down from the original version. Since it is rarely necessary to refer to the BDOS and the BIOS combined as a single entity, no further references to the FDOS will be made in this book.

The BDOS and the CCP are the proprietary parts of CP/M. Unless you are willing to pay several thousand dollars, you cannot get the source code for them. You do not need to. CP/M is designed so that all of the code that varies from one machine to another is contained in the BIOS, and you do get the BIOS source code from Digital Research. Several companies make specialized BIOSs for different computer systems. In many cases they, as well as some CP/M hardware manufacturers, do not make the source code for their BIOS available; they have put time and effort into building their BIOS, and they wish to preserve the proprietary nature of what they have done.

You may have to build a special configuration of CP/M for a specific computer. This involves no more than the following four steps:

1. Make a version of the BDOS and CCP for the memory size of your computer.
2. Write a modified version of the BIOS that matches the hardware in your computer.
3. Write a small program to load CP/M into memory when you press the RESET button on your computer.
4. Join all of the pieces together and write them out to a diskette.

These steps will be explained in Chapters 7, 8, and 9.

In the third step, you write a small program that loads CP/M into memory when you press the RESET button on your computer. This program is normally called the bootstrap loader. You may also see it called the “boot” or even the “cold start” loader. “Bootstrap” refers to the idea that when the computer is first turned on, there is no program to execute. The task of getting that very first program into the computer is, conceptually, as difficult as attempting to pick yourself up off the ground by pulling on your own bootstraps. In the early days of computing, this operation was performed by entering instructions manually—setting large banks
of switches (the computer was built to read the switches as soon as it was turned on). Today, microcomputers contain some small fragment of a program in "non-volatile" read-only memory (ROM)—memory that retains data when the computer is turned off. This stored program, usually a Programmable Read Only Memory (PROM) chip, can load your bootstrap program, which in turn loads CP/M.

**CP/M Diskette Format**

The standard version of CP/M is formatted on an 8-inch, single-sided diskette. Diskettes other than this type will probably have different layouts; hard disks definitely will be different.

The physical format of the standard 8-inch diskette is shown in Figure 2-1. The

![Floppy disk layout](image)
Figure 2-2. Layout of CP/M on tracks 0 and 1 of floppy disk

The diskette has a total of 77 concentric tracks numbered from zero (the outermost) to 76 (the innermost). Each of these tracks is divided radially into 26 sectors. These physical sectors are numbered from 1 to 26; physical sector zero does not exist. Each sector has enough space for 128 bytes of data.

Even when CP/M is implemented on a large hard disk with much larger sector sizes, it still works with 128-byte sectors. The BIOS has extra instructions that convert the real sectors into CP/M-style 128-byte sectors.

A final note on physical format: The soft-sectored, single-sided, single-density, 8-inch diskette (IBM 3740 format) is the only standard format. Any other formats will be unique to the hardware manufacturer that uses them. It is unlikely that you can read a diskette on one manufacturer's computer if it was written on another's, even though the formats appear to be the same. For example, a single-sided, double-density diskette written on an Intel Development System cannot be read on a Digital Microsystems computer even though both use double-density format. If you want to move data from one computer to another, use 8-inch, single-sided, single-density format diskettes, and it should work.
In order to see how CP/M is stored on a diskette, consider the first two tracks on the diskette, track 0 and track 1. Figure 2-2 shows how the data is stored on these tracks.

**Loading CP/M**

The events that occur after you first switch on your computer and put the CP/M diskette into a disk drive are the same as those that occur when you press the **RESET** button—the computer generates a **RESET** signal.

The **RESET** button stops the central processor unit (CPU). All of the internals of the CPU are set to an initial state, and all the registers are cleared to zero. The program counter is also cleared to zero so that when the **RESET** signal goes away (it only lasts for a few milliseconds), the CPU starts executing instructions at location 0000H in memory.

Memory chips, when they first receive power, cannot be relied upon to contain any particular value. Therefore, hardware designers arrange for some initial instructions to be forced into memory at location 0000H and onward. It is this feat that is like pulling yourself up by your own bootstraps. How can you make the computer obey a particular instruction when there is "nothing" (of any sensible value) inside the machine?

There are two common techniques for placing preliminary instructions into memory:

* **Force-feeding**

  With this approach, the hardware engineer assumes that when the **RESET** signal is applied, some part of the computer system, typically the floppy disk controller, can masquerade as memory. Just before the CPU is unleashed, the floppy disk controller will take control of the computer system and copy a small program into memory at location 0000H and upward. Then the CPU is allowed to start executing instructions at location 0000H. The disk controller preserves the instructions even when power is off because they are stored in nonvolatile PROM-based firmware. These instructions make the disk controller read the first sector of the first track of the system diskette into memory and then transfer control to it.

* **Shadow ROM**

  This is a variation of the force-feeding technique. The hardware manufacturer arranges some ROM at location 0000H. There is also some normal read/write memory at location 0000H, but this is electronically disabled when the **RESET** signal has been activated. The CPU, unleashed at location 0000H, starts to execute the ROM instruction. The first act of the ROM program is to copy itself into read/write memory at some convenient location higher up in memory and transfer control of the machine up to this copy. Then the real memory at location 0000H can be turned on, the ROM turned off, and the first sector on the disk read in.
With either technique, the result is the same. The first sector of the disk is read into memory and control is transferred to the first instruction contained in the sector.

This first sector contains the main CP/M bootstrap program. This program initializes some aspects of the hardware and then reads in the remainder of track 0 and most of the sectors on track 1 (the exact number depends on the overall length of the BIOS itself). The CP/M bootstrap program will contain only the most primitive diskette error handling, trying to read the disk over and over again if the hardware indicates that it is having problems reading a sector.

The bootstrap program loads CP/M to the correct place in memory; the load address is a constant in the bootstrap. If you need to build a version of CP/M that uses more memory, you will need to change this load address inside the bootstrap as well as the address to which the bootstrap will jump when all of CP/M has been read in. This address too is a constant in the bootstrap program.

The bootstrap program transfers control to the first instruction in the BIOS, the cold boot entry point. “Cold” implies that the operation is starting cold from an empty computer.

The cold boot code in the BIOS will set up the hardware in your computer. That is, it programs the various chips that control the speed at which serial ports transmit and receive data. It initializes the serial port chips themselves and generally readies the computer system. Its final act is to transfer control to the first instruction in the BDOS in order to start up CP/M proper.

Once the BDOS receives control, it initializes itself, scans the file directory on the system diskette, and hands over control to the CCP. The CCP then outputs the “A>” prompt to the console and waits for you to enter a command. CP/M is then ready to do your bidding.

At this point, it is worthwhile to review which CP/M parts are in memory, where in memory they are, and what functions they perform.

This overview will look at memory first. Figure 2-3 shows the positions in memory of the Console Command Processor, the Basic Disk Operating System, and the Basic Input/Output System.

By touching upon these major memory components—the CCP, BDOS, and BIOS—this discussion will consider which modules interact with them, how requests for action are passed to them, and what functions they can perform.

**Console Command Processor**

As you can see in Figure 2-3, the CCP is the first part of CP/M that is encountered going “up” through memory addresses. This is significant when you consider that the CCP is only necessary in between programs. When CP/M is idle, it needs the CCP to interact with you, to accept your next command. Once CP/M has started to execute the command, the CCP is redundant; any console interaction will be handled by the program you are running rather than by the CCP.
Therefore, the CCP leads a very jerky existence in memory. It is loaded when you first start CP/M. When you ask CP/M, via the CCP, to execute a program, this program can overwrite the CCP and use the memory occupied by the CCP for its own purposes. When the program you asked for has finished, CP/M needs to reload the CCP, now ready for its interaction with you. This process of reloading the CCP is known as a warm boot. In contrast with the cold boot mentioned before, the warm boot is not a complete "start from cold"; it's just a reloading of the CCP. The BDOS and BIOS are not touched.

How does a program tell CP/M that it has finished and that a warm boot must be executed? By jumping to location 0000H. While the BIOS was initializing itself during the cold boot routine, it put an instruction at location 0000H to jump to the warm boot routine, which is also in the BIOS. Once the BIOS warm boot routine
has reloaded the CCP from the disk, it will transfer control to the CCP. (The cold and warm boot routines are discussed further in Chapter 6.)

This brief description indicates that every command you enter causes a program to be loaded, the CCP to be overwritten, the program to run, and the CCP to be reloaded when the program jumps to location 0000H on completing its task. This is not completely true. Some frequently needed commands reside in the CCP. Using one of these commands means that CP/M does not have to load anything from a diskette; the programs are already in memory as part of the CCP. These commands, known as “intrinsic” or “resident” commands, are listed here with a brief description of what they do. (All of them are described more thoroughly in Chapter 4.) The “resident” commands are

- **DIR** Displays which files are on a diskette
- **ERA** Erases files from a diskette
- **REN** Changes the names of files on diskette
- **TYPE** Displays the contents of text files on the console
- **SAVE** Saves some of memory as a file on diskette
- **USER** Changes User File Group.

### Basic Disk Operating System

The BDOS is the heart of CP/M. The CCP and all of the programs that you run under CP/M talk to the BDOS for all their outside contacts. The BDOS performs such tasks as console input/output, printer output, and file management (creating, deleting, and renaming files and reading and writing sectors).

The BDOS performs all of these things in a rather detached way. It is concerned only with the logical tasks at hand rather than the detailed action of getting a sector from a diskette into memory, for example. These “low-level” operations are done by the BDOS in conjunction with the BIOS.

But how does a program work with the BDOS? By another strategically placed jump instruction in memory. Remember that the cold boot placed the jump to the BIOS warm boot routine in location 0000H. At location 0005H, it puts a jump instruction that transfers control up to the first instruction of the BDOS. Thus, any program that transfers control to location 0005H will find its way into the BDOS. Typically, programs make a CALL instruction to location 0005H so that once the BDOS has performed the task at hand, it can return to the calling program at the correct place. The program enlisting the BDOS’s help puts special values into several of the CPU registers before it makes the call to location 0005H. These values tell the BDOS what operation is required and the other values needed for the specific operation.
Basic Input/Output System

As mentioned before, the BDOS deals with the input and output of information in a detached way, unencumbered by the physical details of the computer hardware. It is the BIOS that communicates directly with the hardware, the ports, and the peripheral devices wired to them.

This separation of logical input/output in the BDOS from the physical input/output in the BIOS is one of the major reasons why CP/M is so popular. It means that the same version of CP/M can be adapted for all types of computers, regardless of the oddities of the hardware design. Digital Research will tell you that there are over 200,000 computers in the world running CP/M. Just about all of them are running identical copies of the CCP and BDOS. Only the BIOS is different. If you write a program that plays by the rules and only interacts with the BDOS to get things done, it will run on almost all of those 200,000 computers without your having to change a single line of code.

You probably noticed the word “almost” in the last paragraph. Sometimes programmers make demands of the BIOS directly rather than the BDOS. This leads to trouble. The BIOS should be off limits to your program. You need to know what it is and how it works in order to build a customized version of CP/M, but you must never write programs that talk directly to the BIOS if you want them to run on other versions of CP/M.

Now that you understand the perils of talking to the BIOS, it is safe to describe how the BDOS communicates with the BIOS. Unlike the BDOS, which has a single entry point and uses a value in a register to specify the function to be performed, the BIOS has several entry points. The first few instructions in the BIOS are all independent entry points, each taking up three bytes of memory. The BDOS will enter the BIOS at the appropriate instruction, depending on the function to be performed. This group of entry points is similar in function to a railroad marshalling yard. It directs the BDOS to the correct destination in the BIOS for the function it needs to have done. The entry point group consists of a series of JUMP instructions, each one three bytes long. The group as a whole is called the BIOS jump table, or jump vector. Each entry point has a predefined meaning. These points are detailed and will be discussed in Chapter 6.

CCP, BDOS, and BIOS Interactions

Figure 2-4 summarizes the functions that the CCP, BDOS, and BIOS perform, the ways in which these parts of CP/M communicate among themselves, and the way in which one of your programs running under CP/M interacts with the BDOS.
The CP/M Programmer's Handbook

Figure 2-4. CP/M's functional breakdown
This chapter gives you a close look at the CP/M file system. The Basic Disk Operating System (BDOS) is responsible for this file system: It keeps a directory of the files on disk, noting where data are actually stored on the disk. Because the file system automatically keeps track of this information, you can ignore the details of which tracks and sectors on the disk have data for a given file.

How CP/M Views the Disk

To manage files on the disk, CP/M works with the disk in logical terms rather than in physical terms of tracks and sectors. CP/M treats the disk as three major areas.

These are the reserved area, which contains the bootstrap program and CP/M itself; the file directory, containing one or more entries for each file stored on the disk; and the data storage area, which occupies the remainder of the disk. You will
be looking at how CP/M allocates the storage to the files as your programs create
them.

The Basic Input/Output System (BIOS) has built-in tables that tell CP/M the
respective sizes of the three areas. These are the disk definition tables, described
later in this chapter.

Allocation Blocks

Rather than work with individual 128-byte sectors, CP/M joins several of these
sectors logically to form an allocation block. Typically, an allocation block will
contain eight 128-byte sectors (which makes it 1024 or 1 K bytes long). This makes
for easier disk manipulation because the magnitude of the numbers involved is
reduced. For example, a standard 8-inch, single-density, single-sided floppy disk
has 1950 128-byte sectors; hard disks may have 120,000 or more. By using
allocation blocks that view the disk eight sectors at a time, the number of storage
units to be managed is substantially reduced. The total number is important
because numeric information is handled as 16-bit integers on the 8080 and Z80
microprocessors, and therefore the largest unsigned number possible is OFFFFH
(65,535 or 64K decimal).

Whenever CP/M refers to a specific allocation block, all that is needed is a
simple number. The first allocation block is number 0, the next is number 1, and so
on, up to the total remaining capacity of the disk.

The typical allocation block contains 1024 (1 K) bytes, or eight 128-byte
sectors. For the larger hard disks, the allocation block can be 16,384 (16K) bytes,
which is 128 128-byte sectors. CP/M is given the allocation via an entry in the disk
definition tables in the BIOS.

The size of the allocation block is not arbitrary, but it is a compromise. The
originator of the working BIOS for the system—either the manufacturer or the
operating system’s designer—chooses the size by considering the total storage
capacity of the disk. This choice is tempered by the fact that if a file is created with
only a single byte of data in it, that file would be given a complete allocation block.
Large allocation blocks can waste disk storage if there are many small files, but
they can be useful when a few very large files are called for.

This can be seen better by considering the case of a 1 K-byte allocation block. If
you create a very small file containing just a single byte of data, you will have
allocated an entire allocation block. The remaining 1023 bytes will not be used.
You can use them by adding to the file, but when you first create this one-byte file,
they will be just so much dead space. This is the problem: Each file on the disk will
normally have one partly filled allocation block. If these blocks are very large, the
amount of wasted (unused) space can be very large. With 16K-byte blocks, a
10-megabyte disk with only 3 megabytes of data on it could become logically full,
with all allocation blocks allocated.

On the other hand, when you use large allocation blocks, CP/M’s performance
is significantly improved because the BDOS refers to the file directory less
frequently. For example, it can read a 16K-byte file with only a single directory reference.

Therefore, when considering block allocation, keep the following questions in mind:

**How big is the logical disk?**
With a larger disk, you can tolerate space wasted by incomplete allocation blocks.

**What is the mean file size?**
If you anticipate many small files, use small allocation blocks so that you have a larger “supply” of blocks. If you anticipate a smaller number of large files, use larger allocation blocks to get faster file operations.

When a file is first created, it is assigned a single allocation block on the disk. Which block is assigned depends on what other files you already have on the disk and which blocks have already been allocated to them. CP/M maintains a table of which blocks are allocated and which are available. As the file accumulates more data, it will fill up the first allocation block. When this happens, CP/M will extend the file and allocate another block to it. Thus, as the file grows, it occupies more blocks. These blocks need not be adjacent to each other on the disk. The file can exist as a series of allocation blocks scattered all over the disk. However, when you need to see the entire file, CP/M presents the allocation blocks in the correct order. Thus, application programs can ignore allocation blocks. CP/M keeps track of which allocation blocks belong to each file through the file directory.

**The File Directory**

The file directory is sandwiched between the reserved area and the data storage area on the disk. The actual size of the directory is defined in the BIOS’s disk definition tables. The directory can have some binary multiple of entries in it, with one or more entries for each file that exists on the disk. For a standard 8-inch floppy diskette, there will be room for 64 directory entries; for a hard disk, 1024 entries would not be unusual. Each directory entry is 32 bytes long.

Simple arithmetic can be used to calculate how much space the directory occupies on a standard floppy diskette. For example, for a floppy disk the formula is $64 \times 32 = 2048$ bytes = 2 allocation blocks of 1024 bytes each.

The directory entry contains the name of the file along with a list of the allocation blocks currently used by the file. Clearly, a single 32-byte directory entry cannot contain all of the allocation blocks necessary for a 5-megabyte file, especially since CP/M uses only 16 bytes of the 32-byte total for storage of allocation block numbers.

**Extents**

Often CP/M will need to control files that need many allocation blocks. It does this by creating more than one directory entry. Second and subsequent directory
entries have the same file name as the first. One of the other bytes of the directory entry is used to indicate the directory entry sequence number. Each new directory entry brings with it a new supply of bytes that can be used to hold more allocation block numbers. In CP/M jargon, each directory entry is called an extent. Because the directory entry for each extent has 16 bytes for storing allocation block numbers, it can store either 16 one-byte numbers or 8 two-byte numbers. Therefore, the total number of allocation blocks possible in each extent is either 8 (for disks with more than 255 allocation blocks) or 16 (for smaller disks).

File Control Blocks

Before CP/M can do anything with a file, it has to have some control information in memory. This information is stored in a file control block, or FCB. The FCB has been described as a motel for directory entries—a place for them to reside when they are not at home on the disk. When operations on a file are complete, CP/M transforms the FCB back into a directory entry and rewrites it over the original entry. The FCB is discussed in detail at the end of this chapter.

As a summary, Figure 3-1 shows the relationships between disk sectors, allocation blocks, directory entries, and file control blocks.

The Making of a File

To reinforce what you already know about the CP/M file system, this section takes you on a “walk-through” of the events that occur when a program running under CP/M creates a file, writes data to it, and then closes the file.

Assume that a program has been loaded in memory and the CPU is about to start executing it. First, the program will declare space in memory for an FCB and will place some preset values there, the most important of which is the file name. The area in the FCB that will hold the allocation block numbers as they are assigned is initially filled with binary 0’s. Because the first allocation block that is available for file data is block 1, an allocation block number of 0 will mean that no blocks have been allocated.

The program starts executing. It makes a call to the BDOS (via location 0005H) requesting that CP/M create a file. It transfers to the BDOS the address in memory of the FCB. The BDOS then locates an available entry in the directory, creates a new entry based on the FCB in the program, and returns to the program, ready to write data to the file. Note that CP/M makes no attempt to see if there is already a file of the same name on the disk. Therefore, most real-world programs precede a request to make a file with a request to delete any existing file of the same name.

The program now starts writing data to the file, 128-byte sector by 128-byte sector. CP/M does not have any provision for writing one byte at a time. It handles data sector-by-sector only, flushing sectors to the disk as they become full.
The first time a program asks CP/M (via a BDOS request) to write a sector onto the file on the disk, the BDOS finds an unused allocation block and assigns it to the file. The number of the allocation block is placed inside the FCB in memory. As each allocation block is filled up, a new allocation block is found and assigned, and its number is added to the list of allocation blocks inside the FCB. Finally, when the FCB has no more room for allocation block numbers, the BDOS

- Writes an updated directory entry out to the disk.

Figure 3-1. The hierarchical relationship between sectors, allocation blocks, directory entries, and FCBs
The CP/M Programmer's Handbook

- Seeks out the next spare entry in the directory.
- Resets the FCB in memory to indicate that it is now working on the second extent of the file.
- Clears out the allocation block area in the FCB and waits for the next sector from the program.

Thus the process continues. New extents are automatically opened until the program determines that it is time to finish, writes the last sector out to the disk, and makes a BDOS request to close the file. The BDOS then converts the FCB into a final directory entry and writes to the directory.

Directory Entry

The directory consists of a series of 32-byte entries with one or more entries for each file on the disk. The total number of entries is a binary multiple. The actual number depends on the disk format (it will be 64 for a standard floppy disk and perhaps 2048 for a hard disk).

Figure 3-2 shows the detailed structure of a directory entry. Note that the description is actually Intel 8080 source code for the data definitions you would need in order to manipulate a directory entry. It shows a series of EQU instructions—equate instructions, used to assign values or expressions to a label, and in this case used to access an entry. It also shows a series of DS or define storage instructions used to declare storage for an entry. The comments on each line describe the function of each of the fields. Where data elements are less than a byte long, the comment identifies which bits are used.

As you study Figure 3-2, you will notice some terminology that as yet has not been discussed. This is described in detail in the sections that follow.

File User Number (Byte 0)  The least significant (low order) four bits of byte 0 in the directory entry contain a number in the range 0 to 15. This is the user number in which the file belongs. A better name for this field would have been file group number. It works like this: Suppose several users are sharing a computer system with a hard disk that cannot be removed from the system without a lot of trouble. How can each user be sure not to tamper with other users’ files? One simple way would be for each to use individual initials as the first characters of any file names. Then each could tell at a glance whether a file was another’s and avoid doing anything to anyone else’s files. A drawback of this scheme is that valuable character positions would be used in the file name, not to mention the problems resulting if several users had the same initials.

The file user number is prefixed to each file name and can be thought of as part of the name itself. When CP/M is first brought up, User 0 is the default user—the one that will be chosen unless another is designated. Any files created will go into the directory bearing the user number of 0. These files are referred to as being in user area 0. However, with a shared computer system, arrangements must be made
for multiple user areas. The USER command makes this possible. User numbers and areas can range from 0 through 15. For example, a user in area 7 would not be able to get a directory of, access, or erase files in user area 5.

This user-number byte serves a second purpose. If this byte is set to a value of 0E5H, CP/M considers that the file directory entry has been deleted and completely ignores the remaining 31 bytes of data. The number 0E5H was not chosen whimsically. When IBM first defined the standard for floppy diskettes, they chose the binary pattern 11100101 (0E5H) as a good test pattern. A new floppy diskette formatted for use has nothing but bytes of 0E5H on it. Thus, the process of erasing a file is a "logical" deletion, where only the first byte of the directory entry is changed to 0E5H. If you accidentally delete a file (and provided that no other directory activity has occurred) it can be resurrected by simply changing this first byte back to a reasonable user number. This process will be explained in Chapter 11.

**File Name and Type (Bytes 4 - 8 and 9 - 11)**  
As you can see from Figure 3-2, the file name in a directory entry is eight bytes long; the file type is three. These two fields are used to name a file unambiguously. A file name can be less than eight characters and the file type less than three, but in these cases, the unused character positions are filled with spaces.

Whenever file names and file types are written together, they are separated by a period. You do not need the period if you are not using the file type (which is the same as saying that the file type is all spaces). Some examples of file names are

READ. ME
LONGNAME.TYP
1
1.2

```assembly
 0000 = FDEUSER EQU 0 ;File user number (LS 4 bits)
 0001 = FDENAME EQU 1 ;File name (8 bytes)
 0009 = FDETYPE EQU 9 ;File type
 0009 = FDEROD EQU 9 ;Offsets for bits used in type
 000A = FDESYS EQU 10 ;Bit 7 = 1 - Read only
 000B = FDECHANGE EQU 11 ;Bit 7 = 0 = File Written To
 000C = FDEEXTENT EQU 12 ;Extent number
 000F = FDERECUSED EQU 15 ;Records used in this extent
 0010 = FDEABUSED EQU 16 ;Allocation blocks used
```

**Figure 3-2.** Data declarations for CP/M's file directory entries
A file name and type can contain the characters A through Z, 0 through 9, and some of the so-called "mark" characters such as "/", "-", and "—". You can also use lowercase letters, but be careful. When you enter commands into the system using the CCP, it converts all lowercases to uppercases, so it will never be able to find files that actually have lowercase letters in their directory entries. Avoid using the "mark" characters excessively. Ones you can use are ! @ # $ % ( ) - + /.

Characters that you must not use are < > . ; : = ? * [ ]

These characters are used by CP/M in normal command lines, so using them in file names will cause problems.

You can use odd characters in file names to your advantage. For example, if you create files with nongraphic characters in their names or types, the only way you can access these files will be from within programs. You cannot manipulate these files from the keyboard except by using ambiguous file names (described in the next section). This makes it more difficult to erase files accidentally since you cannot specify their names directly from the console.

**Ambiguous File Names** CP/M has the capability to refer to one or more file names by using special "wild card" characters in the file names. The "?" is the main wildcard character. Whenever you ask CP/M to do something related to files, it will match a "?" with any character it finds in the file name. In the extreme case, a file name and type of "????????????????????" will match with any and all file names.

As another example, all the chapters of this book were held in files called "CHAP1.DOC," "CHAP2.DOC," and so on. They were frequently referred to, however, as "CHAP??..DOC." Why two question marks? If only one had been used, for example, "CHAP?.DOC," CP/M would not have been able to match this with "CHAP10.DOC" nor any other chapter with two digits. The matching that CP/M does is strictly character-by-character.

Because typing question marks can be tedious and special attention must be paid to the exact number entered, a convenient shorthand is available. The asterisk character "*" can be used to mean "as many ?'s as you need to fill out the name or the type field." Thus, "???????????????" can be written "*.*" and "CHAP??..DOC" could also be rewritten "CHAP*.DOC."

The use of "*" is allowed only when you are entering file names from the console. The question mark notation, however, can be used for certain BDOS operations, with the file name and type field in the FCB being set to the "?" as needed.

**File Type Conventions** Although you are at liberty to think up file names without constraint, file types are subject to convention and, in one or two cases, to the mandate of CP/M itself.
The types that will cause problems if you do not use them correctly are

```
ASM
Assembly language source for the ASM program
MAC
Macro assembly language
HEX
Hexadecimal file output by assemblers
REL
Relocatable file output by assemblers
COM
Command file executed by entering its name alone
PRN
Print file written to disk as a convenience
LIB
Library file of programs
SUB
Input for CP/M SUBMIT utility program
```

Examples of conventional file types are

```
C
C source code
PAS
Pascal source code
COB
COBOL source code
FTN
FORTRAN source code
APL
APL programs
TXT
Text files
DOC
Documentation files
INT
Intermediate files
DTA
Data files
```
The CP/M Programmer's Handbook

IDX
Index files

$$$
Temporary files

The file type is also useful for keeping several copies of the same file, for example, “TEST.001,” “TEST.002,” and so on.

File Status  Each one of the states Read-Only, System, and File Changed requires only a single bit in the directory entry. To avoid using unnecessary space, they have been slotted into the three bytes used for the file type field. Since these bytes are stored as characters in ASCII (which is a seven-bit code), the most significant bit is not used for the file type and thus is available to show status.

Bit 7 of byte 9 shows Read-Only status. As its name implies, if a file is set to be Read-Only, CP/M will not allow any data to be written to the file or the file to be deleted.

If a file is declared to be System status (bit 7 of byte 10), it will not show up when you display the file directory. Nor can the file be copied from one place to another with standard CP/M utilities such as PIP unless you specifically ask the utility to do so. In normal practice, you should set your standard software tools and application programs to be both Read-Only and System status/Read-Only, so that you cannot accidentally delete them, and System status, so that they do not clutter up the directory display.

The File Changed bit (bit 7 of byte 11) is always set to 0 when you close a file to which you have been writing. This can be useful in conjunction with a file backup utility program that sets this bit to 1 whenever it makes a backup copy. Just by scanning the directory, this utility program can determine which files have changed since it was last run. The utility can be made to back up only those files that have changed. This is much easier than having to remember which files you have changed since you last made backup copies.

With a floppy disk system, there is less need to worry about backing up on a file-by-file basis — it is just as easy to copy the whole diskette. This system is useful, however, with a hard disk system with hundreds of files stored on the disk.

File Extent (Byte 12)  Each directory entry represents a file extent. Byte 12 in the directory entry identified the extent number. If you have a file of less than 16,384 bytes, you will need only one extent—number 0. If you write more information to this file, more extents will be needed. The extent number increases by 1 as each new extent is created.

The extent number is stored in the file directory because the directory entries are in random sequence. The BDOS must do a sequential search from the top of the directory to be sure of finding any given extent of a file. If the directory is large, as it could be on a hard disk system, this search can take several seconds.
Reserved Bytes 13 and 14  These bytes are used by the proprietary parts of CP/M's file system. From your point of view, they will be set to 0.

Record Number (Byte 15)  Byte 15 contains a count of the number of records (128-byte sectors) that have been used in the last partially filled allocation block referenced in this directory entry. Since CP/M creates a file sequentially, only the most recently allocated block is not completely full.

Disk Map (Bytes 16-31)  Bytes 16–31 store the allocation block numbers used by each extent. There are 16 bytes in this area. If the total number of allocation blocks (as defined by you in the BIOS disk tables) is less than 256, this area can hold as many as 16 allocation block numbers. If you have described the disk as having more than 255 allocation blocks, CP/M uses this area to store eight two-byte values. In this case allocation blocks can take on much larger values.

A directory entry can store either 8 or 16 allocation block numbers. If the file has not yet expanded to require this total number of allocation blocks, the unused positions in the entry are filled with zeros. You may think this would create a problem because it appears that several files will have been allocated block 0 over and over. In fact, there is no problem because the file directory itself always occupies block 0 (and depending on its size several of the blocks following). For all practical purposes, block 0 “does not exist,” at least for the storage of file data.

Note that if, by accident, the relationship between files and their allocation blocks is scrambled—that is, either the data in a given block is overwritten, or two or more active directory entries contain the same block number—CP/M cannot access information properly and the disk becomes worthless.

Several commercially available utility programs manipulate the directory. You can use them to inspect and change a damaged directory, reviving accidentally erased files if you need to. There are other utilities you can use to logically remove bad sectors on the disk. These utilities find the bad areas, work backward from the track and sector numbers, and compute the allocation block in which the error occurs. Once the block numbers are known, they create a dummy file, either in user area 15 or, in some cases, in an “impossible” user area (one greater than 15), that appears to “own” all the bad allocation blocks.

A good utility program protects the integrity of the directory by verifying that each allocation block is “owned” by only one directory entry.

---

Disk Definition Tables

As mentioned previously, the BIOS contains tables telling the BDOS how to view the disk storage devices that are part of the computer system. These tables are built by you. If you are using standard 8-inch, single-sided, single-density floppy
diskettes, you can use the examples in the Digital Research manual *CP/M 2 Alteration Guide*. But if you are using some other, more complex system, you must make some careful judgments. Any mistakes in the *disk definition tables* can create serious problems, especially when you try to correct diskettes created using the erroneous tables. You, as a programmer, must ensure the correctness of the tables by being careful.

One other point before looking at table structures: Because the tables exist and define a particular disk “shape” does not mean that such a disk need necessarily be connected to the system. The tables describe *logical* disks, and there is no way for the physical hardware to check whether your disk tables are correct. You may have a computer system with a single hard disk, yet describe the disk as though it were divided into several *logical* disks. CP/M will view each such “disk” independently, and they should be thought of as separate disks.

**Disk Parameter Header Table**

This table is the starting point in the disk definition tables. It is the topmost structure and contains nothing but the addresses of other structures. There is one entry in this table for each logical disk that you choose to describe. There is an entry point in the BIOS that returns the address of the parameter header table for a specific logical disk.

An example of the code needed to define a disk parameter header table is shown in Figure 3-3.

**Sector Skewing (Skewtable)**

To define sector *skewing*, also called sector *interlacing*, picture a diskette spinning in a disk drive. The sectors in the track over which the head is positioned are passing by the head one after another—sector 1, sector 2, and so on—until the diskette has turned one complete revolution. Then the sequence repeats. A standard 8-inch diskette has 26 sectors on each track, and the disk spins at 360 rpm. One turn of the diskette takes 60/360 seconds, about 166 milliseconds per track, or 6 milliseconds per sector.

Now imagine CP/M loading a program from such a diskette. The BDOS takes a finite amount of time to read and process each sector since it reads only a single sector at a time. It has to make repeated reads to load a program. By the time the BDOS has read and loaded sector n, it will be too late to read sector n + 1. This sector will have already passed by the head and will not come around for another 166 milliseconds. Proceeding in this fashion, almost 4½ seconds are needed to read one complete track.

This problem can be solved by simply numbering the sectors *logically* so that there are several physical sectors between each logical sector. This procedure, called *sector skewing or interlace*, is shown in Figure 3-4. Note that unlike physical sectors, logical sectors are numbered from 0 to 25.

Figure 3-4 shows the standard CP/M sector interlace for 8-inch, single-sided, single-density floppy diskettes. You see that logical sector 0 has six sectors between
DPBASE:  ;Base of the parameter header
0000 1000  DW  SKEWTABLE  ;used to access the headers
0002 0000  DW  0  ;sector conversion table
0004 0000  DW  0  ;Scratch pad areas used by CP/M
0006 0000  DW  0
0008 2A00  DW  DIRBUF  ;Pointer to Directory Buffer
000A A000  DW  DPBO  ;Pointer to disk parameter block
000C B900  DW  WACD  ;Pointer to work area (used to
000E C900  DW  ALVECO  ;check for changed diskettes)

The following equates would normally be derived from
values found in the disk parameter Block. They are shown here only for the sake of completeness.
003F = NODE  EQU  63  ;Number of directory entries
00F2 = NOAB  EQU  242  ;Number of allocation blocks

Example data definitions for those objects pointed
to by the disk parameter header

SKEWTABLE:  ;Sector skew table.
0010 017D013  DB  01,07,13,19  ;Logical sectors 0,1,2,3
0014 19050B11  DB  25,05,11,17  ;4,5,6,7
0018 1703090F  DB  23,03,09,15  ;8,9,10,11
001C 1502080E  DB  21,02,08,14  ;12,13,14,15
0020 141A060C  DB  20,26,06,12  ;16,17,18,19
0024 121B040A  DB  18,24,04,10  ;20,21,22,23
0028 1016  DB  16,22  124,25
002A  DIRBUF:  DS  128  ;Directory buffer
00AA  DPBO:  DS  15  ;Disk parameter block
This is normally a table of
constants.
A dummy definition is shown
here.
00B9  WACD:  DS  (NODE+1)/4  ;Work area to check directory
00C9  ALVECO:  DS  (NOAB/8)+1  ;Allocation vector #0

Figure 3-3. Data declarations for a disk parameter header

it and logical sector 1. There is a similar gap between each of the logical sectors, so
that there are six “sector times” (about 38 milliseconds) between two adjacent
logical sectors. This gives ample time for the software to access each sector.
However, several revolutions of the disk are still necessary to read every sector in
turn. In Figure 3-4, the vertical columns of logical sectors show which sectors are
read on each successive revolution of the diskette.

The wrong interlace can strongly affect performance. It is not a gradual effect,
either; if you “miss” the interlace, the perceived performance will be very slow. In
the example given here, six turns of the diskette are needed to read the whole
track — this lasts one second as opposed to 4½ without any interlacing. But don’t
imagine that you can change the interlace with impunity; files written with one
interlace stay that way. You must be sure to read them back with the same interlace
with which they were written.
Some disk controllers can simplify this procedure. When you format the diskette, they can write the sector addresses onto the diskette with the interlace already built in. When CP/M requests sector n, the controller's electronics wait until they see the requested sector's header fly by. They then initiate the read or write operation. In this case you can embed the interlace right into the formatting of the diskette.

Because the wrong interlace gives terrible performance, it is easy to know when you have the right one. Some programmers use the time required to format a diskette as the performance criterion to optimize the interlace. This is not good practice because under normal circumstances you will spend very little time formatting diskettes. The time spent loading a program would be a better arbiter, since far more time is spent doing this. You might argue that doing a file update would be even more representative, but most updates produce slow and sporadic disk activity. This kind of disk usage is not suitable for setting the correct interlace.

Hard disks do not present any problem for sector skewing. They spin at 3600 rpm or faster, and at that speed there simply is no interlace that will help. Some

<table>
<thead>
<tr>
<th>Physical Sector</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>22</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td>18</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td>23</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td>10</td>
<td></td>
<td>19</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td>1</td>
<td></td>
<td>14</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td>5</td>
<td></td>
<td>18</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td>1</td>
<td></td>
<td>14</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td>6</td>
<td></td>
<td>19</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td>2</td>
<td></td>
<td>23</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td>11</td>
<td></td>
<td>24</td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td>2</td>
<td></td>
<td>19</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td>7</td>
<td></td>
<td>20</td>
</tr>
<tr>
<td>13</td>
<td></td>
<td></td>
<td>3</td>
<td></td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td></td>
<td></td>
<td></td>
<td>12</td>
<td></td>
<td>25</td>
</tr>
<tr>
<td>15</td>
<td></td>
<td></td>
<td>4</td>
<td></td>
<td>25</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td>8</td>
<td></td>
<td>21</td>
</tr>
<tr>
<td>17</td>
<td></td>
<td></td>
<td>5</td>
<td></td>
<td>17</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td></td>
<td></td>
<td></td>
<td>13</td>
<td></td>
<td>21</td>
</tr>
<tr>
<td>19</td>
<td></td>
<td></td>
<td>6</td>
<td></td>
<td>21</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td></td>
<td></td>
<td></td>
<td>14</td>
<td></td>
<td>21</td>
</tr>
<tr>
<td>21</td>
<td></td>
<td></td>
<td>2</td>
<td></td>
<td>21</td>
<td></td>
</tr>
<tr>
<td>22</td>
<td></td>
<td></td>
<td></td>
<td>7</td>
<td></td>
<td>21</td>
</tr>
<tr>
<td>23</td>
<td></td>
<td></td>
<td>3</td>
<td></td>
<td>21</td>
<td></td>
</tr>
<tr>
<td>24</td>
<td></td>
<td></td>
<td></td>
<td>13</td>
<td></td>
<td>21</td>
</tr>
<tr>
<td>25</td>
<td></td>
<td></td>
<td>4</td>
<td></td>
<td>21</td>
<td></td>
</tr>
<tr>
<td>26</td>
<td></td>
<td></td>
<td></td>
<td>8</td>
<td></td>
<td>21</td>
</tr>
</tbody>
</table>

**Note:** Additional sector between logical sectors 12 and 13

*Figure 3-4. Physical to logical sector skewing*
tricks can be played to improve the performance of a hard disk—these will be discussed in the section called "Special Considerations for Hard Disks," later in this chapter.

To better understand these theories, study an example of the standard interlace table, or skewtable. Bear in mind that the code that will access this table will first be given a logical sector. It will then have to return the appropriate physical sector.

Figure 3-5 shows the code for the skew table and the code that can be used to access the table. The table is indexed by a logical sector and the corresponding table entry is the physical sector. You can see that the code assumes that the first logical sector assigned by CP/M will be sector number 0. Hence there is no need to subtract 1 from the sector number before using it as a table subscript.

**Unused Areas in the Disk Parameter Header Table** The three words shown as 0's in Figure 3-3 are used by CP/M as temporary variables during disk operations.

**Directory Buffer (DIRBUF)** The directory buffer is a 128-byte area used by CP/M to store a sector from the directory while processing directory entries. You only need one directory buffer; it can be shared by all of the logical disks in the system.

**Disk Parameter Block (DPBO)** The disk parameter block describes the particular characteristics of each logical disk. In general, you will need a separate parameter block for each type of logical disk. Logical disks can share a parameter block only if their
characteristics are identical. You can, for example, use a single parameter block to
describe all of the single-sided, single-density diskette drives that you have in the
system. However, you would need another parameter block to describe double­
sided, double-density diskette drives. It is also rare to be able to share parameter
blocks when a physical hard disk is split up into several logical disks. You will
understand why after looking at the contents of a parameter block, described later
in this chapter.

**Work Area to Check for Changed Diskettes (WACD)** One of the major problems that
CP/M faces when working with removable media such as floppy diskettes is that
the computer operator, without any warning, can open the diskette drive and
substitute a different diskette. On early versions of CP/M, this resulted in the
newly inserted diskette being overwritten with data from the original diskette.

With the current version of CP/M, you can request that CP/M check if the
diskette has been changed. Given this request, CP/M examines the directory
entries whenever it has worked on the directory and, if it detects that the diskette
has been changed, declares the whole diskette to be Read-Only status and inhibits
any further writing to the diskette. This status will be in effect until the next warm
boot operation occurs. A warm boot occurs whenever a program terminates or a
CONTROL-C is entered to the CCP, resetting the operating system.

The value of WACD is the address of a buffer, or temporary storage area, that
CP/M can use to check the directory. The length of this buffer is defined (some­
what out of place) in the disk parameter block.

**Allocation Vector (ALVEC0)** CP/M views each disk as a set of allocation blocks, assign­
ning blocks to individual files as those files are created or expanded, and relin­
quishing blocks as files are deleted.

CP/M needs some mechanism for keeping track of which blocks are used and
which are free. It uses the allocation vector to form a bit map, with each bit in the
map corresponding to a specific allocation block. The most significant bit (bit 7) in
the first byte corresponds to the first allocation block, number 0. Bit 6 corresponds
to block 1, and so on for the entire disk.

Whenever you request CP/M to use a logical disk, CP/M will log in the disk.
This consists of reading down the file directory and, for each active entry or extent,
interacting with the allocation blocks “owned” by that particular file extent. For
each block number in the extent, the corresponding bit in the allocation vector is
set to 1. At the end of this process, the allocation vector will accurately represent a
map of which blocks are in use and which are free.

When CP/M goes looking for an unused allocation block, it tries to find one
near the last one used, to keep the file from becoming too fragmented.

In order to reserve enough space for the allocation vector, you need to reserve
one bit for each allocation block. Computing the number of allocation blocks is
discussed in the section “Maximum Allocation Block Number,” later in this
chapter.
Chapter 3: The CP/M File System

Disk Parameter Block

The disk parameter block in early versions of CP/M was built into the BDOS and was a closely guarded secret of the CP/M file system. To make CP/M adaptable to hard disk systems, Digital Research decided to move the parameter blocks out into the BIOS where everyone could adapt them. Because of the proprietary nature of CP/M's file system, you will still see several odd-looking fields, and you may find the explanation given here somewhat superficial. However, the lack of explanation in no way detracts from your ability to use CP/M as a tool.

Figure 3-6 shows the code necessary to define a parameter block for 8-inch, single-sided diskettes. This table is pointed to by—that is, its address is given in—an entry in the disk parameter header. Each of the entries shown in the disk parameter block is explained in the following sections.

Sectors Per Track This is the number of 128-byte sectors per track. The standard diskette shown in the example has 26 sectors. As you can see, simply telling CP/M that there are 26 sectors per track does not indicate whether the first sector is numbered 0 or 1. CP/M assumes that the first sector is 0; it is left to a sector translate subroutine to decipher which physical sector this corresponds to.

Hard disks normally have sector sizes larger than 128 bytes. This is discussed in the section on considerations for hard disks.

Block Shift, Block Mask, and Extent Mask These mysteriously named fields are used internally by CP/M during disk file operations. The values that you specify for them depend primarily on the size of the allocation block that you want.

Allocation block size can vary from 1024 bytes (1K) to 16,384 bytes (16K). There is a distinct trade-off between these two extremes, as discussed in the section on allocation blocks at the beginning of this chapter.

An allocation block size of 1024 (1K) bytes is suggested for floppy diskettes with capacities up to 1 megabyte, and a block size of 4096 (4K) bytes for larger floppy or hard disks.

```
0000 1A00 DW 26  ;Sectors per track
0001 0003 DB 3   ;Block shift
0002 0007 DB 7   ;Block mask
0003 0003 DB 3   ;Extent mask
0004 3F00 DW 242 ;Max. allocation block number
0005 F000 DW 63  ;Number of directory entries
0006 0C DB 1100080008 ;Bit map for allocation blocks
0007 00 DB 0000800000000000 ;used for directory
0008 0B 1000 DW 16  ;No. of bytes in dir. check buffer
0009 0D 0200 DW 2   ;No. of tracks before directory
```

Figure 3-6. Data declarations for the disk parameter block for standard diskettes
If you can define which block size you wish to use, you can now select the values for the block shift and the block mask from Table 3-1.

**Table 3-1. Block Shift and Mask Value**

<table>
<thead>
<tr>
<th>Allocation Block Size</th>
<th>Block Shift</th>
<th>Block Mask</th>
</tr>
</thead>
<tbody>
<tr>
<td>1,024</td>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>2,048</td>
<td>4</td>
<td>15</td>
</tr>
<tr>
<td>4,096</td>
<td>5</td>
<td>31</td>
</tr>
<tr>
<td>8,192</td>
<td>6</td>
<td>63</td>
</tr>
<tr>
<td>16,384</td>
<td>7</td>
<td>127</td>
</tr>
</tbody>
</table>

Select your required allocation block size from the left-hand column. This tells you which values of block shift and mask to enter into the disk parameter block.

The last of these three variables, the *extent mask*, depends not only on the block size but also on the total storage capacity of the logical disk. This latter consideration is only important for computing whether or not there will be fewer than 256 allocation blocks on the logical disk. Just divide the chosen allocation block size into the capacity of the logical disk and check whether you will have fewer than 256 blocks.

Keeping this answer and the allocation block size in mind, refer to Table 3-2 for the appropriate value for the extent mask field of the parameter block. Select the appropriate line according to the allocation block size you have chosen. Then, depending on the total number of allocation blocks in the logical disk, select the extent mask from the appropriate column.

**Table 3-2. Extent Mask Value**

<table>
<thead>
<tr>
<th>Allocation Block Size</th>
<th>Number of Allocation Blocks</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1 to 255</td>
</tr>
<tr>
<td>1,024</td>
<td>0</td>
</tr>
<tr>
<td>2,048</td>
<td>1</td>
</tr>
<tr>
<td>4,096</td>
<td>3</td>
</tr>
<tr>
<td>8,192</td>
<td>7</td>
</tr>
<tr>
<td>16,384</td>
<td>15</td>
</tr>
</tbody>
</table>

**Maximum Allocation Block Number** This value is the *number* of the last allocation block in the logical disk. As the first block number is 0, this value is *one less* than the total number of allocation blocks on the disk. Where only a partial allocation block exists, the number of blocks is rounded down.
Chapter 3: The CP/M File System

Figure 3-7 has an example for standard 8-inch, single-sided, single-density diskettes. Note that CP/M uses two reserved tracks on this diskette format.

**Number of Directory Entries Minus 1**  Do not confuse this entry with the number of files that can be stored on the logical disk; it is only the number of *entries* (minus one). Each extent of each file takes one directory entry, so very large files will consume several entries. Also note that the value in the table is *one less* than the number of entries.

On a standard 8-inch diskette, the value is 63 entries. On a hard disk, you may want to use 1023 or even 2047. Remember that CP/M performs a sequential scan down the directory and this takes a noticeable amount of time. Therefore, you should balance the number of logical disks with your estimate of the largest file size that you wish to support.

As a final note, make sure to choose a number of entries that fits evenly into one or more allocation blocks. Each directory entry needs 32 bytes, so you can compute the number of bytes required. Make sure this number can be divided by your chosen allocation block size without a remainder.

**Allocation Blocks for the Directory**  This is a strange value; it is not a number, but a bit map. Looking at Figure 3-6, you see the example value written out in full as a binary value to illustrate how this value is defined. This 16-bit value has a bit set to 1 for each allocation block that is to be used for the file directory.

This value is derived from the number of directory entries you want to have on the disk and the size of the allocation block you want to use. One given, or

<table>
<thead>
<tr>
<th>Physical characteristics:</th>
<th>Calculate:</th>
</tr>
</thead>
<tbody>
<tr>
<td>77 Tracks/Diskette</td>
<td>77 Tracks/Diskette</td>
</tr>
<tr>
<td>26 Sectors/Track</td>
<td>- 2 Tracks Reserved for CP/M</td>
</tr>
<tr>
<td>128 Bytes/Sector</td>
<td>75 Tracks for File Storage</td>
</tr>
<tr>
<td>2 Tracks Reserved for CP/M</td>
<td>×26 Number of Sectors</td>
</tr>
<tr>
<td>1024 Bytes/Allocation Block</td>
<td>1950 Sectors for File Storage</td>
</tr>
<tr>
<td></td>
<td>×128 Bytes per Sector</td>
</tr>
<tr>
<td></td>
<td>249,600 Bytes for File Storage</td>
</tr>
<tr>
<td></td>
<td>+ 1024 Bytes/Allocation Block</td>
</tr>
<tr>
<td></td>
<td>243.75 Total Number of Allocation Blocks</td>
</tr>
<tr>
<td></td>
<td>242 Number of the last allocation block (rounded and based on first block being Block 0)</td>
</tr>
</tbody>
</table>

*Figure 3-7.* Computing the maximum allocation block number for standard diskettes
constant, in this derivation is that the size of each directory entry is 32 bytes.

In the example, 64 entries are required (remember the number shown is one less than the required value). Each entry has 32 bytes. The total number of bytes required for the directory thus is 64 times 32, or 2048 bytes. Dividing this by the allocation block size of 1024 indicates that two allocation blocks must be reserved for the directory. You can see that the example value shows this by setting the two most significant bits of the 16-bit value.

As a word of warning, do not be tempted to declare this value using a DW (define word) pseudo-operation. Doing so will store the value byte-reversed.

**Size of Buffer for Directory Checking**  
As mentioned before in the discussion of the disk parameter header, CP/M can be requested to check directory entries whenever it is working on the directory. In order to do this, CP/M needs a buffer area, called the work area to check for changed diskettes, or WACD, in which it can hold working variables that keep a compressed record of what is on the directory. The length of this buffer area is kept in the disk parameter block; its address is specified in the parameter header. Because CP/M keeps a compressed record of the directory, you need only provide one byte for every four directory entries. You can see in Figure 3-6 that 16 bytes are specified to keep track of the 64 directory entries.

**Number of Tracks Before the Directory**  
Figure 3-8 shows the layout of CP/M on a standard floppy diskette. You will see that the first two tracks are reserved, containing the initial bootstrap code and CP/M itself. Hence the example in Figure 3-6, giving the code for a standard floppy disk, shows two reserved tracks (the number of tracks before the directory).

This track offset value, as it is sometimes called, provides a convenient method of dividing a physical disk into several logical disks.

**Special Considerations for Hard Disks**

If you want to run CP/M on a hard disk, you must provide code and build tables that make CP/M work as if it were running on a very large floppy disk. You must even include 128-byte sectors. However, this is not difficult to do.

To adapt hard disks to the 128-byte sector size, you must provide code in the disk driver in your BIOS that will present the illusion of reading and writing 128-byte sectors even though it is really working on sectors of 512 bytes. This code is called the blocking/deblocking routine.

If hard disks have sector sizes other than 128 bytes, what of the number of sectors per track, and the number of tracks?

Hard disks come in all sizes. The situation is further confused by the disk controllers, the hardware that controls the disk. In many cases, you can think of the hard disk as just a series of sectors without any tracks at all. The controller, given a relative sector number by the BIOS, can translate this sector number into which track, read/write head (if there is more than one platter), and sector are actually being referenced.
Furthermore, most hard disks rotate so rapidly that there is nothing to be gained by using a sector-skewing algorithm. There is just no way to read more than one physical sector per revolution; there is not enough time.

In many cases it is desirable to divide up a single, physical hard disk into several smaller, logical disks. This is done mainly for performance reasons: Several smaller disks, along with smaller directories, result in faster file operations.

The disk parameter header will have 0's for the skewtable entry and the pointer to the WACD buffer. In general, hard disks cannot be changed, at least not without turning off the power and swapping the entire disk drive. If you are using one of the new generation of removable hard disks, you will need to use the directory checking feature of CP/M.

The disk parameter block for a hard disk will be quite different from that used for a floppy diskette. The number of sectors per track needs careful consideration. Remember, this is the number of 128-byte sectors. The conversion from the physical sector size to 128-byte sectors will be done in the disk driver in the BIOS.
If you have a disk controller that works in terms of sectors and tracks, all you need do is compute the number of 128-byte sectors on each track. Multiply the number of physical sectors per track by their size in bytes and then divide the product by 128 to give the result as the number of 128-byte sectors per physical track.

But what of those controllers that view their hard disks as a series of sectors without reference to tracks? They obscure the fact that the sectors are arranged on concentric tracks on the disk's surface. In this case, you can play a trick on CP/M. You can set the "sectors per track" value to the number of 128-byte sectors that will fit into one of the disk's physical sectors. To do this, divide the physical sector size by 128. For example, a 512-byte physical sector size will give an answer of four 128-byte sectors per "track." You can now view the hard disk as having as many "tracks" as there are physical sectors. By using this method, you avoid having to do any kind of arithmetic on CP/M's sector numbers; the "track" number to which CP/M will ask your BIOS to move the disk heads will be the relative physical sector. Once the controller has read this physical sector for you, you can look at the 128-byte sector number, which will be 0, 1, 2, or 3 (for a 512-byte physical sector) in order to select which 128 bytes need to be moved in or out of the disk buffer.

The block shift, block mask, and extent mask will be computed as before. Use a 4096-byte allocation block size. This will yield a value of 5 for the block shift, 31 for the block mask, and given that you will have more than 256 allocation blocks for each logical disk, an extent mask value of 1.

The maximum allocation block number will be computed as before. Keep clear in your mind whether you are working with the number of physical sectors (which will be larger than 128 bytes) or with 128-byte sectors when you are computing the storage capacity of each logical disk.

The number of directory entries (less 1) is best set to 511 for logical disks of 1 megabyte and either 1023 or 2047 for larger disks. Remember that under CP/M version 2 you cannot have a logical disk larger than 8 megabytes.

The allocation blocks for the directory are also computed as described for floppy disks.

As a rule, the size of the directory check buffer (WADC) will be set to 0, since there is no need to use this feature on hard disk systems with fixed media.

The number of tracks before the directory (track offset) can be used to divide up the physical disk into smaller logical disks, as shown in Figure 3-9.

There is no rule that says the tracks before a logical disk's directory cannot be used to contain other complete logical disks. You can see this in Figure 3-9. CP/M behaves as if each logical disk starts at track 0 (and indeed they do), but by specifying increasingly larger numbers of tracks before each directory, the logical disks can be staggered across the available space on the physical disk.

Figure 3-10 shows the calculations involved in the first phase of building disk parameter blocks for the hard disk shown in Figure 3-9. The physical characteristics are those imposed by the design of the hard disk. As a programmer, you do not have any control over these; however, you can choose how much of the physical
Chapter 3: The CP/M File System

Figure 3-9. Dividing hard disks into logical disks

disk is assigned to each logical disk, the allocation block size, and the number of directory entries. You can see that logical disk A is much smaller than disks B and C, and that B and C are the same size. Disk A will be the systems disk from which most programs will be loaded, so its smaller directory size will make program loading much faster. The allocation block size for disk A is also smaller in order to reduce the amount of space wasted in partially filled allocation blocks.

Figure 3-10 also shows the calculations involved in computing the maximum allocation block number. Again, note that once the total number of allocation blocks has been computed, it is necessary to round it down in the case of any fractional components and then subtract 1 to get the maximum number (the first block being 0).

Figure 3-11 shows the actual values that will be put into the parameter blocks. It is assumed that the disk controller is one of those types that view the physical disk as a series of contiguous sectors and make no reference to tracks; the internal electronics and firmware in the controller take care of these details. For this reason, CP/M is told that each physical sector is a “track” in CP/M’s terms. Each “track” has 512 bytes and can therefore store four 128-byte sectors. You can see this is the value that is in the sectors/track field.

The block shift and mask values are obtained from Table 3-1, using the allocation block size previously chosen. Then, with both the allocation block size and the maximum number of allocation blocks (see Figure 3-10), the extent mask can be obtained from Table 3-2. You can see in Figure 3-11 that extent mask values of 1 were obtained for all three logical disks even though two different allocation block sizes have been chosen, and even though disk A has less than 256 blocks and disks B and C have more.
Physical Characteristics:

- 364 Tracks/Disk
- 20 Sectors/Track
- 512 Bytes/Sector
- 10,240 Bytes/Track

Choose Logical Characteristics:

- Reserved Area
- Disk A:
  - Tracks: 48
  - Allocation Block Size: n/a
- Disk B:
  - Tracks: 153
  - Allocation Block Size: 2048
- Disk C:
  - Tracks: 153
  - Allocation Block Size: 4096

Calculate:

<table>
<thead>
<tr>
<th>Tracks</th>
<th>Allocation Block Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>A: 48</td>
<td>2048</td>
</tr>
<tr>
<td>B: 153</td>
<td>4096</td>
</tr>
<tr>
<td>C: 153</td>
<td>4096</td>
</tr>
</tbody>
</table>

**Figure 3-10.** Computing the maximum allocation block number for a hard disk

<table>
<thead>
<tr>
<th>DPBA</th>
<th>DPBB</th>
<th>DPBC</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>15</td>
<td>31</td>
<td>31</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>239</td>
<td>381</td>
<td>381</td>
</tr>
<tr>
<td>2000</td>
<td>1023</td>
<td>1023</td>
</tr>
</tbody>
</table>

**Figure 3-11.** Disk parameter tables for a hard disk

The bit map showing how many allocation blocks are required to hold the file directory is computed by multiplying the number of directory entries by 32 and dividing the product by the allocation block size. This yields results of 4 for disk A and 8 for disks B and C. As you can see, the bit maps have the appropriate number of bits set.

Since most of the hard disks on the market today do not have removable media, the lengths of the directory checking buffer are set to 0.

The number of "tracks" before the directory requires a final touch of skull-duggery. Having already indicated to CP/M that each "track" has four sectors, you need to continue in the same vein and express the number of real tracks before the directories in units of 512-byte physical sectors.

As a final note, if you are specifying these parameter blocks for a disk controller that requires you to communicate with it in terms of physical tracks and 128-byte sectors, then the number of sectors per track must be set to 80 (twenty...
Chapter 3: The CP/M File System

512-byte sectors per physical track). You would also have to change the number of tracks before the directory by stating the number of physical tracks (shown in parentheses on Figure 3-11).

Adding Additional Information to the Parameter Block

Normally, some additional information must be associated with each logical disk. For example, in a system that has several physical disks, you need to identify where each logical disk resides. You may also want to identify some other physical parameters, disk drive types, I/O port numbers, and addresses of driver subroutines.

You may be tempted to extend the disk parameter header entry because there is a separate header entry for each logical disk. But the disk parameter header is exactly 16 bytes long; adding more bytes makes the arithmetic that we need to use in the BIOS awkward. The best place to put these kinds of information is to prefix them to the front of each disk parameter block. The label at the front of the block must be left in the same place lest CP/M become confused. Only special additional code that you write will be "smart" enough to look in front of the block in order to find the additional parameter information.

File Organizations

CP/M supports two types of files: sequential and random. CP/M views both types as made up of a series of 128-byte records. Note that in CP/M's terms, a record is the same as a 128-byte sector. This terminology sometimes gets in the way. It may help to think of 128-byte sectors as physical records. Applications programs manipulate logical records that bear little or no relation to these physical records. There is code in the applications programs to manipulate logical records.

CP/M does not impose any restrictions on the contents of a file. In many cases, though, certain conventions are used when textual data is stored. Each line of text is terminated by ASCII CARRIAGE RETURN and LINE FEED. The last sector of a text file is filled with ASCII SUB characters; in hexadecimal this is 1AH.

File Control Blocks

In order to get CP/M to work on a file, you need to provide a structure in which both you and the BDOS can keep relevant details about the file, its name and type, and so on. The file control block (FCB) is a derivative of the file directory entry, as you can see in Figure 3-12. This figure shows both a series of equates that can be used to access an entry and a series of DB (define byte) instructions to declare an example.

The first difference you will see between the file directory entry and the FCB is that the very first byte is serving a different purpose. In the FCB, it is used to
specify on which disk the file is to be found. You may recall that in the directory, this byte indicates the user number for a given entry. When you are actually processing files, the current user number is set either by the operator in a command from the console or by a BDOS function call; this predefines which subset of files in the directory will be processed. Therefore, the FCB does not need to keep track of the user number.

The disk number in the FCB’s first byte is stored in an odd way. A value of 0 indicates to CP/M that it should look for the file on the current default disk. This default disk is selected either by an entry from the console or by making a specific BDOS call from within a program. In general, the default disk should be preset to the disk that contains the set of programs with which you are working. This avoids unnecessary typing on the keyboard when you want to load a program.

A disk number value other than 0 represents a letter of the alphabet based on a simple codification scheme of A = 1, B = 2, and so on.

As you can see from Figure 3-12, the file name and type must be set to the required values, and for sequential file processing, the remainder of the FCB can be set to zeros. Strictly speaking, the last three bytes of the FCB (the random record number and the random record overflow byte) need not even be declared if you are never going to process the file randomly.

This raises a subtle conceptual point. Random files are only random files because you process them randomly. Though this sounds like a truism, what it means is that CP/M’s files are not intrinsically random or sequential. What they are depends on how you choose to process them at any given point. Therefore,

```
0000 = FCBE*DISK EQU 0 ;Disk drive (0 = default, 1=A)
0001 = FCBE*NAME EQU 1 ;File name (8 bytes)
0009 = FCBE*TYPE EQU 9 ;File type
0009 = FCBE*RO EQU 9 ;Offsets for bits used in type
000A = FCBE*SYS EQU 10 ;Bit 7 = 1 – system status
000B = FCBE*CHANGE EQU 11 ;Bit 7 = 0 – file written to
000C = FCBE*EXTENT EQU 12 ;Extent number
000F = FCBE*REUSED EQU 15 ;Records used in this extent
0010 = FCBE*ABUSED EQU 16 ;Allocation blocks used
0020 = FCBE*SEQREC EQU 32 ;Sequential rec. to read/write
0021 = FCBE*RANREC EQU 33 ;Random rec. to read/write
0023 = FCBE*RANREC0 EQU 35 ;Random rec. overflow byte (MS)
```

Figure 3-12. Data declarations for the FCB
while the manner in which you process them will be different, there is nothing special built into the file that predicates how it will be used.

Sequential Files

A sequential file begins at the beginning and ends at the end. You can view it as a contiguous series of 128-byte “records.”

In order to create a sequential file, you must declare a file control block with the required file name and type and request the BDOS to create the file. You can then request the BDOS to write, “record” by “record” (really 128-byte sector by 128-byte sector) into the file. The BDOS will take care of opening up new extents as it needs to. When you have written out all the data, you must make a BDOS request to close the file.

To read an existing file, you also need an FCB with the required file name and type declared. You then make a BDOS request to open the file for processing and a series of Read Sequential requests, each one bringing in the next “record” until either your program detects an end of file condition (by examining the data coming in from the file) or the BDOS discovers that there are no more sectors in the file to read. There is no need to close a file from which you have been reading data — but do close it. This is not necessary if you are going to run the program only under CP/M, but it is necessary if you want to run under MP/M (the multiuser version of CP/M).

What if you need to append further information to an existing file? One option is to create a new file, copy the existing file to the new one, and then start adding data to the end of the new file. Fortunately, with CP/M this is not necessary. In the FCB used to read a file, the name and the type were specified, but you can also specify the extent number. If you do, the BDOS will proceed to open (if it can find it) the extent number that you are asking for. If the BDOS opens the extent successfully, all you need do is check if the number of records used in the extent (held in the field FCB$RECUSED) is less than 128 (80H). This indicates the extent is not full. By taking this record number and placing it into the FCB$SEQREC (sequential record number) byte in the FCB, you can make CP/M jump ahead and start writing from the effective end of the file.

Random Files

Random files use a simple variation of the technique described above. The main difference is that the random record number must be set in the FCB. The BDOS automatically keeps track of file extents during Read/Write Random requests. (These requests are explained more fully in Chapter 5.)

Conceptually, random files need a small mind-twist. After creating a file as described earlier, you must set the random record number in the FCB before each Write Random request. This is the two-byte value called FCB$RANREC in Figure 3-12. Then, when you give the Write Random request to the BDOS, it will
look at the record number; compute in which extent the record must exist; if necessary, create the directory entry for the extent; and finally, write out the data record. Using this scheme, you can dart backward and forward in the file putting records at random throughout the file space, with CP/M creating the necessary directory entries each time you venture into a part of the file that has not yet been written to.

The same technique is used to read a file randomly. You set the random record number in the FCB and then give a system call to the BDOS to open the correct extent and read the data. The BDOS will return an error if it cannot find the required extent or if the particular record is nonexistent.

Problems lie in wait for the unwary. Before starting to do any random reading or writing, you must open up the file at extent 0 even though this extent may not contain any data records. For a new file, this can be done with the Create File request, and for an existing file with the normal Open File request. If you create a sparse file, one that has gaps in between the data, you may have some problems manipulating the file. It will appear to have several extents, each one being partially full. This will fool some programs that normally process sequential files; they don’t expect to see a partial extent except at the end of a file, and may treat the wrong spot as the end.
The Console Command Processor (CCP)

The Console Command Processor processes commands that you enter from the console. As you may recall from the brief overview in Chapter 2, the CCP is loaded into memory immediately below the BDOS. In practice, many programs deliberately overwrite the CCP in order to use the memory it normally occupies. This gives these programs an additional 800H bytes (2K bytes).

When one of these “transient programs” terminates, it relinquishes control to the BIOS, which in turn reloads a fresh copy of the CCP from the system tracks of the disk back into memory and then transfers control to it. Consequently, the CCP leads a sporadic existence—an endless series of being loaded into memory, accepting a command from you at the console, being overwritten by the program.
you requested to be loaded, and then being brought back into memory when the program terminates.

This chapter discusses what the CCP does for you in those brief periods when it is in memory.

Functions of the CCP

Simply put, once the CCP has control of the machine, so do you. The CCP announces its presence by displaying a prompt of two characters: a letter of the alphabet for the current default disk drive and a “greater than” sign. In the example A>, the A tells you that the default disk drive is currently set to be logical drive A, and the “>,” that the message was output by the CCP.

Once you see the prompt, the CCP is ready for you to enter a command line. A command line consists of two major parts: the name of the command and, optionally, some values for the command. This last part is known as the command tail.

The command itself can be one of two things: either the name of a file or the name of one of the frequently used commands built into the CCP.

If you enter the name of one of the built-in commands, the CCP does not need to go out to the disk system in order to load the command for execution. The executable code is already inside the CCP.

If the name of the command you entered does not match any of the built-in commands (the CCP has a table of their names), the CCP will search the appropriate logical disk drive for a file with a matching name and a file type of “COM” (which is short for command). You do not enter “.COM” when invoking a command — the CCP assumes a file type of “COM.”

If you do not precede the name of the COM file with a logical disk drive specification, the CCP will search the current default drive. If you have prefixed the COM file’s name with a specific logical drive, the CCP will look only on that drive for the program. For example, the command MYPROG will cause the CCP to look for a file called “MYPROG.COM” on the current default drive, whereas C:MYPROG would make the CCP search only on drive C.

If you enter a command name that matches neither the CCP’s built-in command table nor the name of any COM file on the specified disk, the CCP will output the command name followed by a question mark, indicating it is unable to find the file.

Editing the CCP Command Line

The CCP uses a line buffer to store what you type until you strike either a CARRIAGE RETURN or a LINE FEED. If you make an error or change your mind, you can modify the incomplete command, even to the point of discarding it.
You edit the command line by entering *control characters* from the console. Control characters are designated either by the combination of keys required to generate them from the keyboard or by their official name in the ASCII character set. For example, `CONTROL-J` is also known as CARRIAGE RETURN or CR.

Whenever CP/M has to represent control characters, the convention is to indicate the "control" aspect of a character with a caret ("^"). For example, `CONTROL-A` will appear as "^A", `CONTROL-Z` as "^Z", and so on. But if you press the `CONTROL` key with the normal shift key and the "6" key, this will produce a `CONTROL-^` or "^^". The representation of control keys with the caret is only necessary when outputting to the console or the printer — internally, these characters are held as their appropriate binary values.

**CONTROL-C: Warm Boot**  
If you enter a `CONTROL-C` as the first character of a command line, the CCP will initiate a warm boot operation. This operation resets CP/M completely, including the disk system. A fresh copy of the CCP is loaded into memory and the file directory of the current default disk drive is scanned, rebuilding the allocation bit map held in the BIOS (as discussed in Chapter 3).

The only time you would initiate a warm boot operation is after you have changed a diskette (or a disk, if you have removable media hard disks). Thus, CP/M will reset the disk system.

Note that a `CONTROL-C` only initiates a warm boot if it is the first character on a command line. If you enter it in any other position, the CCP will just echo it to the screen as "^C". If you have already entered several characters on a command line, use `CONTROL-U` or `CONTROL-X` to cancel the line, and then use `CONTROL-C` to initiate a warm boot. You can tell a warm boot has occurred because there will be a noticeable pause after the `CONTROL-C` before the next prompt is displayed. The system needs a finite length of time to scan the file directory and rebuild the allocation bit map.

**CONTROL-E: Physical End-of-Line**  
The `CONTROL-E` command is a relic of the days of the teletype and terminals that did not perform an automatic carriage return and line feed when the cursor went off the screen to the right. When you type a `CONTROL-E`, CP/M sends a CARRIAGE RETURN/LINE FEED command to the console, but does not start to execute the command line you have typed thus far. `CONTROL-E` is, in effect, a *physical* end-of-line, not a *logical* one.

As you can see, you will need to use this command only if your terminal either overprints (if it is a hard copy device) or does not wrap around when the cursor gets to the right-hand end of the line.

**CONTROL-H: Backspace**  
The `CONTROL-H` command is the ASCII backspace character. When you type it, the CCP will "destructively" backspace the cursor. Use it to correct typing errors you discover before you finish entering the command line. The last character you typed will disappear from the screen. The CCP does this by sending a three-character sequence of backspace, space, backspace to the console.
The CCP ignores attempts to backspace over its own prompt. It also takes care of backspacing over control characters that take two character positions on the line. The CCP sends the character sequence backspace, backspace, space, space, backspace, backspace, erasing both characters.

**CONTROL-J: Line Feed/CONTROL-M: Carriage Return**  The CONTROL-J command is the ASCII LINE FEED character; CONTROL-M is the CARRIAGE RETURN. Both of these characters terminate the command line. The CCP will then execute the command.

**CONTROL-P: Printer Echo**  The CONTROL-P command is used to turn on and off a feature called *printer echo*. When it is turned on, every character sent to the console is also sent to CP/M's list device. You can use this command to get a hard copy of information that normally goes only to the console.

CONTROL-P is a “toggle.” The first time you type CONTROL-P it turns on printer echo; the next time you type CONTROL-P it turns off printer echo. Whenever CP/M does a warm boot, printer echo is turned off.

There is no easy way to know whether printer echo is on or off. Try typing a few CARRIAGE RETURNS, and see whether the printer responds; if it does not, type CONTROL-P and try again.

One of the shortcomings in most CP/M implementations is that the printer drivers (the software in the BIOS that controls or “drives” the printer) do not behave very intelligently if the printer is switched off or not ready when you or your program asks it to print. Under these circumstances, the software will wait forever and the system will appear to be dead. So if you “hang” the system in this way when you type a CONTROL-P, check that the printer is turned on and ready. Otherwise, you may have to reset the entire system.

**CONTROL-R: Repeat Command Line**  The CONTROL-R command makes the CCP repeat or retype the current input line. The CCP outputs a “#” character, a CARRIAGE RETURN/LINE FEED, and then the entire contents of the command line buffer. This is a useful feature if you are working on a teletype or other hard copy terminal and have used the RUB or DEL characters. Since these characters do not destructively delete a character, you can get a visually confusing line of text on the terminal. The CONTROL-R character gives you a fresh copy of the line without any of the logically deleted characters cluttering it up. In this way you can see exactly what you have typed into the command line buffer.

See the discussion of the RUB and DEL characters for an example of CONTROL-R in use.

**CONTROL-S: Stop Screen Output**  The CONTROL-S command is the ASCII XOFF (also called DC3) character; XOFF is an abbreviation for “Transmit Off.” Typing CONTROL-S will temporarily stop output to the console. In a standard version of
CP/M, the CCP will resume output when *any* character is entered (including another CONTROL-S) from the console. Thus, you can use CONTROL-S as a toggle switch to turn console output on and off.

In some implementations of CP/M, the console driver itself (the low-level code in the BIOS that controls the console) will be maintaining a communication protocol with the console; therefore, a better way of resuming console output after pausing with a CONTROL-S is to use CONTROL-Q, the ASCII XON or "Transmit On" character. Entering a CONTROL-Q instead of relying on the fact that *any* character may be used to continue the output is a fail-safe measure.

The commands CONTROL-S and CONTROL-Q are most useful when you have large amounts of data on the screen. By “riding” the CONTROL-S and CONTROL-Q keys, you can let the data come to the screen in small bursts that you can easily scan.

**CONTROL-U or CONTROL-X: Undo Command Line** The commands CONTROL-U and CONTROL-X perform the same function: They erase the current partially entered command line so that you can undo any mistakes and start over. The CONTROL-U command was originally intended for hard copy terminals. The CCP outputs a “#” character, then a CARRIAGE RETURN/LINE FEED, and then some blanks to leave the cursor lined up and ready for you to enter the next command line. It leaves what you originally entered in the previous line on the screen. The CONTROL-X command is more suited to screens; the CCP destructively backspaces to the beginning of the command line so that you can reenter it.

**RUB or DEL: Delete Last Character** The rub out or delete function (keys marked RUB, RUBOUT, DEL, or DELETE) nondestructively deletes the last character that you typed. That is, it deletes the last character from the command line buffer and echoes it back to the console.

Here is an example of a command line with the last few characters deleted using the RUB key:

```
A>RUN PAYROLLLLLORYAPSALES
       ^^^^^^^
DELeted
```

You can see that the command line very quickly becomes unreadable. If you lose track of what are data characters and what has been deleted, you can use CONTROL-R to get a fresh copy of what is in the command line buffer.

The example above would then appear as follows:

```
A>RUN PAYROLLLLLORYAPSALES#
RUN SALES_
```

The “#” character is output by the CCP to indicate that the line has been
The CP/M Programmer's Handbook

The "_" represents the position of the cursor, which is now ready to continue with the command line.

**Built-In Commands**

When you enter a command line and press either CARRIAGE RETURN or LINE FEED, the CCP will check if the command name is one of the set of built-in commands. (It has a small table of command names embedded in it, against which the entered command name is checked.) If the command name matches a built-in one, the CCP executes the command immediately.

The next few sections describe the built-in commands that are available; however, refer to *Osborne CP/M User Guide*, second edition by Thom Hogan (Berkeley: Osborne/McGraw-Hill, 1982) for a more comprehensive discussion with examples of the various forms of each command.

**X: — Changing Default Disk Drives** The default drive is the currently active drive that CP/M uses for all file access whenever you do not nominate a specific drive. If you wish to change the default drive, simply enter the new default drive's identifying letter followed by a colon. The CCP responds by changing the name of the disk that appears in the prompt line.

On hard disks, this simple operation may take a second or two to complete because the BDOS, requested by the CCP to log in the drive, must read through the disk directory and rebuild the allocation vector for the disk. If you have a diskette or a disk that is removable, changing it and performing a warm boot has the same effect of refreshing CP/M's image of which allocation blocks are used and which are available. It takes longer on a hard disk because, as a rule, the directories are much larger.

**DIR — Directory of Files** In its simplest form, the DIR command displays a listing of the files set to Directory status in the current user number (or file group) on the current default drive. Therefore, when you do not ask for any files after the DIR command, a file name of "*.*" is assumed. This is a total wildcard, so all files that have not been given System status will be displayed. This is the only built-in command where an omitted file name reference expands to "all file names, all file types."

You can display the directory of a different drive by specifying the drive in the same command line as the DIR command.

You can qualify the files you want displayed by entering a unique or ambiguous file name or extension. Only those files that match the given file name specification will be displayed, and even then, only those files that are not set to System status will appear on the screen. (The standard CP/M utility program STAT can be used to change files from SYS to DIR status.)
Another side effect of the DIR command and files that are SYS status is best illustrated by an example. Imagine that the current logical drive B has two files on it called SYSFILE (which has SYS status) and NONSYS (which does not). Look at the following console dialog, in which user input is underlined:

```
B>DIR<cr>
B: NONSYS
B>DIR JUNK<cr>
NO FILE
B>DIR SYSFILE<cr>
B>
```

Do you see the problem? If a file is not on the disk, the CCP will display NO FILE (or NOT FOUND in earlier versions of CP/M). However, if the file does exist but is a SYS file, the CCP does not display it because of its status; nor does the CCP say NO FILE. Instead it quietly returns to the prompt. This can be confusing if you are searching for a file that happens to be set to SYS status. The only safe way to find out if the file does exist is to use the STAT utility.

**ERA — Erase a File**

The ERA command logically removes files from the disk (*logically* because only the file directory is affected; the actual data blocks are not changed).

The logical delete changes the first byte of each directory entry belonging to a file to a value of 0E5H. As you may recall from the discussion on the file directory entry in Chapter 3, this first byte usually contains the file user number. If it is set to 0E5H, it marks the entry as being deleted.

ERA makes a complete pass down the file directory to logically delete all of the extents of the file.

Unlike DIR, the ERA command does not assume “all files, all types” if you omit a file name. If it did, it would be all too easy to erase all of your files by accident. You must enter “*.#” to erase all files, and even then, you must reassure the CCP that you really want to erase all of them from the disk. The actual dialog looks like the following:

```
A>era b:*.<cr>
ALL (Y/N)?y<cr>
A>
```

If you change your mind at the last minute, you can press “n” and the CCP will not erase any files.

One flaw in CP/M is that the ERA command only asks for confirmation when you attempt to erase all of your files using a name such as “*.#” or “*.??”. Consider the impact of the following command:

```
A>ERA *.C<cr>
A>
```

The CCP with no hesitation has wiped out all files that have a file type starting with the letter “C” in the current user number on logical disk A.
If you need to use an ambiguous file name in an ERA command, check which files you will delete by first using a STAT command with exactly the same ambiguous file name. STAT will show you all the files that match the ambiguous name, even those with SYS status that would not be displayed by a DIR command.

There are several utility programs on the market with names like UNERA or WHOOPS, which take an ambiguous file name and reinstate the files that you may have accidentally erased. A design for a version of UNERASE is discussed in Chapter 11.

If you attempt to erase a file that is not on the specified drive, the CCP will respond with a NO FILE message.

**REN — Rename a File**  
The REN command renames a file, changing the file name, the file type, or both. In order to rename, you need to enter two file names, the new name and the current file name.

To remember the correct name format, think of the phrase *new = old*. The actual command syntax is

```
A> ren newfile,typ=oldfile,typ<cr>
A> _
```

You can use a logical disk drive letter to specify on which drive the file exists. If you specify the drive, you only need to enter it on one of the file names. If you enter the drive with both file names, it must be the same letter for both.

Unlike the previous built-in command, REN cannot be used with ambiguous file names. If you try, the CCP echoes back the ambiguous names and a question mark, as in the following dialog:

```
A> ren chap*.doc=chapter*.doc<cr>
CHAP*.DOC=CHAPTER*.DOC?
A> _
```

If the REN command cannot find the old file, it will respond NO FILE. If the new file already exists, the message FILE EXISTS will be displayed. If you receive a FILE EXISTS message and want to check that the new file does exist, remember that it is better to use the STAT command than DIR. The extant file may be declared to be SYS status and therefore will not appear if you use the DIR command.

**TYPE — Type a Text File**  
The TYPE command copies the specified file to the console. You cannot use ambiguous file names, and you will need to press CONTROL-S if the file has more data than can fill one screen. With the TYPE command, the data in the file will fly past on the screen unless you stop the display by pressing CONTROL-S. Be careful, because if you type any other character, the TYPE command will abort and return control to the CCP.
Once you have had time to see what is displayed on the screen, you can press CONTROL-Q to resume the output of data to the console. With standard CP/M implementations, you will discover that any character can be used to restart the flow of data; however, use CONTROL-Q as a fail-safe measure. CONTROL-S (X-OFF) and CONTROL-Q (X-ON) conform to the standard protocol which should be used.

If you need to get hard copy output of the contents of the file, you should type a CONTROL-P command before you press the CARRIAGE RETURN at the end of the TYPE command line.

As you may have inferred, the TYPE command should only be used to output ASCII text files. If for some reason you use the TYPE command with a file that contains binary information, strange characters will appear on the screen. In fact, you may program your terminal into some state that can only be remedied by turning the power off and then on again. The general rule therefore is only use the TYPE command with ASCII text files.

**SAVE — Save Memory Image on Disk** The SAVE command is the hardest of the CCP's commands to explain. It is more useful to the programmer than to a typical end user. The format of this command is

```
A>SAVE n FILENAME,TYP<cr>
A>
```

The SAVE command creates a file of the specified name and type (or overwrites an existing file of this name and type), and writes into it the specified number \( n \) of memory pages. A page in CP/M is 256 (100H) bytes. The SAVE command starts writing out memory from location 100H, the start of the Transient Program Area (TPA). Before you use this command, you will normally have loaded a program into the TPA. The SAVE command does just what its name implies: It saves an image of the program onto a disk file.

More often than not, when you use the SAVE command the file type will be “.COM.” With the file saved in this way, the CCP will be able to load and execute the file.

**USER — Change User Numbers** As mentioned before, the directory of each logical disk consists of several directories that are physically interwoven but logically separated by the user number. When you use a specific user number, those files that were created when you were in another user number are logically not available to you.

The USER command provides a way for you to move from one user number to another. The command format is

```
A>USER n<cr>
A>
```

where \( n \) can be any number from 0 to 15. Any other number will provoke the CCP to echoing back your entry, followed by a question mark.
But once you have switched back and forth between user numbers several times, it is easy to become confused about which user number you are in. The STAT command can be used to find the current user number. If you are in a user number that does not make a copy of STAT available to you however, all you can do is use the USER command to set yourself to another user number. You cannot find out which user number you were in; you can only tell the system the user number you want to go to.

In the custom BIOS systems discussed later, there is a way of displaying the current user number each time a warm boot occurs. If you are building a system in which you plan to utilize CP/M's user number features, you should give this display of the current user number serious thought. If you are in the wrong user number and erase files, you can create serious problems.

Some implementations of CP/M have modified the CCP so that the prompt shows the current user number as well as the default drive (similar to the prompt used in MP/M). However, this use of a nonstandard CCP is not a good practice. As a rule, customization should be confined to the BIOS.

**Program Loading**

The first area to consider when loading a program is the first 100H bytes of memory, called the base page. Several fields — units in this area of memory — are set to predetermined values before a program takes control.

To aid in this discussion, imagine a program called COPYFILE that copies one file to another. This program expects you to specify the source and destination file names on the command line. A typical command would read

```
A>copyfile tofile.typ fromfile.typ display
```

Notice the word “display.” COPYFILE will, if you specify the “display” option, output the contents of the source file (“fromfile.typ”) on the console as the transfer takes place.

When you press the CARRIAGE RETURN key at the end of the command line, the CCP will search the current default drive (“A” in the example) and load a file called COPYFILE.COM into memory starting at location 100H. The CCP then transfers control to location 100H — just past the base page — and COPYFILE starts executing.

**Base Page**

The base page normally starts from location 0000H in memory, but where there is other material in low memory addresses, it may start at a higher address. Figure 4-1 shows the assembly language code you will need to access the base page. RAM is assumed to start at location 0000H in this example.
Some versions of CP/M, such as the early Heathkit/Zenith system, have ROM from location 0000H to 42FFH. Digital Research, responding to market pressure, produced a version of CP/M that assumed RAM starting at 4300H. If you have one of these systems, you must add 4300H to all addresses in the following paragraphs except for those that refer to addresses at the top of memory. These will not be affected by the presence of ROM in low memory.

The individual values used in fields in the base page are described in the following sections.

**Warmboot** The three-byte *warmboot* field contains an instruction to jump up to the high end of RAM. This JMP instruction transfers control into the BIOS and triggers a warm boot operation. As mentioned before, a warm boot causes CP/M to reload the CCP and rebuild the allocation vector for the current default disk. If you need

---

### Figure 4-1. Base page data declarations

<table>
<thead>
<tr>
<th>Address</th>
<th>Symbol</th>
<th>Type</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>RAM</td>
<td>EQU</td>
<td>0</td>
<td>Start of RAM (and the base page)</td>
</tr>
<tr>
<td>0000</td>
<td>ORG</td>
<td>RAM</td>
<td>3</td>
<td>Set location counter to RAM base</td>
</tr>
<tr>
<td>0000</td>
<td>WARMBOOT;</td>
<td>DS</td>
<td>3</td>
<td>Contains a JMP to warm boot entry</td>
</tr>
<tr>
<td>0002</td>
<td>BIOSPAGE;</td>
<td>EDU</td>
<td>RAM+2</td>
<td>BIOS Jump vector page</td>
</tr>
<tr>
<td>0003</td>
<td>IBYTE;</td>
<td>DS</td>
<td>1</td>
<td>Input/output redirection byte</td>
</tr>
<tr>
<td>0004</td>
<td>CURUSER;</td>
<td>DS</td>
<td>1</td>
<td>Current user (bits 7-4)</td>
</tr>
<tr>
<td>0004</td>
<td>CURDISK</td>
<td>EQU</td>
<td>CURUSER</td>
<td>Default logical disk (bits 3-0)</td>
</tr>
<tr>
<td>0005</td>
<td>BDOSE;</td>
<td>DS</td>
<td>3</td>
<td>Contains a JMP to BDOS entry</td>
</tr>
<tr>
<td>0007</td>
<td>TOPRAM</td>
<td>EQU</td>
<td>BDOSE+2</td>
<td>Top page of usable RAM</td>
</tr>
<tr>
<td>005C</td>
<td>FCB1;</td>
<td>DS</td>
<td>16</td>
<td>File control block #1</td>
</tr>
<tr>
<td>006C</td>
<td>FCB2;</td>
<td>DS</td>
<td>16</td>
<td>File control block #2</td>
</tr>
<tr>
<td>0080</td>
<td>ORG</td>
<td>RAM+80H</td>
<td>Bypass unused locations</td>
<td></td>
</tr>
<tr>
<td>0080</td>
<td>COMTAIL;</td>
<td>COMTAIL;COUNT:</td>
<td>DS</td>
<td>Complete command tail</td>
</tr>
<tr>
<td>0081</td>
<td>COMTAIL;CHARS:</td>
<td>DS</td>
<td>127</td>
<td>Count of the number of chars in command tail (CR not incl.)</td>
</tr>
<tr>
<td>0080</td>
<td>ORG</td>
<td>RAM+80H</td>
<td>Redefine command tail area</td>
<td></td>
</tr>
<tr>
<td>0080</td>
<td>DMABUFFER;</td>
<td>DS</td>
<td>128</td>
<td>Default &quot;DMA&quot; address used as a 128-byte record buffer</td>
</tr>
<tr>
<td>0100</td>
<td>TPA;</td>
<td>ORG</td>
<td>RAM+100H</td>
<td>Bypass unused locations</td>
</tr>
</tbody>
</table>

Start of transient program area into which programs are loaded.
The CP/M Programmer's Handbook

To cause a warm boot from within one of your assembly language programs, code

```
JMP 0 ;Warm Boot
```

**BIOSPAGE** The BIOS has several different entry points; however, they are all clustered together at the beginning of the BIOS. The first few instructions of the BIOS look like the following:

```
JMP ENTRY1
JMP ENTRY2
JMP ENTRY3 ; and so on
```

Because of the way CP/M is put together, the first jump instruction *always* starts on a page boundary. Remember that a page is 256 (100H) bytes of memory, so a page boundary is an address where the least significant eight bits are zero. For example, the BIOS jump vector (as this set of JMPs is called) may start at an address such as F200H or E600H. The exact address is determined by the size of the BIOS.

By looking at the BIOSPAGE, the most significant byte of the address in the warmboot JMP instruction, the page address of the BIOS jump vector can be determined.

**IOBYTE** CP/M is based on a philosophy of separating the physical world from CP/M's own logical view of the world. This philosophy also applies to the character-oriented devices that CP/M supports.

The IOBYTE consists of four two-bit fields that can be used to assign a physical device to each of the logical ones. It is important to understand that the IOBYTE itself is just a passive data structure. Actual assignment occurs only when the physical device drivers examine the IOBYTE, interpreting its contents and selecting the correct physical drive for the cooperation of the BIOS. These device drivers are the low-level (that is, close to machine language) code in the BIOS that actually interfaces and controls the physical device.

The four logical devices that CP/M knows about are

1. *The console.* This is the device through which you communicate with CP/M. It is normally a terminal with a screen and a keyboard. The console is a bidirectional device: It can be used as a source for information (input) and a destination to which you can send information (output).

   In CP/M terminology, the console is known by the symbolic name of "CON:". Note the ":"—this differentiates the device name from a disk file that might be called "CON."

2. *The list device.* This is normally a printer of some sort and is used to make hard copy listings. CP/M views the printer as an output device only. This creates problems for printers that need to tell CP/M they are busy, but this
problem can be remedied by adding code to the low-level printer driver. 
CP/M's name for this logical device is "LST:"

3. The paper tape reader. It is unusual to find a paper tape reader in use today. Originally, CP/M ran on an Intel Microcomputer Development System called the MDS-800, and this system had a paper tape reader. This device can be used only as a source for information.
CP/M calls this logical device "RDR:"

4. The paper tape punch. This, too, is a relic from CP/M's early days and the MDS-800. In this case, the punch can be used only for output.
The logical device name used by CP/M is "PUN:"

The physical arrangement of the IOBYTE fields is shown in Figure 4-2.
Each two-bit field can take on one of four values: 00, 01, 10, and 11. The particular value can be interpreted by the BIOS to mean a specific physical device, as shown in Table 4-1.
Although the actual interpretation of the IOBYTE is performed by the BIOS, the STAT utility can set the IOBYTE using the logical and physical device names, and PIP (Peripheral Interchange Program) can be used to copy data from one device to another. In addition, you can write a program that simply changes the

<table>
<thead>
<tr>
<th>Bit Number</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Logical Device</td>
<td>List</td>
<td>Punch</td>
<td>Reader</td>
<td>Console</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Figure 4-2.** Arrangement of the IOBYTE

**Table 4-1.** IOBYTE Values

<table>
<thead>
<tr>
<th>Logical Device</th>
<th>Physical Device</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>00</td>
</tr>
<tr>
<td>Console (CON:)</td>
<td>TTY:</td>
</tr>
<tr>
<td>Reader (RDR:)</td>
<td>TTY:</td>
</tr>
<tr>
<td>Punch (PUN:)</td>
<td>TTY:</td>
</tr>
<tr>
<td>List (LST:)</td>
<td>TTY:</td>
</tr>
</tbody>
</table>
The CP/M Programmer's Handbook

contents of the IOBYTE. But be careful: Changes in the IOBYTE take effect immediately.

The values in the IOBYTE have the following meanings:

**Console (CON:)**

00 Teletype driver (TTY:)
This driver is assumed to be connected to a hard copy device being used as the main console.

01 CRT driver (CRT:)
The driver is assumed to be connected to a CRT terminal.

10 Batch mode (BAT:)
This is a rather special case. It is assumed that appropriate drivers will be called so that console input comes from the logical reader (RDR:) and console output is sent to the logical list device (LST:).

11 User defined console (UC1:)
Meaning depends on the individual BIOS implementation. If, for example, you have a high-resolution graphics screen, you could arrange for this setting of the IOBYTE to direct console output to it. You might make console input come in from some graphic tablet, joystick, or other device.

**Reader (RDR:)**

00 Teletype driver (TTY:)
This refers to the paper tape reader device that was often found on teletype consoles.

01 Paper tape reader (PTR:)
This presumes some kind of high-speed input device connected to the system. Modern systems rarely have such a device, so this setting is often used to connect the logical reader to the input side of a communications line.

10 User defined reader #1 (UR1:)

11 User defined reader #2 (UR2:)
Both of these settings can be used to direct the physical driver to some other specialized devices. These values are included only because they would otherwise have been unassigned. They are rarely used.

**Punch (PUN:)**

00 Teletype driver (TTY:)
This refers to the paper tape punch that was often found on teletype consoles.

01 Paper tape punch (PTP:)
This presumes that there is some kind of high-speed paper tape punch connected to the system. Again, this is rarely the case, so this setting is often used to connect the logical punch to the output side of a communications line.

10  User defined punch #1 (UP1:)
11  User defined punch #2 (UP2:)
These two settings correspond to the two user defined readers, but they are practically never used.

List (LST:)

00  Teletype driver (TTY:)
Output will be printed on a teletype.

01  CRT driver (CRT:)
Output will be directed to the screen on a CRT terminal.

10  Line printer driver (LPT:)
Output will go to a high-speed printing device. Although the name line printer implies a specific type of hardware, it can be any kind of printer.

11  User defined list device (UL1:)
Whoever writes the BIOS can arrange for this setting to cause logical list device output to go to a device other than the main printer.

To repeat: The IOBYTE is not actually used by the main body of CP/M. It is just a passive data structure that can be manipulated by the STAT utility. Whether the IOBYTE has any effect depends entirely on the particular BIOS implementation.

CURUSER
The CURUSER field is the most significant four bits (high order nibble) of its byte. It contains the currently selected user number set by the CCP USER command, by a specific call to the BDOS, or by a program setting this nibble to the required value. This last way of changing user numbers may cause compatibility problems with future versions of CP/M, so use it only under controlled conditions.

CURDISK
The CURDISK field is the least significant four bits of the byte it shares with CURUSER. It contains a value of 0 if the current disk is A:, 1 if it is B:, and so on.
The CURDISK field can be set from the CCP, by a request to the BDOS, or by a program altering this field. The caveat given for CURUSER regarding compatibility also applies here.

BDOSE
This three-byte field contains an instruction to jump to the entry point of the BDOS. Whenever you want the BDOS to do something, you can transfer the request to the BDOS by placing the appropriate values in registers and making a CALL to this JMP instruction. By using a CALL, the return address will be
placed on the stack. The subsequent JMP to the BDOS does not put any additional information onto the stack, which operates on a last-in, first-out basis; so when the system returns from the BDOS, it will return directly to your program.

**TOPRAM** Because the BDOS, like the BIOS, starts on a page boundary, the most significant byte of the address of the BDOS entry tells you in which page the BDOS starts. You must subtract 1 from the value in TOPRAM to get the highest page number that you can use in your program. Note that when you use this technique, you assume that the CCP will be overwritten since it resides in memory just below the BDOS.

**FCB1 and FCB2** As a convenience, the CCP takes the first two parameters that appear in the command tail (see next section), attempts to parse them as though they were file names, and places the results in FCB1 and FCB2. The results, in this context, mean that the logical disk letter is converted to its FCB representation, and the file name and type, converted to uppercase, are placed in the FCB in the correct bytes. In addition, any use of "*" in the file name is expanded to one or more question marks. For example, a file name of "abc*." will be converted to a name of "ABC?????" and type of "???".

Notice that FCB2 starts only 16 bytes above FCB1, yet a normal FCB is at least 33 bytes long (36 bytes if you want to use random access). In many cases, programs only require a single file name. Therefore, you can proceed to use FCB1 straight away, not caring that FCB2 will be overwritten.

In the case of the COPYFILE program example on previous pages, two file names are required. Before FCB1 can be used, the 16 bytes of FCB2 must be moved into a skeleton FCB that is declared in the body of COPYFILE itself.

**COMTAIL** The command tail is everything on the command line other than the command name itself. For example, the command tail in the COPYFILE command line is shown here:

```
A>copyfile tofile.type fromfile.type display
```

The CCP takes the command tail (converted to uppercase) and stores it in the COMTAIL area.

**COMTAIL$COUNT** This is a single-byte binary count of the number of characters in the command tail. The count does not include a trailing CARRIAGE RETURN or a blank between the command name and the command tail. For example, if you enter the command line

```
A>PRINT ABC*. *
```
the COMTAIL$COUNT will be six, which is the number of characters in the string "ABC*.*".

**COMTAIL$CHARS** These are the actual characters in the command tail. This field is not blank-filled, so you must use the COMTAIL$COUNT in order to detect the end of the command tail.

**DMA$BUFFER** In Figure 4-1, the DMA$BUFFER is actually the same area of memory as the COMTAIL. This is a space-saving trick that works because most programs process the contents of the command tail before they do any disk input or output.

The DMA$BUFFER is a sector buffer (hence it has a length of 128 bytes). The use of the acronym DMA (direct memory access) refers back to the Intel MDS-800. This system had hardware that could move data to and from diskettes by going directly to memory, bypassing the CPU completely. The term is still used even though you may have a computer system that does not use DMA for its disk I/O. You can substitute the idea of "the address to/from which data is read/written" in place of the DMA concept.

You can request CP/M to use a DMA address other than DMA$BUFFER, but whenever the CCP is in control, the DMA address will be set back here.

**TPA**

This is the transient program area into which the CCP loads programs. The TPA extends up to the base of the BDOS.

The TPA is also the starting address for the memory image that is saved on disk whenever you use the CCP SAVE command.

**Memory Dumps of the Base Page**

The following are printouts showing the contents of the base page (the first 100H bytes of memory) as the COPYFILE program will see it.

This is an example of the first 16 bytes of memory:

```
00001 C3 03 F2 95 00 C3 00 C2 FF F6 F5 FF F3 F2 FF F0
```

- Arbitrary data left from system startup
- JMP to BDOS Entry Point
  (Note 0C200H is starting page of BDOS)
- Current default disk (0 = A, 1 = B)
- Current User (User = 0)
- Settings of the IOBYTE
- JMP WARMBOOT
  (Note that the BIOS Jump Vector is at 0F200H)
The command line, as you recall, was

```
A>copyfile tofile.typ fromfile.typ display
```

The FCBI and FCB2 areas will be set by the CCP as follows:

```
Logical Disk   Logical Disk

0060: 49 4C 4E 53 49 4F 4E 00 00 00 00 00 46 52 4F 4E 00 60
           FILE     TVP . FROM

0070: 4D 46 49 4C 45 54 59 50 00 00 00 00 00 F2 34 F3
           FILETVP 4.

Since the logical disks were not specified in the file names in the command line, the CCP has set the disk code in both FCBI and FCB2 to 00H, meaning "use the default disk." The file name and type have been converted to uppercase, separated, and put into the FCBs in their appointed places.

The complete command tail has been stored in COMTAIL as follows:

```
31 in decimal

0080: 1F 54 4F 46 49 4C 45 2E 54 59 50 20 46 52 4F 4E
           FILE.TVP DISPLAV

0090: 00 43 4C 4F 4E 53 49 4F 4E 00 00 00 00 00 00 00 00 00 00
           C R C K . . . .

00A0: 65 65 65 65 65 65 65 65 65 65 65 65 65 65 65 65
           E E E E E E E E E E E E E E E E E E

00B0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

00C0: 65 65 65 65 65 65 65 65 65 65 65 65 65 65 65 65
           E E E E E E E E E E E E E E E E E E

00D0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

00E0: 65 65 65 65 65 65 65 65 65 65 65 65 65 65 65 65
           E E E E E E E E E E E E E E E E E E

00F0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

0100: 01 F9

Program Start
```

You can see that the command tail length is 01 FH (31 decimal). This is followed immediately by the command tail characters themselves. Note that the command tail stops at location 9FH. The remainder of the data that you can see is the residue of some previous directory operation by the CCP. You can see the file name CRCK.COM in a directory entry, followed by several 0E5Hs that are unused directory space.

Finally, at location 0100H are the first two bytes of the program.
Processing the Command Tail

One of the first problems facing you if you write a program that can accept parameters from the command tail is to process the command tail itself, isolating each of the parameters. You should use a standard subroutine to do this. This subroutine splits the command line into individual parameters and returns a count of the number of parameters, as well as a pointer to a table of addresses. Each address in this table points in turn to a null-byte-terminated string. Each parameter is placed in a separate string.

Figure 4-3 contains the listing of this subroutine, CTP (Command Tail Processor).

```
0100 ORG 100H
0100 CD3601 CALL CTP ; Test bed for CTP
0103 00
10 This subroutine breaks the command tail apart, placing
20 each value in a separate string area.
30 Return parameters:
40 A = 0 - No error (I flag set)
50 B = Count of number of parameters
60 HL -> Table of addresses
70 Each address points to a null-byte-terminated parameter string.
80 If too many parameters are specified, then A = TMP
90 If a given parameter is too long, then A = PTL
100 and 0 points to the first character of the
110 offending parameter in the COMTAIL area.

0080 = COMTAIL EQU SOH ; Command tail in base page
0080 = COMTAIL$COUNT EQU COMTAIL ; Count of chars. in command tail
0001 = CTP$TMP EQU 1 ; Too many parameters error code
0002 = CTP$PTL EQU 2 ; Parameter too long error code

PTABLE: ; Table of pointers to parameters
0104 OCOI DW P1 ; Parameter 1
0106 1A01 DW P2 ; Parameter 2
0108 2801 DW P3 ; Parameter 3
010A 0000 DW 0 ; Terminator

Parameter strings.
The first byte is 0 so that unused parameters appear to be null strings.
The last byte of each is a 0 and is used to detect a parameter that is too long.

010C 0001010101PI DB 0,1,1,1,1,1,1,1,1,1,1,1,1,1,0 ; Param. 1 & terminator
011A 0001010101P2 DB 0,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0 ; Param. 2 & terminator
0128 0001010101P3 DB 0,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0 ; Param. 3 & terminator

; Add more parameter strings here

; Main entry point <<<<
0136 210401 LXI H,PTABLE ; HL -> table of addresses
0139 0600 MVI C,0 ; Set parameter count
0138 3A8000 LDA COMTAIL$COUNT ; Character count
013E 87ORA A ; Check if any params.
013F 82 RZ ; Exit (return params, already set)
0140 E5 PUSH H ; Save on top of stack for later
0141 47 MOV B,A ; B = COMTAIL char. count
0142 218100 LXI H,COMTAIL+1 ; HL -> Command tail chars.
```

Figure 4-3. Command Tail Processor (CTP)
Figure 4-3. Command Tail Processor (CTP) (continued)

Available Memory

Many programs need to use all of available memory, and so very early in the program they need to set the stack pointer to the top end of the available RAM. As mentioned before, the CCP can be overwritten as it will be reloaded on the next warm boot.
Figure 4-4 shows the code used to set the stack pointer. This code determines the amount of memory in the TPA and sets the stack pointer to the top of available RAM.

**Communicating with the BIOS**

If you are writing a utility program to interact with a customized BIOS, there will be occasions where you need to make a direct BIOS call. However, if your program ends up on a system running Digital Research's MP/M Operating System, you will have serious problems if you try to call the BIOS directly. Among other things, you will crash the operating system.

If you need to make such a call and you are aware of the dangers of using direct BIOS calls, Figure 4-5 shows you one way to do it.

Remember that the first instructions in the BIOS are the jump vector—a sequence of JMP instructions one after the other. Before you can make a direct call, you need to know the relative page offset of the particular JMP instruction you want to go to. The BIOS jump vector always starts on a page boundary, so all you need to know is the least significant byte of its address.
Returning to CP/M

Once your program has run, you will need to return control back to CP/M. If your program has not overwritten the CCP and has left the stack pointer as it was when your program was entered, you can return directly to the CCP using a RET instruction.

Figure 4-6 shows how a normal program would do this if you use a local stack, one within the program. The CCP stack is too small; it has room for only 24 16-bit values.

The advantage of returning directly to the CCP is speed. This is true especially on a hard disk system, where the time needed to perform a warm boot is quite noticeable.

If your program has overwritten the CCP, you have no option but to transfer control to location 0000H and let the warm boot occur. To do this, all you need do is execute

```
EXIT: JMP 0 ;Warm Boot
```

(As a hint, if you are testing a program and it suddenly exits back to CP/M, the odds are that it has inadvertently blundered to location 0000H and executed a warm boot.)
The Basic Disk Operating System is the real heart of CP/M. Unlike the Console Command Processor, it must be in memory all the time. It provides all of the input/output services to CP/M programs, including the CCP.

As a general rule, unless you are writing a system-dependent utility program, you should use the BDOS for all of your program's input/output. If you circumvent the BDOS you will probably create problems for yourself later.
The BDOS does all of the system input/output for you. These services can be grouped into two types of functions:

**Simple Byte-by-Byte I/O**
This is sending and receiving data between the computer system and its logical devices—the console, the "reader" and "punch" (or their substitutes), and the printer.

**Disk File I/O**
This covers such tasks as creating new files, deleting old files, opening existing files, and reading and writing 128-byte long "records" to and from these files.

The remainder of this chapter explains each of the BDOS functions, shows how to make each operating system request, and gives additional information for each function. You should also refer to Digital Research's manual, *CP/M 2 Interface Guide*, for their standard description of these functions.

**BDOS Function Calls**

The BDOS function calls are described in the order of their function code numbers. Figure 5-1 summarizes these calls.

**Naming Conventions**

In practice, whenever you write programs that make BDOS calls, you should include a series of equates for the BDOS function code numbers. We shall be making reference to these values in subsequent examples, so they are shown in Figure 5-2 as they will appear in the programs.

The function names used to define the equates in Figure 5-2 are shorter than those in Figure 5-1 to strike a balance between the abbreviated function names used in Digital Research's documentation and the need for clearer function descriptions.

**Making a BDOS Function Request**

All BDOS functions are requested by issuing a CALL instruction to location 0005H. You can also request a function by transferring control to location 0005H with the return address on the stack.

In order to tell the BDOS what you need it to do, you must arrange for the internal registers of the CPU to contain the required information before the CALL instruction is executed.
<table>
<thead>
<tr>
<th>Function Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Simple Byte-by-Byte I/O</strong></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>Overall system and BDOS reset</td>
</tr>
<tr>
<td>1</td>
<td>Read a byte from the console keyboard</td>
</tr>
<tr>
<td>2</td>
<td>Write a byte to the console screen</td>
</tr>
<tr>
<td>3</td>
<td>Read a byte from the logical reader device</td>
</tr>
<tr>
<td>4</td>
<td>Write a byte to the logical punch device</td>
</tr>
<tr>
<td>5</td>
<td>Write a byte to the logical list device</td>
</tr>
<tr>
<td>6</td>
<td>Direct console I/O (no CCP-style editing)</td>
</tr>
<tr>
<td>7*</td>
<td>Read the current setting of the IOBYTE</td>
</tr>
<tr>
<td>8*</td>
<td>Set a new value of the IOBYTE</td>
</tr>
<tr>
<td>9</td>
<td>Send a &quot;$&quot;-terminated string to the console</td>
</tr>
<tr>
<td>10</td>
<td>Read a string from the console into a buffer</td>
</tr>
<tr>
<td>11</td>
<td>Check if a console key is waiting to be read</td>
</tr>
<tr>
<td>12</td>
<td>Return the CP/M version number</td>
</tr>
<tr>
<td><strong>Disk File I/O</strong></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>Reset disk system</td>
</tr>
<tr>
<td>14</td>
<td>Select specified logical disk drive</td>
</tr>
<tr>
<td>15</td>
<td>Open specified file for reading/writing</td>
</tr>
<tr>
<td>16</td>
<td>Close specified file after reading/writing</td>
</tr>
<tr>
<td>17</td>
<td>Search file directory for first match with filename</td>
</tr>
<tr>
<td>18</td>
<td>Search file directory for next match with filename</td>
</tr>
<tr>
<td>19</td>
<td>Delete (erase) file</td>
</tr>
<tr>
<td>20</td>
<td>Read the next &quot;record&quot; sequentially</td>
</tr>
<tr>
<td>21</td>
<td>Write the next &quot;record&quot; sequentially</td>
</tr>
<tr>
<td>22</td>
<td>Create a new file with the specified name</td>
</tr>
<tr>
<td>23</td>
<td>Rename a file to a new name</td>
</tr>
<tr>
<td>24</td>
<td>Indicate which logical disks are active</td>
</tr>
<tr>
<td>25</td>
<td>Return the current default disk drive number</td>
</tr>
<tr>
<td>26</td>
<td>Set the DMA address (read/write address)</td>
</tr>
<tr>
<td>27</td>
<td>Return the address of an allocation vector</td>
</tr>
<tr>
<td>28*</td>
<td>Set specified logical disk drive to Read-Only status</td>
</tr>
<tr>
<td>29</td>
<td>Indicate which disks are currently Read-Only status</td>
</tr>
<tr>
<td>30</td>
<td>Set specified file to System or Read-Only status</td>
</tr>
<tr>
<td>31</td>
<td>Return address of disk parameter block (DPB)</td>
</tr>
<tr>
<td>32*</td>
<td>Set/Get the current user number</td>
</tr>
<tr>
<td>33</td>
<td>Read a &quot;record&quot; randomly</td>
</tr>
<tr>
<td>34</td>
<td>Write a &quot;record&quot; randomly</td>
</tr>
<tr>
<td>35</td>
<td>Return logical file size (even for random files)</td>
</tr>
<tr>
<td>36</td>
<td>Set record number for the next random read/write</td>
</tr>
<tr>
<td>37</td>
<td>Reset specified drive</td>
</tr>
<tr>
<td>40</td>
<td>Write a &quot;record&quot; randomly with zero fill</td>
</tr>
</tbody>
</table>

*These do not work under MP/M.

**Figure 5-1.** BDOS function calls
The function code number of the specific function call you want performed must be in register C.

If you need to hand a single-byte value to the BDOS, such as a character to be sent to the console, then you must arrange for this value to be in register E. If the value you wish to pass to the BDOS is a 16-bit value, such as the address of a buffer or a file control block (FCB), this value must be in register pair DE.

When the BDOS hands back a single-byte value, such as a keyboard character or a return code indicating the success or failure of the function you requested, it will be returned in register A. When the BDOS returns a 16-bit value, it will be in register pair HL.

On return from the BDOS, registers A and L will contain the same value, as will registers B and H. This odd convention stems from CP/M's origins in PL/M (Programming Language/Microprocessor), a language used by Intel on their MDS system. Thus, PL/M laid the foundations for what are known as "register calling conventions."

Figure 5-2. Equates for BDOS function code numbers

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>SYSRESET</td>
</tr>
<tr>
<td>0001</td>
<td>CONIN</td>
</tr>
<tr>
<td>0002</td>
<td>CONOUT</td>
</tr>
<tr>
<td>0003</td>
<td>READIN</td>
</tr>
<tr>
<td>0004</td>
<td>PUNOUT</td>
</tr>
<tr>
<td>0005</td>
<td>LISTOUT</td>
</tr>
<tr>
<td>0006</td>
<td>DIRCONID</td>
</tr>
<tr>
<td>0007</td>
<td>GETIO</td>
</tr>
<tr>
<td>0008</td>
<td>SETIO</td>
</tr>
<tr>
<td>0009</td>
<td>PRINTS</td>
</tr>
<tr>
<td>000A</td>
<td>READCONS</td>
</tr>
<tr>
<td>000B</td>
<td>CONST</td>
</tr>
<tr>
<td>000C</td>
<td>GETVER</td>
</tr>
<tr>
<td>000D</td>
<td>DSRESET</td>
</tr>
<tr>
<td>000E</td>
<td>SLDISK</td>
</tr>
<tr>
<td>000F</td>
<td>OPEN</td>
</tr>
<tr>
<td>0010</td>
<td>CLOSE</td>
</tr>
<tr>
<td>0011</td>
<td>SEARCH</td>
</tr>
<tr>
<td>0012</td>
<td>SEARCHN</td>
</tr>
<tr>
<td>0013</td>
<td>ERASE</td>
</tr>
<tr>
<td>0014</td>
<td>READSEQ</td>
</tr>
<tr>
<td>0015</td>
<td>WRITESEQ</td>
</tr>
<tr>
<td>0016</td>
<td>CREATE</td>
</tr>
<tr>
<td>0017</td>
<td>RENAME</td>
</tr>
<tr>
<td>0018</td>
<td>GETACDISK</td>
</tr>
<tr>
<td>0019</td>
<td>GETCUDISK</td>
</tr>
<tr>
<td>001A</td>
<td>SETDMA</td>
</tr>
<tr>
<td>001B</td>
<td>GETALVEC</td>
</tr>
<tr>
<td>001C</td>
<td>SETDSKRO</td>
</tr>
<tr>
<td>001D</td>
<td>GETRODISK</td>
</tr>
<tr>
<td>001E</td>
<td>SETPAT</td>
</tr>
<tr>
<td>001F</td>
<td>GETDPB</td>
</tr>
<tr>
<td>0020</td>
<td>SETGETUN</td>
</tr>
<tr>
<td>0021</td>
<td>READRAN</td>
</tr>
<tr>
<td>0022</td>
<td>WRITERAN</td>
</tr>
<tr>
<td>0023</td>
<td>GETFSIZ</td>
</tr>
<tr>
<td>0024</td>
<td>SETRANREC</td>
</tr>
<tr>
<td>0025</td>
<td>RESETD</td>
</tr>
<tr>
<td>0026</td>
<td>WRITERANZ</td>
</tr>
</tbody>
</table>
Chapter 5: The Basic Disk Operating System

The BDOS makes no guarantee about the contents of the other registers. If you need to preserve a value that is in a register, either store the value in memory or push it onto the stack. The BDOS uses its own stack space, so there is no need to worry about it consuming your stack.

To sum up, when you make a function request to the BDOS that requires a byte value, the code and the required entry and exit parameters will be as follows:

```
MVI C, FUNCTION#CODE
MVI E, SINGLE#BYTE
CALL BDOS
```

; C = function code
; E = single byte value
; Location 5
; A = return code or value
; or HL = return value

For those function requests that need to have an address passed to the BDOS, the calling sequence is

```
MVI C, FUNCTION#CODE
LXI D, ADDRESS
CALL BDOS
```

; C = function code
; DE = address
; Location 5
; A = return code or value
; or HL = return value

If a function request involves disk files, you will have to tell the BDOS the address of the FCB that you have created for the file. (Refer back to Chapter 3 for descriptions of the FCB.)

Many file processing functions return a value in register A that is either OFFH, indicating that the file named in the FCB could not be found, or equal to a value of 0, 1, 2, or 3. In the latter case, the BDOS is returning what is called a “directory code.” The number is the directory entry number that the BDOS matched to the file name in your FCB. At any given moment, the BDOS has a 128-byte sector from the directory in memory. Each file directory entry is 32 bytes, so four of them (numbered 0, 1, 2, and 3) can be processed at a time. The directory code indicates which one has been matched to your FCB.

References to CP/M “records” in the following descriptions mean 128-byte sectors. Do not confuse them with the logical records used by applications programs. Think of CP/M records as 128-byte sectors throughout.

**Function 0: System Reset**

Function Code: \( C = 00H \)
Entry Parameters: None
Exit Parameters: Does not return

**Example**

```
0000 = B$SYSRESET
0005 = BDOS
EQU 0
EQU 5

0000 0E00 MVI C, 0
0002 C30500 JMP BDOS
```

; System Reset
; BDOS entry point

; Set function code
; Note: you can use a JMP since
; you don’t get control back
Purpose

The system reset function makes CP/M do a complete reset, exactly the same as the warm boot function invoked when you transfer control to the WARM-BOOT point (refer to Figure 4-1).

In addition to resetting the BDOS, this function reloads the CCP, rebuilds the allocation vectors for the currently logged disks, sets the DMA address (used by CP/M to address the disk read/write buffer) to 80H, marks all disks as being Read/Write status, and transfers control to the CCP. The CCP then outputs its prompt to the console.

Notes

This function is most useful when you are working in a high-level language that does not permit a jump instruction to an absolute address in memory. Use it when your program has finished and you need to return control back to CP/M.

Function 1: Read Console Byte

Function Code: C = 01H
Entry Parameters: None
Exit Parameters: A = Data byte from console

Example

```
0001 = B$CONIN EQU 1 ;Console input
0005 = BDOS EQU 5 ;BDOS entry
0000 OE01 MVI C, B$CONIN ;Get function code
0002 CD0500 CALL BDOS
```

Purpose

This function reads the next byte of data from the console keyboard and puts it into register A. If the character input is a graphic character, it will be echoed back to the console. The only control characters that are echoed are CARRIAGE RETURN, LINE FEED, BACKSPACE, and TAB. In the case of a TAB character, the BDOS outputs as many spaces as are required to move the cursor to the next multiple of eight columns. All of the other control characters, including CONTROL-C, are input but are not echoed.

This function also checks for CONTROL-S (XOFF) to see if console output should be suspended, and for CONTROL-P (printer echo toggle) to see if console output should also be sent to the list device. If CONTROL-S is found, further output will be suspended until you type another character. CONTROL-P will enable the echoing of console output the first time it is pressed and disable it the second time.

If there is no incoming data character, this function will wait until there is one.

Notes

This function often hinders rather than helps, because it echoes the input. Whenever you need console input at the byte-by-byte level, you will usually want to suppress this echo back to the console. For instance, you may know that the "console" is actually a communications line such as a modem. You may be trying to accept a password that should not be echoed back. Or you may need to read a
cursor control character that would cause an undesirable side effect on the terminal if echoed there.

In addition, if you need more than a single character from the console, your program will be easier to use if the person at the console can take full advantage of the CCP-style line editing. This can best be done by using the Read Console String function (code 10, 0AH).

Read Console String also is more useful for single character input, especially when you are expecting a "Y" or "N" (yes or no) response. If you use the Read Console Byte function, the operator will have only one chance to enter the data. When you use Read Console String, however, users have the chance to type one character, change their minds, backspace, and type another character.

**Function 2: Write Console Byte**

Function Code: C = 02H
Entry Parameters: E = Data byte to be output
Exit Parameters: None

**Example**

0002 = B$CONOUT  EQU 2 ;Write Console Byte
0005 = BDOS       EQU 5 ;BDOS entry
0000 0E02         MVI C,B$CONOUT  ;Function code
0002 1E2A         MVI E, ' '   ;E = data byte to be output
0004 CD0500       CALL BDOS

**Purpose**

This function outputs the data byte in register E to the console. As with function 1, if the data byte is a TAB character, it will be expanded by the BDOS to the next column that is a multiple of eight. The BDOS also checks to see if there is an incoming character, and if there is, checks to see if it is a CONTROL-S (in which case console output is suspended) or CONTROL-P (in which case echoing of console output to the printer is toggled on or off).

**Notes**

You may have problems using this function to output cursor-addressing control sequences to the console. If you try to output a true binary cursor address to position 9, the BDOS will interpret this as a TAB character (ASCII code 9) and dutifully replace it with zero to eight blanks. If you need to output binary values, you must set the most significant bit of the character (use an ORI 80H, for example) so that it will not be taken as the ASCII TAB.

Here are two general-purpose subroutines that you will need for outputting messages. The first one, shown in Figure 5-3, outputs a null-byte-terminated message from a specified address. The second, in Figure 5-4, does essentially the same thing except that the message string follows immediately after the call to the subroutine.
The CP/M Programmer's Handbook

;MSGOUT (message out)
;Output null-byte-terminated message.

;Calling sequence
;  MESSAGE:  DB  'Message',0
;  LXI  H,MESSAGE
;  CALL  MSGOUT

;Exit Parameters
;  HL -> Null byte terminator

0002 = B$CONOUT  EQU  2  ;Write Console Byte
0005 = BDOS       EQU  5  ;BDOS entry point

MSGOUT:
0000 7E          MOV  A,M       ;Get next byte for output
0001 B7          ORA  A
0002 C8          RZ
0003 23          INX  H       ;Update message pointer
0004 E5          PUSH  H      ;Save updated pointer
0005 5F          MOV  E,A     ;Ready for BDOS
0006 0E02        MVI  C,B$CONOUT
0008 CD0500      CALL  BDOS
0009 5F          POP  H       ;Recover message pointer
000C C30000      JMP  MSGOUT   ;Go back for next character

Figure 5-3. Write console byte example, output null-byte terminated message from specified address

;MSGOUTI (message out in-line)
;Output null-byte-terminated message that follows the CALL to MSGOUTI.

;Calling sequence
;  CALL  MSGOUTI
;  DB  'Message',0
;  ... next instruction

;Exit Parameters
;  HL -> instruction following message

0002 = B$CONOUT  EQU  2  ;Write Console Byte
0005 = BDOS       EQU  5  ;BDOS entry point

MSGOUTI:
0000 E1          POP  H       ;HL -> message
0001 7E          MOV  A,M     ;Get next data byte
0002 23          INX  H       ;Update message pointer
0003 B7          ORA  A       ;Check if null byte
0004 C20800      JNZ  MSGOUTIC ;No, continue
0007 E9          PCHL

MSGOUTIC:
0008 E5          PUSH  H      ;Save message pointer
0009 5F          MOV  E,A     ;Ready for BDOS
000A 0E02        MVI  C,B$CONOUT ;Function code
000C CD0500      CALL  BDOS
000F C30000      JMP  MSGOUTI   ;Go back for next character.

Figure 5-4. Write console byte example, output null-byte terminated message following call to subroutine
Function 3: Read “Reader” Byte

Function Code: \( C = 03H \)
Entry Parameters: None
Exit Parameters: \( A = \) Character input

Example

```
0003 = BSREADIN EQU 3 ;Read "Reader" Byte
0005 = BDOS EQU 5 ;BDOS entry
0000 0E03 MVI C, BSREADIN ;Function code
0002 CD0500 CALL BDOS ;A = reader byte
```

Purpose

This function reads the next character from the logical “reader” device into register \( A \). In practice, the physical device that is accessed depends entirely on how your BIOS is configured. In some systems, there is no reader at all; this function will return some arbitrary value such as \( 1AH \) (the ASCII CONTROL-Z character, used by CP/M to denote “End of File”).

Control is not returned to the calling program until a character has been read.

Notes

Since the physical device (if any) used when you issue this request depends entirely on your particular BIOS, there can be no default standard for all CP/M implementations. This is one of the weaker parts of the BDOS.

You should “connect” the reader device by means of BIOS software to a serial port that can be used for communication with another system. This is only a partial solution to the problem, however, because this function call does not return control to your program until an incoming character has been received. There is no direct way that you can “poll” the reader device to see if an incoming character has been received. Once you make this function call, you lose control until the next character arrives; there is no function corresponding to the Read Console Status (function code 11, OBH) that will simply read status and return to your program.

One possible solution is to build a timer into the BIOS reader driver that returns control to your program with a dummy value in \( A \) if a specified period of time goes by with no incoming character. But this brings up the problem of what dummy value to use. If you ever intend to send and receive files containing pure binary information, there is no character in ASCII that you might not encounter in a legitimate context. Therefore, any dummy character you might choose could also be true data.

The most cunning solution is to arrange for one setting of the IOBYTE (which controls logical-device-to-physical-device mapping) to connect the console to the serial communication line. This done, you can make use of the Read Console Status function, which will return not the physical console status but the serial line status. Your program can then act appropriately if no characters are received within a specified time. Figure 5-11 shows a subroutine that uses this technique in the Set IOBYTE function (code 8, 08H).
Figure 5-5 shows an example subroutine to read lines of data from the reader device. It reads characters from the reader, stacking them in memory until either a LINE FEED or a specified number of characters has been received. Note that CARRIAGE RETURNS are ignored, and the input line is terminated by a byte of 00H. The convention of 00H-byte terminated strings and no CARRIAGE RETURNS is used because it makes for much easier program logic. It also conforms to the conventions of the C language.
Function 4: Write "Punch" Byte

Function Code: \( C = 04H \)
Entry Parameters: \( E = \) Byte to be output
Exit Parameters: None

Example

```
0004 = B$PUNOUT EQU 4 ; Write "Punch" Byte
0005 = BDOS EQU 5
0000 0E04 MVI C.B$PUNOUT ; Function code
0002 1E2A MVI E,"'" ; Data byte to output
0004 CDOS00 CALL BDOS
```

Purpose

This function is a counterpart to the Read "Reader" Byte described above. It outputs the specified character from register \( E \) to the logical punch device. Again, the actual physical device used, if any, is determined by the BIOS. There is no set standard for this device; in some systems the punch device is a "bit bucket," so called because it absorbs all data that you output to it.

Notes

The problems and possible solutions discussed under the Read "Reader" Byte function call also apply here. One difference, of course, is that this function outputs data, so the problem of an indefinite loop waiting for the next character is less likely to occur. However, if your punch device is connected to a communications line, and if the output hardware is not ready, the BIOS line driver will wait forever. Unfortunately, there is no legitimate way to deal with this problem since the BDOS does not have a function call that checks whether a logical device is ready for output.

Figure 5-6 shows a useful subroutine that outputs a 00H-byte terminated string to the punch. Wherever it encounters a LINE FEED, it inserts a CARRIAGE RETURN into the output data.

Function 5: Write List Byte

Function Code: \( C = 05H \)
Entry Parameters: \( E = \) Byte to be output
Exit Parameters: None

Example

```
0005 = B$LSTOUT EQU 5 ; Write List Byte
0005 = BDOS EQU 5
0000 0E05 MVI C.B$LSTOUT ; Function code
0002 1E2A MVI E,"'" ; Data byte to output
0004 CDOS00 CALL BDOS
```

Purpose

This function outputs the specified byte in register \( E \) to the logical list device. As with the reader and the punch, the physical device used depends entirely on the BIOS.
;WL$PUN
;Write line to punch device. Output terminates
;when a OOH byte is encountered.
;A carriage return is output when a line feed is
;encountered.

;Calling sequence
;  LXI  H, BUFFER
;  CALL  WL$PUN

;Exit parameters
;  HL -> OOH byte terminator

0004  =  B$PUNOUT  EQU  4
0005  =  BDOS     EQU  5
0006  =  CR       EQU  ODH ;Carriage return
000A  =  LF       EQU  OAH ;Line feed

WL$PUN:
0000  E5  PUSH  H ;Save buffer pointer
0001  7E  MOV  A,M ,Get
0002  B7  ORA  A ;Check
0003  CA2000  WL$PUNX ;Yes, exit
0006  FEA0  CPI  LF ;Check if line feed
0008  CC1600  C2  WL$PUNLF ;Yes, O/P CR
000B  5F  MOV  E,A ;Character to be output
000C  0E04  MVI  C,B$PUNOUT ;Function code
000E  C0500  CALL  BDOS ;Output character
0011  E1  POP  H ;Recover buffer pointer
0012  23  INX  H ;Increment to next char.
0013  C30000  JMP  WL$PUN ;Output next char

WL$PUNLF:
0016  0E04  MVI  C,B$PUNOUT ;Line feed encountered
0018  0E0D  MVI  E,CR ;Function code
001A  C0500  CALL  BDOS ;Output a CR
001D  3E0A  MVI  A,LF ;Recreate line feed
001F  C9  RET  ;Output LF

WL$PUNX:
0020  E1  POP  H ;Exit
0021  C9  RET  ;Balance the stack

Figure 5-6. Write line to punch device

Notes

One of the major problems associated with this function is that it does not deal
with error conditions very intelligently. You cannot be sure which physical device
will be used as the logical list device, and most standard BIOS implementations
will cause your program to wait forever if the printer is not ready or has run out of
paper. The BDOS has no provision to return any kind of error status to indicate
that there is a problem with the list device. Therefore, the BIOS will have to be
changed in order to handle this situation.

Figure 5-7 is a subroutine which outputs data to the list device. As you can see,
this is essentially a repeat of Figure 5-6, which performs the same function for the
logical punch device.
Chapter 5: The Basic Disk Operating System

;WLST
;Write line to list device. Output terminates
;when a 00H byte is encountered.
;A carriage return is output when a line feed is encountered.

;Calling sequence
;LXI H,BUFFER
;CALL WLST

;Exit parameters
;HL -> 00H byte terminator

0005 = BsLSTOUT EQU 5
0005 = BDOS EQU 5

000D = CR EQU 0DH ; Carriage return
000A = LF EQU 0AH ; Line feed

Figure 5-7. Write line to list device

Function 6: Direct Console I/O

Function Code: C = 06H
Entry Parameters: E = 0FFH for Input
E = Other than 0FFH for output
Exit Parameters: A = Input byte or status

Example

0006 = BDIRCONIO EQU 6 ; Direct (raw) Console I/O
0005 = BDOS EQU 5 ; BDOS entry point

; Example of console input
0000 0EO6
0002 1EFF
000A CD0500

MVI C,BDIRCONIO ; Function code
MVI E,OFFH ; OffH means input
CALL BDOS ; A = 00 if no char. waiting
; A = NZ if character input
Purpose

This function serves double duty: it both inputs and outputs characters from the console. However, it bypasses the normal control characters and line editing features (such as CONTROL-P and CONTROL-S) normally associated with console I/O. Hence the name “direct” (or “unadorned” as Digital Research describes it). If the value in register E is not OFFH, then E contains a valid ASCII character that is output to the console. The logic used is most easily understood when written in pseudo-code:

```
if this is an input request (E = OFFH)
{
    if console status indicates a character is waiting
    {
        read the char from the console and
        return to caller with char in A
    }
    else (no input character waiting) and
    return to caller with A = 00
}
else (output request)
{
    output the char in E to the console and
    return to caller
}
```

Notes

This function works well provided you never have to send a value of 0FFH or expect to receive a value of 00H. If you do need to send or receive pure binary data, you cannot use this function, since these values are likely to be part of the data stream.

To understand why you might want to send and receive binary data, remember that the logical “reader” does not have any method for you to check its status to see if an incoming character has arrived. All you can do is attempt to read a character (Read Reader Byte, function code 3). However, the BDOS will not give control back to you until a character arrives (which could be a very long time). One possibility is to logically assign the console to a communications line by the use of the IOBYTE (or some similar means) and then use this Direct I/O call to send and receive data to and from the line. Then you could indeed “poll” the communications line and avoid having your program go into an indefinite wait for an incoming character. An example subroutine using this technique is shown in Figure 5-11 under Set IOBYTE (function code 8).

Figure 5-8 shows a subroutine that uses the Direct Console Input and Output. Because this example is more complex than any shown so far, the code used to check the subroutine has also been included.

Function 7: Get IOBYTE Setting

Function Code: C = 07H
Entry Parameters: None
Exit Parameters: A = IOBYTE current value
Figure 5-8.  Read/write string from/to console using raw I/O
Figure 5-8. (Continued)
<table>
<thead>
<tr>
<th>RCS$SC:</th>
<th>RCS$UC:</th>
<th>RCS$GC:</th>
</tr>
</thead>
<tbody>
<tr>
<td>017F D5</td>
<td>PUSH D</td>
<td>Save character in C in buffer; HL -&gt; buffer pointer</td>
</tr>
<tr>
<td>0180 E5</td>
<td>PUSH H</td>
<td>Save terminator table pointer</td>
</tr>
<tr>
<td>0181 23</td>
<td>INX H</td>
<td>Save buffer pointer; HL -&gt; actual count in buffer</td>
</tr>
<tr>
<td>0182 5E</td>
<td>MOV E,M</td>
<td>Get actual count</td>
</tr>
<tr>
<td>0183 IC</td>
<td>INR E</td>
<td>Count of 0 points to first data byte</td>
</tr>
<tr>
<td>0184 1600</td>
<td>MOV D,O</td>
<td>Make word value of actual count</td>
</tr>
<tr>
<td>0186 19</td>
<td>DAD D</td>
<td>HL -&gt; next free data byte</td>
</tr>
<tr>
<td>0187 71</td>
<td>MOV M,C</td>
<td>Save data byte away</td>
</tr>
<tr>
<td>0188 E1</td>
<td>POP H</td>
<td>Recover buffer pointer</td>
</tr>
<tr>
<td>0189 D1</td>
<td>POP D</td>
<td>Recover terminator table; pointer</td>
</tr>
<tr>
<td>018A C9</td>
<td>RET</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>RCS$UC:</th>
<th>RCS$GC:</th>
</tr>
</thead>
<tbody>
<tr>
<td>018B E5</td>
<td>PUSH H</td>
</tr>
<tr>
<td>018C 7E</td>
<td>MOV A,M</td>
</tr>
<tr>
<td>018D 23</td>
<td>INX H</td>
</tr>
<tr>
<td>018E 34</td>
<td>INR M</td>
</tr>
<tr>
<td>018F BE</td>
<td>CMP M</td>
</tr>
<tr>
<td>0190 E1</td>
<td>POP H</td>
</tr>
<tr>
<td>0191 C9</td>
<td>RET</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>RCS$WT:</th>
<th>RCS$NA:</th>
<th>RCS$NE:</th>
</tr>
</thead>
<tbody>
<tr>
<td>0192 D5</td>
<td>PUSH D</td>
<td>Get character and execute; ECHO, ABORT and FOLD options</td>
</tr>
<tr>
<td>0193 E5</td>
<td>PUSH H</td>
<td>Save buffer pointer</td>
</tr>
<tr>
<td>0194 C5</td>
<td>PUSH B</td>
<td>Save option flags</td>
</tr>
<tr>
<td>0195 0E06</td>
<td>MVI C,B$DIRCONIO</td>
<td>Function code</td>
</tr>
<tr>
<td>0197 1EFF</td>
<td>MVI E,OFFH</td>
<td>Specify input</td>
</tr>
<tr>
<td>0199 CD5000</td>
<td>CALL BDOS</td>
<td></td>
</tr>
<tr>
<td>019D C9501</td>
<td>ORA A</td>
<td>Check if data waiting</td>
</tr>
<tr>
<td>019D C9502</td>
<td>JZ RCS$WT</td>
<td>Go back and wait</td>
</tr>
<tr>
<td>01A0 C1</td>
<td>POP B</td>
<td>Recover option flags</td>
</tr>
<tr>
<td>01A1 4F</td>
<td>MOV C, A</td>
<td>Save data byte</td>
</tr>
<tr>
<td>01A2 3E02</td>
<td>MVI A, RCS$ABORT</td>
<td>Check if abort option enabled</td>
</tr>
<tr>
<td>01A4 A0</td>
<td>ANA B</td>
<td>Save terminator table pointer</td>
</tr>
<tr>
<td>01A5 CAAEO1</td>
<td>JZ RCS$NA</td>
<td>No abort</td>
</tr>
<tr>
<td>01A8 3E03</td>
<td>MVI A,CTL$C</td>
<td>Check for control-C</td>
</tr>
<tr>
<td>01A9 B9</td>
<td>CMP C</td>
<td></td>
</tr>
<tr>
<td>01AB C0000</td>
<td>JZ O</td>
<td>Warm boot</td>
</tr>
<tr>
<td>01AE 3E04</td>
<td>MVI A, RCS$FOLD</td>
<td>Check if folding enabled</td>
</tr>
<tr>
<td>01B0 A0</td>
<td>ANA B</td>
<td></td>
</tr>
<tr>
<td>01B1 C4E501</td>
<td>CNZ TOUPPER</td>
<td>Convert to uppercase</td>
</tr>
<tr>
<td>01B4 3E01</td>
<td>MVI A, RCS$ECHO</td>
<td>Check if echo required</td>
</tr>
<tr>
<td>01B6 A0</td>
<td>ANA B</td>
<td></td>
</tr>
<tr>
<td>01B7 CAD101</td>
<td>JZ RCS$NE</td>
<td>No echo required</td>
</tr>
<tr>
<td>01BA C5</td>
<td>PUSH B</td>
<td>Save options and character</td>
</tr>
<tr>
<td>01BB 59</td>
<td>MOV E,C</td>
<td>Move character for output</td>
</tr>
<tr>
<td>01BC 0E06</td>
<td>MVI C,B$DIRCONIO</td>
<td>Echo character</td>
</tr>
<tr>
<td>01BE CD5000</td>
<td>CALL BDOS</td>
<td></td>
</tr>
<tr>
<td>01C1 C1</td>
<td>POP B</td>
<td>Recover options and character</td>
</tr>
<tr>
<td>01C2 3E00</td>
<td>MVI A, CR</td>
<td>Check if carriage return</td>
</tr>
<tr>
<td>01C4 B9</td>
<td>CMP C</td>
<td></td>
</tr>
<tr>
<td>01C5 CD2101</td>
<td>JNZ RCS$NE</td>
<td>No</td>
</tr>
<tr>
<td>01C8 C5</td>
<td>PUSH B</td>
<td>Save options and character</td>
</tr>
<tr>
<td>01C9 0E06</td>
<td>MVI C,B$DIRCONIO</td>
<td>Function code</td>
</tr>
<tr>
<td>01CB 1E0A</td>
<td>MVI E, LF</td>
<td>Output line feed</td>
</tr>
<tr>
<td>01CD CD5000</td>
<td>CALL BDOS</td>
<td></td>
</tr>
<tr>
<td>01DD C1</td>
<td>POP B</td>
<td>Recover options and character</td>
</tr>
<tr>
<td>01DE E1</td>
<td>POP H</td>
<td>Recover buffer pointer</td>
</tr>
<tr>
<td>01D2 D1</td>
<td>POP D</td>
<td>Recover terminator table</td>
</tr>
<tr>
<td>01D3 C9</td>
<td>RET</td>
<td>Character in C</td>
</tr>
</tbody>
</table>

Figure 5-8. (Continued)
Figure 5-8. (Continued)
Chapter 5: The Basic Disk Operating System

Example

```
0007 = B$GETIO EQU 7 ;Get IOBYTE
0005 = BDOS EQU 5 ;BDOS entry point
0000 OE07 MVI C.B$GETIO ;Function code
0002 CD0500 CALL BDOS ;A = IOBYTE
```

Purpose

This function places the current value of the IOBYTE in register A.

Notes

As we saw in Chapter 4, the IOBYTE is a means of associating CP/M’s logical devices (console, reader, punch, and list) with the physical devices supported by a particular BIOS. Use of the IOBYTE is completely optional. CP/M, to quote from the Digital Research CP/M 2.0 Alteration Guide, “...tolerate[s] the existence of the IOBYTE at location 0003H.”

In practice, the STAT utility provided by Digital Research does have some features that set the IOBYTE to different values from the system console.

Figure 5-9 summarizes the IOBYTE structure. A more detailed description was given in Chapter 4.

Each two-bit field can take on one of four values: 00, 01, 10, and 11. The value can be interpreted by the BIOS to mean a specific physical device, as shown in Table 4-1.

Figure 5-10 has equates that are used to refer to the IOBYTE. You can see that the values shown are declared using the SHL (shift left) operator in the Digital Research Assembler. This is just a reminder that the values are structured this way in the IOBYTE itself.

```
+-------+-------+-------+-------+
<table>
<thead>
<tr>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>No.</td>
<td>I</td>
<td>7</td>
<td>6</td>
</tr>
<tr>
<td>-------</td>
<td>-------</td>
<td>-------</td>
<td>-------</td>
</tr>
<tr>
<td>Logical Device</td>
<td>List</td>
<td>Punch</td>
<td>Reader Console</td>
</tr>
</tbody>
</table>
```

Figure 5-9. The IOBYTE structure
The CP/M Programmer's Handbook

Figure 5-10. IOBYTE equates

Function 8: Set IOBYTE

Function Code: \( C = 08H \)
Entry Parameters: \( E = \text{New IOBYTE value} \)
Exit Parameters: None

Example
This listing shows you how to assign the logical reader device to the BIOS’s console driver. It makes use of some equates from Figure 5-10.

```assembly
0000 Sl1008
2 SHL 2
0007
0008
0009
000A
000B
000C
000D
000E
000F
0010
0011
0012
0013
0014
0015
0016
0017
0018
0019
001A
001B
001C
001D
001E
001F
0100
0100 0E07
ORG
MVI
C.B#GETIO
;First, get current IOBYTE
```
This function sets the IOBYTE to a new value which is given in register E. Because of the individual bit fields in the IOBYTE, you will normally use the Get IOBYTE function, change some bits in the current value, and then call the Set IOBYTE function.

Notes

You can use the Set IOBYTE, Get IOBYTE, and Direct Console I/O functions together to create a small program that transforms your computer system into a "smart" terminal. Any data that you type on your keyboard can be sent out of a serial communications line to another computer, and any data received on the line can be sent to the screen.

Figure 5-11 shows this program and illustrates the use of all of these functions. For this program to function correctly, your BIOS must check the IOBYTE and detect whether the logical console is connected to the physical console (with the IOBYTE set to TTY:) or to the input side of the serial communications line (with the IOBYTE set to RDR:).

Figure 5-11 shows how to use the Get and Set IOBYTE functions to make a simple terminal emulator. For this example to work, the BIOS must detect the Console Value as 3 (IOSCUC1) and connect Console Status, Input, and Output functions to the communications line.

```assembly
0006 = BDIRCONIO EQU 6 ;Direct console input/output
0007 = BDIOETIO EQU 7 ;Get IOBYTE
0008 = BDIOETIO EQU 8 ;Set IOBYTE
0005 = BDOS EQU 5 ;BDOS entry point
0003 = IOCOM EQU 00000001B ;Console mask for IOBYTE
0001 = IOCCRT EQU 1 ;Console -> CRT;
0003 = IOSCUC1 EQU 3 ;Console -> user console #1

TERMS:
0000 CD2A00 CALL SETCRTC ;Connect console -> CRT;

TERMCKS:
0003 CD5200 CALL CONST ;Get CRT status
0006 CA2400 CALL TERMINOR ;No console input
0009 CD4BOO CALL CONIN ;Get keyboard character
0000 CD3000 CALL SETCOMM ;Connect console -> comm. line
000F CD4500 CALL CONOUT ;Output to comm. line

TERMSCS:
0012 CD5200 CALL CONST ;Check comm. status
0015 CA0000 CALL TERM ;Get "console" status
0018 CD4B00 CALL CONIN ;No incoming comm. character

Figure 5-11. Simple terminal emulator
Function 9: Display "$"-Terminated String

Function Code: \( C = 09H \)
Entry Parameters: \( DE = \) Address of first byte of string
Exit Parameters: None

Example

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>B$PRINTS</td>
<td>EQU 9</td>
<td>Print &quot;$&quot;-Terminated String</td>
</tr>
<tr>
<td>0005</td>
<td>BDOS</td>
<td>EQU 5</td>
<td>BDOS entry point</td>
</tr>
<tr>
<td>0000</td>
<td>CR</td>
<td>EQU 0DH</td>
<td>Carriage return</td>
</tr>
<tr>
<td>0000A</td>
<td>LF</td>
<td>EQU 0AH</td>
<td>Line feed</td>
</tr>
<tr>
<td>0009</td>
<td>TAB</td>
<td>EQU 09H</td>
<td>Horizontal tab</td>
</tr>
</tbody>
</table>
Chapter 5: The Basic Disk Operating System

This function outputs a string of characters to the console device. The address of this string is in registers DE. You must make sure that the last character of the string is "$"; the BDOS uses this character as a marker for the end of the string. The "$" itself does not get output to the console.

While the BDOS is outputting the string, it expands tabs as previously described, checks to see if there is an incoming character, and checks for CONTROL-S (XOFF, which stops the output until another character is entered) or CONTROL-P (which turns on or off echoing of console characters to the printer).

Notes

One of the biggest drawbacks of this function is its use of "$" as a terminating character. As a result, you cannot output a string with a "$" in it. To be truly general-purpose, it would be better to use a subroutine that used an ASCII NUL (00H) character as a terminator, and simply make repetitive calls to the BDOS CONOUT function (code 2). Figure 5-3 is an example of such a subroutine.

Figure 5-12 shows an example of a subroutine that outputs one of several messages. It selects the message based on a message code that you give it as a parameter. Therefore, it is useful for handling error messages; the calling code can pass it an 8-bit error code. You may find it more flexible to convert this subroutine to using 00H-byte-terminated messages using the techniques shown in Figure 5-3.

Purpose

<table>
<thead>
<tr>
<th>HL -&gt; message table</th>
</tr>
</thead>
<tbody>
<tr>
<td>DB 3 ;Number of messages in table</td>
</tr>
<tr>
<td>DW MSG0 ;Address of text (A = 0)</td>
</tr>
<tr>
<td>DW MSG1 ;(A = 1)</td>
</tr>
<tr>
<td>DW MSG2 ;(A = 2)</td>
</tr>
<tr>
<td>MSG0: DB 'Message text$'</td>
</tr>
<tr>
<td>...etc.</td>
</tr>
<tr>
<td>A = Message code (from 0 on up)</td>
</tr>
<tr>
<td>B = Output CR/LF if non-zero</td>
</tr>
</tbody>
</table>

Figure 5-12. Display $-terminated message on console
Figure 5-12. (Continued)

Function 10: Read Console String

Function Code:  
Entry Parameters: DE = Address of string buffer
Exit Parameters: String buffer with console bytes in it

Example

0000 A = BREADCONS EQU 10 :Read Console String
0004 = BDOS EQU 5 :BDOS entry point
This function reads a string of characters from the console device and stores them in a buffer (address in DE) that you define. Full line editing is possible: the operator can backspace, cancel the line and start over, and use all the normal control functions. What you will ultimately see in the buffer is the final version of the character string entered, without any of the errors or control characters used to do the line editing.

The buffer that you define has a special format. The first byte in the buffer tells the BDOS the maximum number of characters to be accepted. The second byte is reserved for the BDOS to tell you how many characters were actually placed in the buffer. The following bytes contain the characters of the string.

Character input will cease either when a CARRIAGE RETURN is entered or when the maximum number of characters, as specified in the buffer, has been received. The CARRIAGE RETURN is not stored in the buffer as a character—it just serves as a terminator.

If the first character entered is a CARRIAGE RETURN, then the BDOS sets the "characters input" byte to 0. If you attempt to input more than the maximum number of characters, the "characters input" count will be the same as the maximum value allowed.

This function is useful for accepting console input, especially because of the line editing that it allows. It should be used even for single-character responses, such as “Y/N” (yes or no), because the operator can type “Y”, backspace, and overtype with “N”. This makes for more “forgiving” programs, tolerant of humans who change their minds.

Figure 5-13 shows an example subroutine that uses this function. It accepts console input, matches the input against a table, and transfers control to the appropriate subroutine. Many interactive programs need to do this; they accept an operator command and then transfer control to the appropriate command processor to deal with that command.

This example also includes two other subroutines that are useful in their own right. One compares null-byte-terminated strings (FSCMP), and the other converts, or “folds,” lowercase letters to uppercase (FOLD).
The CP/M Programmer's Handbook

This subroutine returns one of several addresses selected from a table by matching keyboard input against specified strings. It is normally used to switch control to a particular subprocessor according to an option entered by the operator from the keyboard.

Character string comparisons are performed with case-folding; that is, lowercase letters are converted to uppercase.

If the operator input fails to match any of the specified strings, then the carry flag is set. Otherwise, it is cleared.

Entry parameters

HL -> Subprocessor select table

This has the form:

```
DW TEXT0, SUBPROCO
DW TEXT1, SUBPROCI
DW 0 ; Terminator
```

TEXT0: DB 'add', 0
TEXT1: DB 'subtract', 0
SUBPROCO: Code for processing ADD function.
SUBPROCI: Code for processing SUBTRACT function.

Exit parameters

DE -> operator input string (OOH-terminated input string).
Carry Clear, HL -> subprocessor.
Carry Set, HL = 0000H.

Calling sequence

```
LXI H, SUBPROCTAB ; Subprocessor table
CALL RSA
JC ERROR ; Carry set only on error
LXI D, RETURN ; Fake CALL instruction
PUSH D
PCHL
RETURN
```

Entry parameters

HL -> Subprocessor select table

This has the form:

```
DW TEXT0, SUBPROCO
DW TEXT1, SUBPROCI
DW 0 ; Terminator
```

TEXT0: DB 'add', 0
TEXT1: DB 'subtract', 0
SUBPROCO: Code for processing ADD function.
SUBPROCI: Code for processing SUBTRACT function.

Exit parameters

DE -> operator input string (OOH-terminated input string).
Carry Clear, HL -> subprocessor.
Carry Set, HL = 0000H.

Calling sequence

```
LXI H, SUBPROCTAB ; Subprocessor table
CALL RSA
```

000A = BREADCONS EQU 10 ; Read console string into buffer
0005 = BDOS EQU 5 ; BDOS entry point
0050 = RSA$BL EQU 80 ; Buffer length
0050 00 RSA$BUFF: DB RSA$BL ; Max. no. of characters
0001 00 RSA$ACTC: DB 0 ; Actual no. of characters
0002 RSA$BUFFC: DS RSA$BL ; Buffer characters
0052 00 DB 0 ; Safety terminator
0053 2B DCX H ; Adjust Subprocessor pointer
0054 2B DCX H ; for code below
0055 E5 PUSH H ; Top of stack (TOS) -> subproc. table - 2
0056 00E0A MMI C,BREADCONS ; Function code
0058 110000 LIX D, RSA$BUFF ; DE -> buffer
0058 CD0500 CALL BDOS ; Read operator input and
005E 210100 LIX H, RSA$ACTC ; Convert to OOH-terminated
0061 5E MOV E,M ; HL -> actual no. of chars. input
0062 1600 MVI D, 0 ; Make into word value
0064 23 INX H ; HL -> first data character
0065 19 DAD D ; HL -> first UNUSED character in buffer
0066 3600 MVI M, 0 ; Make input buffer OOH terminated

RSA$ML:

```
0068 E1 POP H ; Compare input to specified values
0069 23 INX H ; Main loop
006A 23 INX H ; Recover subprocessor table pointer
006B 5E MOV E,M ; Move to top of next entry
```

Figure 5-13. Read console string for keyboard options
Chapter 5: The Basic Disk Operating System

Figure 5-13. (Continued)
The CP/M Programmer's Handbook

Function 11: Read Console Status

Function Code: C = 0BH
Entry Parameters: None
Exit Parameters: A = 00H if no incoming data byte
A = 0FFH if incoming data byte

Example

```
0000 BCONST EQU 11 ;Get Console Status
0005 BDOS EQU 5 ;BDOS entry point
0000 0EOB
0002 CD0500
0000 0EOC MVI C.BCONST ;Function code
0002 CD0500 CALL BDOS ;A = 00 if no character waiting
                      ;A = OFFH if character waiting
```

Purpose

This function tells you whether a console input character is waiting to be processed. Unlike the Console Input functions, which will wait until there is input, this function simply checks and returns immediately.

Notes

Use this function wherever you want to interrupt an executing program if a console keyboard character is entered. Just put a Console Status call in the main loop of the program. Then, if the program detects that keyboard data is waiting, it can take the appropriate action. Normally this would be to jump to location 0000H, thereby aborting the current program and initiating a warm boot.

Figure 5-11 is an example subroutine that shows how to use this function.

Function 12: Get CP/M Number

Function Code: C = 0CH
Entry Parameters: None
Exit Parameters: HL = Version number code

Example

```
0000 BGETVER EQU 12 ;Get CP/M Version Number
0005 BDOS EQU 5 ;BDOS entry point
0000 0EOC MVI C.BGETVER ;Function code
0002 CD0500 CALL BDOS ;H = 00 for CP/M
                      ;L = version (e.g. 22H for 2.2)
```

Purpose

This function tells you which version of CP/M you are currently running. A two-byte value is returned:

H = 00H for CP/M, H = 01H for MP/M
L = 00H for all releases before CP/M 2.0
L = 20H for CP/M 2.0, 21H for 2.1, 22H for 2.2, and so on for any subsequent releases.
This information is of interest only if your program has some version-specific logic built into it. For example, CP/M version 1.4 does not support the same Random File Input/Output operations that CP/M 2.2 does. Therefore, if your program uses Random I/O, put this check at the beginning to ensure that it is indeed running under the appropriate version of CP/M.

Notes

Figure 5-14 is a subroutine that checks the current CP/M version number, and, if it is not CP/M 2.2, displays an explanatory message on the console and does a warm boot by jumping to location 0000H.

Function 13: Reset Disk System

Function Code: C = ODH
Entry Parameters: None
Exit Parameters: None

Determine the CP/M version number
Purpose

This function requests CP/M to completely reset the disk file system. CP/M then resets its internal tables, selects logical disk A as the default disk, resets the DMA address back to 0080H (the address of the buffer used by the BDOS to read and write to the disk), and marks all logical disks as having Read/Write status.

The BDOS will then have to log in each logical disk as each disk is accessed. This involves reading the entire file directory for the disk and rebuilding the allocation vectors (which keep track of which allocation blocks are free and which are used for file storage).

Notes

This function lets you change the diskettes under program control. If the operator were to simply change diskettes, without CP/M knowing about it, the next access to the (now different) diskette would force CP/M to declare the disk Read-Only, thwarting any further attempts to write on the diskette. If you need to reset one or two disks, rather than the entire disk system, look ahead to the Reset Disk function (code 37) described at the end of this chapter.

Figure 5-15 shows a simple subroutine that outputs a message on the console, requesting that the diskette in a specified drive be changed. It then issues a Reset Disk function call to make sure that CP/M will log in the diskette on the next access to the drive.

```
;CDISK
;Change disk
;This subroutine displays a message requesting the
;user to change the specified logical disk, then waits
;for a carriage return to be pressed. It then issues
;a Disk Reset and returns to the caller.

;Entry parameters
; A = Logical disk to be changed (A = 0, B = 1)

;Exit parameters
; None

;Calling sequence
; MVI A,0
; CALL CDISK

0000 = B@DSKRESET EQU 13 ;Disk Reset function code
0005 = BDOS EQU 5 ;BDOS entry point
0000 0E00 MVI C,B@DSKRESET ;Function code
0002 CD0500 CALL BDOS
```

Figure 5-15. Reset requested disk drive
Chapter 5: The Basic Disk Operating System

Function 14: Select Logical Disk

Function Code: \( C = 0EH \)

Entry Parameters: \( E = \) Logical Disk Code

- \( 00H \) = Drive A
- \( 01H \) = Drive B and so on

Exit Parameters: None

Example

```assembly
000E =                 BSELDSK EQU 14 ; Select Logical Disk
0005 =                 BDOS EQU 5   ; BDOS entry point
0000 0E0E MVI C, BSELDSK ; Function code
0002 1E00 MVI E, 0     ; \( E = 0 \) for A, 1 for B; etc.
0004 CD0500 CALL BDOS
```

Purpose

This function makes the logical disk named in register \( E \) the default disk. All subsequent references to disk files that do not specify the disk will use this default.

When you reference a disk file that does have an explicit logical disk in its name you do not have to issue another Select Disk function; the BDOS will take care of that for you.

Notes

Notice the way in which the logical disk is specified in register \( E \). It is not the same as the disk drive specification in the first byte of the file control block. In the FCB, a value of \( 00H \) is used to mean “use the current default disk” (as specified in the last Select Disk call or by the operator on the console). With this function, a
value of 00H in register A means that A is the selected drive, a value of 01H means drive B, and so on to 0FH for drive P, allowing 16 drives in the system.

If you select a logical disk that does not exist in your computer system, the BDOS will display the following message:

BDOS Err on J: Select

If you type a CARRIAGE RETURN in order to proceed, the BDOS will do a warm boot and transfer control back to the CCP. To avoid this, you must rely on the computer operator not to specify nonexistent disks or build into your program the knowledge of how many logical disk drives are on the system.

Another problem with this function is that you cannot distinguish a logical disk for which the appropriate tables have been built into the BIOS, but for which there is no physical disk drive. The BDOS does not check to see if the drive is physically present when you make the Select Disk call. It merely sets up some internal values ready to access the logical disk. If you then attempt to access this nonexistent drive, the BIOS will detect the error. What happens next is completely up to the BIOS. The standard BIOS will return control to the BDOS, indicating an error condition. The BDOS will output the message

BDOS Err on C: Bad Sector

You then have a choice. You can press CARRIAGE RETURN, in which case the BDOS will ignore the error and attempt to continue with whatever appears to have been read in. Or you can enter a CONTROL-C, causing the program to abort and CP/M to perform a warm boot.

Note that the Select Disk function does not return any values. If your program gets control back, you can assume that the logical disk you asked for at least has tables declared for it.

**Function 15: Open File**

Function Code: C = 0FH

Entry Parameters: DE = Address of file control block

Exit Parameters: A = Directory code

---

**Example**

```
0000 00  FCB:  File control block
0001 46494C454E  DB  'FILENAME'  ;Search on default disk drive
0002 545950  DB  'TYP'  ;File type
0003 00  DB  0  ;Extent
0005 00  DB  0  ;Reserved for CP/M
0006 00  DB  0  ;Records used in this extent
0007 0000000000  DB  0,0,0,0,0,0,0,0 ;Allocation blocks used
0008 00000000  DB  0,0,0,0,0,0,0,0 ;Sequential rec. to read/write
0009 00  DB  0  ;Extent
000A 00  DB  0  ;Reserved for CP/M
000B 00  DB  0  ;Records used in this extent
000C 0000000000  DB  0,0,0,0,0,0,0,0 ;Allocation blocks used
000D 00000000  DB  0,0,0,0,0,0,0,0 ;Sequential rec. to read/write
000E 00  DB  0  ;Extent
000F 00  DB  0  ;Reserved for CP/M
0010 000000000000  DB  0,0,0,0,0,0,0,0 ;Allocation blocks used
0011 00000000  DB  0,0,0,0,0,0,0,0 ;Sequential rec. to read/write
0012 00  DB  0  ;Extent
0013 00  DB  0  ;Reserved for CP/M
0014 00  DB  0  ;Records used in this extent
0015 0000000000  DB  0,0,0,0,0,0,0,0 ;Allocation blocks used
0016 00000000  DB  0,0,0,0,0,0,0,0 ;Sequential rec. to read/write
0017 00  DB  0  ;Extent
0018 00  DB  0  ;Reserved for CP/M
0019 00  DB  0  ;Records used in this extent
001A 0000000000  DB  0,0,0,0,0,0,0,0 ;Allocation blocks used
001B 00000000  DB  0,0,0,0,0,0,0,0 ;Sequential rec. to read/write
001C 00  DB  0  ;Extent
001D 00  DB  0  ;Reserved for CP/M
001E 00  DB  0  ;Records used in this extent
001F 0000000000  DB  0,0,0,0,0,0,0,0 ;Allocation blocks used
```

---

**Example**

```
000F = B@OPEN  EQU 15 ;Open File
0005 = BDOS  EQU 5 ;BDOS entry point
          FCB:  ;File control block
          FCB@DISK:  DB 0  ;Search on default disk drive
          FCB@NAME:  DB 'FILENAME'  ;File name
          FCB@TYPE:  DB 'TYP'  ;File type
          FCB@EXTENT:  DB 0  ;Extent
          FCB@RESV:  DB 0  ;Reserved for CP/M
          FCB@REUSED:  DB 0  ;Records used in this extent
          FCB@ABUSED:  DB 0,0,0,0,0,0,0,0 ;Allocation blocks used
          FCB@SEQREC:  DB 0  ;Sequential rec. to read/write
```
Chapter 5: The Basic Disk Operating System

Purpose

This function opens a specified file for reading or writing. The FCB, whose address must be in register DE, tells CP/M the user number, the logical disk, the file name, and the file type. All other bytes of the FCB will normally be set to 0.

The code returned by the BDOS in register A indicates whether the file has been opened successfully. If A contains OFFH, then the BDOS was unable to find the correct entry in the directory. If A = 0, 1, 2, or 3, then the file has been opened.

Notes

The Open File function searches the entire file directory on the specified logical disk looking for the file name, type, and extent specified in the FCB; that is, it is looking for an exact match for bytes 1 through 14 of the FCB. The file name and type may be ambiguous; that is, they may contain "?" characters. In this case, the BDOS will open the first file in the directory that matches the ambiguous name in the FCB. If the file name or type is shorter than eight or three characters respectively, then the remaining characters must be filled with blanks.

When the BDOS searches the file directory, it expects to find an exact match with each character of the file name and type, including lowercase letters or nongraphic characters. However, the BDOS uses only the least significant seven bits of each character—the most significant bit is used to indicate special file status characteristics, or attributes.

By matching the file extent as well as the name and type, you can, if you wish, open the file at some point other than its beginning. For normal sequential access, you would not usually want to do this, but if your program can predict which file extent is required, this is a method of moving directly to it.

It is also possible to open the same file more than once. Each instance requires a separate FCB. The BDOS is not aware that this is happening. It is really only safe to do this when you are reading the file. Each FCB can be used to read the file independently.

Once the file has been found in the directory, the number of records and the allocation blocks used are copied from the directory entry into the FCB (bytes 16 through 31). If the file is to be accessed sequentially from the beginning of the file, the current record (byte 32) must be set to zero by your program.

The value returned in register A is the relative directory entry number of the entry that matched the FCB. As previously explained, the buffer that CP/M uses holds a 128-byte record from the directory with four directory entries numbered 0, 1, 2, and 3. This directory code is returned by almost all of the file-related BDOS functions, but under normal circumstances you will be concerned only with whether the value returned in A is OFFH or not.

Figure 5-16 shows a subroutine that takes a 00H-byte terminated character
string, creates a valid FCB, and then opens the specified file. Shown as part of this example is the subroutine BF (Build FCB). It performs the brunt of the work of converting a string of ASCII characters into an FCB-style disk, file name, and type.

Figure 5-16. Open file request
Figure 5-16. (Continued)
Figure 5-16. (Continued)

Function 16: Close File

Function Code: C = 10H
Entry Parameters: DE = Address of file control block
Exit Parameters: A = Directory code

Example

```
0010  =  B$CLOSE
0005  =  BDOS
0000  =  FCB:  MVIC,B$CLOSE
0024 OE10  =  MVI C,B$CLOSE
0026 110000 =  LXI D,FCB
0029 CD0500 =  CALL BDOS
```

Purpose

This function terminates the processing of a file to which you have written information. Under CP/M you do not need to close a file that you have been reading. However, if you ever intend for your program to function correctly under MP/M (the multi-user version of CP/M) you should close all files regardless of their use.
The Close File function, like Open File, returns a directory code in the A register. Register A will contain OFFH if the BDOS could not close the file successfully. If A is 0, 1, 2, or 3, then the file has been closed.

Notes

When the BDOS closes a file to which data has been written, it writes the current contents of the FCB out to the disk directory, updating an existing directory entry by matching the disk, name, type, and extent number in the same manner that the Open File function does.

Note that the BDOS does not transfer the last record of the file to the disk during the close operation. It merely updates the file directory. You must arrange to flush any partly filled record to the disk. If the file that you have created is a standard CP/M ASCII text file, you must arrange to fill the unused portion of the record with the standard IAH end-of-file characters as CP/M expects, as explained in the section on the Write Sequential function (code 21).

Function 17: Search for First Name Match

Function Code: C = 11H
Entry Parameters: DE = Address of file control block
Exit Parameters: A = Directory code

Example

```
0011 = B$SEARCHF
0005 = BDOS
0000 00 FCB$DISK: DB 0 ;Search on default disk drive
0001 46494C453F FCB$NAME: DB 'FILE????' ;Ambiguous file name
0009 543F50 FCB$TYP: DB 'T?P' ;Ambiguous file type
000C 00 FCB$EXTENT: DB 0 ;Extent
000D 00 FCB$RESV: DB 0,0 ;Reserved for CP/M
000F 00 FCB$RESCUSED: DB 0 ;Records used in this extent
0010 0000000000 FCB$ABUSED: DB 0,0,0,0,0,0,0,0 ;Allocation blocks used
0018 0000000000
0020 00 FCB$SEQREC: DB 0 ;Sequential rec. to read/write
0021 00 FCB$RANREC: DB 0 ;Random rec. to read/write
0023 00 FCB$RANRECO: DB 0 ;Random rec. overflow byte (MS)
```

Purpose

This function scans down the file directory for the first entry that matches the file name, type, and extent in the FCB addressed by DE. The file name, type, and extent may contain a "?” (ASCII 3FH) in one or more character positions. Where a "?” occurs, the BDOS will match any character in the corresponding position in the file directory. This is known as ambiguous file name matching.

The first byte of an FCB normally contains the logical disk number code. A value of 0 indicates the default disk, while 1 means disk A, 2 is B, and so on up to a
possible maximum of 16 for disk P. However, if this byte contains a "?", the BDOS will search the default logical disk and will match the file name and type regardless of the user number. This function is normally used in conjunction with the Search Next function (which is described immediately after this function). Search First, in the process of matching a file, leaves certain variables in the BDOS set, ready for a subsequent Search Next.

Both Search First and Search Next return a directory code in the A register. With Search First, $A = 0$FFH when no files match the FCB; if a file match is found, $A$ will have a value of 0, 1, 2, or 3.

Notes

To locate the particular directory entry that either the Search First or Search Next function matched, multiply the directory code returned in A by the length of a directory entry (32 bytes). This is easily done by adding the A register to itself five times (see the code in Figure 5-17 near the label GNFC). Then add the DMA address to get the actual address where the matched directory entry is stored.

There are many occasions when you may need to write a program that will accept an ambiguous file name and operate on all of the file names that match it. (The DIR and ERA commands built into the CCP are examples that use ambiguous file names.) To do this, you must use several BDOS functions: the Set DMA Address function (code 26, described later in this chapter), this function (Search First), and Search Next (code 18). All of this is shown in the subroutine given in Figure 5-17.

```
;GNF
;This subroutine returns an FCB setup with either the first file matched by an ambiguous file name, or (if specified by entry parameter) the next file name.

;Note: this subroutine is context sensitive. You must not have more than one ambiguous file name sequence in process at any given time.

Warning: This subroutine changes the DMA address inside the BDOS.

;Entry parameters
;DE -> Possibly ambiguous file name
;   (OO-byte terminated)
;   (Only needed for FIRST request)
;HL -> File control block
;A = 0: Return FIRST file name that matches
;NZ: Return NEXT file name that matches

;Exit parameters
;Carry set: A = 0FFH, no file name matches
;A not = 0FFH, error in input file name
;Carry clear: FCB setup with next name
;HL -> Directory entry returned
;   by Search First/Next

;Calling sequence
LXI D,FILENAME
LXI H,FCB
```

Figure 5-17. Search first/next calls for ambiguous file name
Chapter 5: The Basic Disk Operating System

Figure 5-17. (Continued)

<table>
<thead>
<tr>
<th>1</th>
<th>MVI A, 0 for MVI A, 1 for NEXT</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>CALL GNF</td>
</tr>
<tr>
<td>0011</td>
<td>B$SEARCHF EQU 17 ;Search for first file name</td>
</tr>
<tr>
<td>0012</td>
<td>B$SEARCHN EQU 18 ;Search for next file name</td>
</tr>
<tr>
<td>001A</td>
<td>B$SETDMA EQU 26 ;Set up DMA address</td>
</tr>
<tr>
<td>0005</td>
<td>BDOS EQU 5 ;BDOS entry point</td>
</tr>
<tr>
<td>0080</td>
<td>GNFDMA EQU 80H ;Default DMA address</td>
</tr>
<tr>
<td>008D</td>
<td>GNF$VL EQU 13 ;Save length (no. of chars to move)</td>
</tr>
<tr>
<td>0000</td>
<td>GNF$; DS GNF$VL ;Save area for file name/type</td>
</tr>
</tbody>
</table>

GNF: |
| 0000 E5 | PUSH H ;Save FCB pointer |
| 000E D5 | PUSH D ;Save file name pointer |
| 000F F5 | PUSH PSM ;Save first/next flag |
| 0010 118000 | LXI D, GNFDMA ;Set DMA to known address |
| 0013 0E1A | MVI C, B$SETDMA ;Function code |
| 0015 CD0500 | CALL BDOS ;Recover first/next flag |
| 0018 F1 | POP PSM |
| 0019 E1 | POP H ;Recover file name pointer |
| 001A D1 | POP D ;Recover FCB pointer |
| 001B D5 | PUSH D ;Recover FCB pointer |
| 001C B7 | ORA A ;Check if FIRST or NEXT |
| 001D C23E00 | UNZ GNFN ;NEXT |
| 0020 CD9300 | CALL BF ;Build file control block |
| 0023 E1 | POP H ;Recover FCB pointer (to balance stack) |
| 0024 08 | RC ;Return if error in file name |
| 0025 E5 | PUSH H ;Recover FCB pointer |
| 0026 110000 | LXI D, GNF$V ;Move ambiguous file name to |
| 0029 OEO0 | MVI C, GNF$VL ;Save area |
| 002B C66A00 | CALL MOVE |
| 002E D1 | POP D ;Set save length |
| 002F D5 | PUSH D ;and reserve |
| 0030 0E11 | MVI C, B$SEARCHF ;Search FIRST |
| 0032 CD0500 | CALL BDOS ;Re-find the file |
| 0035 E1 | POP H |
| 0036 FEFF | CPI OFFH ;Check for error |
| 0038 CA7000 | JZ GNFEX ;Error exit |
| 003B C9500 | JMP GNF ;Common code |

GNFN: |
| 003E CD7F00 | CALL GNFZ |
| 0041 D1 | POP D ;Recover FCB address |
| 0042 D5 | PUSH D ;and reserve |
| 0043 OE11 | MVI C, B$SEARCHF ;Re-find the file |
| 0045 CD0500 | CALL BDOS ;Recover FCB pointer |
| 0048 D1 | POP D ;and reserve |
| 0049 D5 | PUSH D |
| 004A 210000 | LXI H, GNF$V ;Move file name from save area |
| 004B 0E0D | MVI C, GNF$VL ;Into FCB |
| 004F C68A00 | CALL MOVE ;Save area length |
| 0052 OE12 | MVI C, B$SEARCHN ;Search NEXT |
| 0054 CD0500 | CALL BDOS |
| 0057 E1 | POP H ;Recover FCB address |
| 0058 FEFF | CPI OFFH ;Check for error |
| 005A CA7000 | JZ GNFEX ;Error exit |

GNFC: |
| 005D E5 | PUSH H ;Save FCB address |
| 005E 07 | ADD A ;Multiply BDOS return code \times 32 |
Figure 5-17. (Continued)
Function 18: Search for Next Name Match

Function Code:  C = 12H
Entry Parameters: None (assumes previous Search First call)
Exit Parameters:  A = Directory code

Example

```
0012 = B@SEARCHN EQU 18 ;Search Next
0005 = BDOS EQU 5 ;BDOS entry point
0000 0E12 MVI C, B@SEARCHN ;Function code
;Notes: No FCB pointer
;You must precede this call
;with a call to Search First
;A = 0, 1, 2, 3
;(A & 32) + DMA -> directory
;entry
;A = 0FFH if file name not
;found
0002 CD0500 CALL BDOS
```

Purpose

This function searches down the file directory for the next file name, type, and extent that match the FCB specified in a previous Search First function call.

Search First and Search Next are the only BDOS functions that must be used together. As you can see, the Search Next function does not require an FCB address as an input parameter—all the necessary information will have been left in the BDOS on the Search First call.

Like Search First, Search Next returns a directory code in the A register; in this case, if A = 0FFH, it means that there are no more files that match the file control block. If A is not 0FFH, it will be a value of 0, 1, 2, or 3, indicating the relative directory entry number.

Notes

There are two ways of using the Search First/Next calls. Consider a simple file copying program that takes as input an ambiguous file name. You could scan the file directory, matching all of the possible file names, possibly displaying them on the console, and storing the names of the files to be copied in a table inside your program. This would have the advantage of enabling you to present the file names
to the operator before any copying occurred. You could even arrange for the operator to select which files to copy on a file-by-file basis. One disadvantage would be that you could not accurately predict how many files might be selected. On some hard disk systems you might have to accommodate several thousand file names.

The alternative way of handling the problem would be to match one file name, copy it, then match the next file name, copy it, and so on. If you gave the operator the choice of selecting which files to copy, this person would have to wait at the terminal as each file was being copied, but the program would not need to have large table areas set aside to hold file names. This solution to the problem is slightly more complicated, as you can see from the logic in Figure 5-17.

The subroutine in Figure 5-17, Get Next File (GNF), contains all of the necessary logic to search down a directory for both alternatives described. It does require that you indicate on entry whether it should search for the first or next file match, by setting A to zero or some nonzero value respectively.

You can see from Figure 5-17 that whenever the subroutine is called to get the next file, you must execute a Search First function to re-find the previous file. Only then can a Search Next be issued.

As with all functions that return a directory code in A, if this value is not OFFH, it will be the relative directory entry number in the directory record currently in memory. This directory record will have been read into memory at whatever address was specified at the last Set DMA Address function call (code 26, I AH). Notwithstanding its odd name, the DMA Address is simply the address into which any record input from disk will be placed. If the Set DMA Address function has not been used to change the value, then the CP/M default DMA address, location 0080H, will be used to hold the directory record.

The actual code for locating the address of the particular directory entry matched by the Search First/Next functions is shown in Figure 5-17 near the label GNFC. The method involves multiplying the directory code by 32 and then adding this product to the current DMA address.

**Function 19: Erase (Delete) File**

Function Code: C = 13H

**Entry Parameters:**
- DE = Address of file control block
- A = Directory code

**Exit Parameters:**

**Example**

```
0013 = B*ERASE
0005 = BDOS
     EQU 19       ; Erase File
     EQU 5        ; BDOS entry point

FCB:
0000 00 FCB*DISK:  DB  0       ; Search on default disk drive
0001 3F3F4C454EFCB*NAME: DB  '?'   ; Ambiguous file name
0009 3F3950 FCB*TYP:  DB  '?'    ; Ambiguous file type
00C 00 FCB*EXTENT:  DB  0       ; Extent
```
Chapter 5: The Basic Disk Operating System

Purpose

This function logically deletes from the file directory files that match the FCB addressed by DE. It does so by replacing the first byte of each relevant directory entry (remember, a single file can have several entries, one for each extent) by the value 0E5H. This flags the directory entry as being available for use.

Notes

Like the previous two functions, Search First and Search Next, this function can take an ambiguous file name and type as part of the file control block, but unlike those functions, the logical disk select code cannot be a “?”.

This function returns a directory code in A in the same way as the previous file operations.

Function 20: Read Sequential

Function Code: C = 14H
Entry Parameters: DE = Address of file control block
Exit Parameters: A = Directory code

Example

```
0014 = B$READSEQ EQU 20 ;Read Sequential
0005 = BDOS EQU 5 ;BDOS entry point

;File control block
0000 00 FCB$DISK: DB 0 ;Search on default disk drive.
0001 45494C454E FCB$NAME: DB 'FILENAME' ;File name
0009 545950 FCB$TYP: DB 'TYP' ;File type
000C 345550 FCB$SEQREC: DB 24 ;Set by file open

;Record will be read into
; address set by prior SETDMA
; call

0024 0E14 MVI C, B$READSEQ ;Function code
0026 110000 LXI D, FCB ;DE -> File control block
0029 CD0500 CALL BDOS ;A = 00 if operation successful
; A = nonzero if no data in file
```

Purpose

This function reads the next record (128-byte sector) from the designated file into memory at the address set by the last Set DMA function call (code 26, 1AH). The record read is specified by the FCB’s sequential record field (FCB$SEQREC in the example listing for the Open File function, code 15). This field is incremented by 1 so that a subsequent call to Read Sequential will get the next record from the file. If the end of the current extent is reached, then the BDOS will
The CP/M Programmer's Handbook

Figure 5-18. Read next character from sequential disk file

This subroutine gets the next character from a sequential disk file. It assumes that the file has already been opened.

Note: this subroutine changes CP/M’s DMA address.

Entry parameters

- DE → file control block

Exit parameters

- A = next character from file
  - (= OFFH on physical end of file)

Note: IAH is normal EOF character for ASCII Files.

Calling sequence

- LXI DE, FCB
- CALL OETC
- CPI IAH
- JZ EOFCHAR
- CPI OFFH
- JZ ACTUALEOF

0014 = B$READSEQ EQU 20 ; Read sequential
001A = B$SETDMA EQU 26 ; Set DMA address
0005 = BDOS EQU 5 ; BDOS entry point
0080 = OETCBS EQU 128 ; Buffer size
0000 OETCBF: DS OETCBS ; Declare buffer
0080 00 OETCC: DB 0 ; Char. count (initially "empty")

OETC:

0061 3A8000 LDA OETCC ; Check if buffer is empty
0064 B7 ORA A ; Yes, fill buffer
0065 CA9900 JZ OETCFB

GETCRE:

0088 3D DCR A ; No, downdate count
0089 328000 STA OETCC ; Save downdated count
008C 47 MOV B,A ; Compute offset of next character
008D 3E7F MVI A,OETCBS-1 ; By subtracting
008F 90 MOV B,A ; (buffer size -- downdated count)
0090 3F MOV E.A ; Make result into word value
0091 1600 MVI D,0
0092 210000 LXI H,OETCFB
0096 19 DAD D ; HL -> base of buffer
0097 7E MOV A.M ; HL -> next character in buffer
0098 C9 RET ; Get next character

OETCFB:

0099 D5 PUSH D ; Save FCB pointer
009A 110000 LXI D,OETCFB ; Set DMA address to buffer
009D 0E1A MVI C,B$SETDMA ; Function code
009F C00500 CALL BDOS
00A2 D1 POP D ; Recover FCB pointer
00A3 0E14 MVI C,B$READSEQ ; Read sequential "record" (sector)
00A5 C00500 CALL BDOS
00A8 B7 ORA A ; Check if read unsuccessful (A = NZ)
00A9 C28400 JNZ OETC ; Yes
00AC 3E80 MVI A,OETCBS ; Reset count
00AE 328000 STA OETCC
00B1 C38800 JMP OETCRE ; Re-enter subroutine

OETCX:

00B4 3EFF MVI A,OFFH ; Physical end of file
00B6 C9 RET ; Indicate such
automatically open the next extent and reset the sequential record field to 0, ready for the next Read function call.

The file specified in the FCB must have been readied for input by issuing an Open File (code 15, 0FH) or a Create File (code 22, 16H) BDOS call.

The value 00H is returned in A to indicate a successful Read Sequential operation, while a nonzero value shows that the Read could not be completed because there was no data in the next record, as at the end of file.

**Notes**

Although it is not immediately obvious, you can change the sequential record number, FCB$SEQREC, and within a given extent, read a record at random. If you want to access any given record within a file, you must compute which extent that record would be in and set the extent field in the file control block (FCB$EXTENT) before you open the file. Thus, although the function name implies sequential access, in practice you can use it to perform a simple type of random access. If you need to do true random access, look ahead to the Random Read function (code 33), which takes care of opening the correct extent automatically.

Figure 5-18 shows an example of a subroutine that returns the data from a sequential file byte-by-byte, reading in records from the file as necessary. This subroutine, GETC, is useful as a low-level "primitive" on which you can build more sophisticated functions, such as those that read a fixed number of characters or read characters up to a CARRIAGE RETURN/LINE FEED combination.

When you read data from a CP/M text file, the normal convention is to fill the last record of the file with 1AH characters (CONTROL-Z). Therefore, two possible conditions can indicate end-of-file: either encountering a 1AH, or receiving a return code from the BDOS function (in the A register) of OFFH. However, if the file that you are reading is not an ASCII text file, then a 1AH character has no special meaning—it is just a normal data byte in the body of the file.

**Function 21: Write Sequential**

Function Code: C = 15H
Entry Parameters: DE = Address of file control block
Exit Parameters: A = Directory code

**Example**

```
0015 = B$WRITSEQ EQU 21 ;Write Sequential
0005 = BDOS EQU 5 ;BDOS entry point

FCB:  
0000 00 FCB$DISK: DB 0 ;File control block
0001 46494C454EFCB$NAME: DB 'FILENAME' ;Search on default disk drive
0009 545950 FCB$TYP: DB 'TYP' ;File name
000C 545950 FCB$TYP: DS 24 ;Set by Open or Create File

0024 0E15 MVI C,B$WRITSEQ ;Record must be in address
0026 110000 LXI D,FCB ;set by prior SETDMA call
0029 CD0500 CALL BDOS ;Function code
```

1DE -> File control block  
;A = 00H if operation  
;successful  
;A = nonzero if disk full
Purpose

This function writes a record from the address specified in the last Set DMA (code 26, 1AH) function call to the file defined in the FCB. The sequential record number in the FCB (FCB$SEQREC) is updated by 1 so that the next call to Write Sequential will write to the next record position in the file. If necessary, a new extent will be opened to receive the new record.

This function is directly analogous to the Read Sequential function, writing instead of reading. The file specified in the FCB must first be activated by an Open File (code 15, 0FH) or create File call (code 22, 16H).

A directory code of 00H is returned in A to indicate that the Write was successful; a nonzero value is returned if the Write could not be completed because the disk was full.

Notes

As with the Read Sequential function (code 20, 14H), you can achieve a simple form of random writing to the file by manipulating the sequential record number (FCB$SEQREC). However, you can only overwrite existing records in the file, and if you want to move to another extent, you must close the file and reopen it with the FCB$EXTENT field set to the correct value. For true random writing to the file, look ahead to the Write Random function (code 34, 22H). This takes care of opening or creating the correct extent of the file automatically.

The only logical error condition that can occur when writing to a file is insufficient room on the disk to accommodate the next extent of the file. Any hardware errors detected will be handled by the disk driver built into the BIOS or BDOS.

Figure 5-19 shows a subroutine, PUTC, to which you can pass data a byte at a time. It assembles this data into a buffer, making a call to Write Sequential whenever the buffer becomes full. You can see that provision is made in the entry parameters (by setting register B to a nonzero value) for the subroutine to fill the remaining unused characters of the buffer with 1AH characters. You must do this to denote the end of an ASCII text file.

Function 22: Create (Make) File

Function Code: C = 16H
Entry Parameters: DE = Address of file control block
Exit Parameters: A = Directory code

Example

```
0016  =  B$CREATE    EQU  22    ;File Create
0005  =  BDOS        EQU  5      ;BDOS entry point

FCB:
0000  00  FCB$DISK:   DB  0      ;Search on default disk drive
0001  46494C454E    DB  'FILENAME'; file name
0002  545550  FCB$TYP:  DB  'TYP'    ;file type
0003  00  FCB$EXTENT: DB  0      ;Extent
```
Chapter 5: The Basic Disk Operating System

000D 0000  FCB$RESV:  DB 0,0 ;Reserved for CP/M
000F 00  FCB$REUSED:  DB 0 ;Records used in this extent
0010 0000000000FCB$ABUSED:  DB 0,0,0,0,0,0,0,0 ;Allocation blocks used
0018 0000000000DB 0,0,0,0,0,0,0,0 ;Sequential rec. to read/write
0020 00  FCB$SEQREC:  DB 0 ;Random rec. to read/write
0021 0000  FCB$RANREC:  DW 0 ;Random rec. overflow byte (MS)
0024OE16  MVI C,B$CREATE ;Note: file to be created
0026 110000  LXI D,FCB ;must not already exist....
0029 CD0500  CALL BDOS ;Function code
          ;DE -> file control block
          ;A = 0,1,2,3 if operation
          ;successful
          ;A = OFFH if directory full

;PUTC
;This subroutine either puts the next character out
;to a sequential file, writing out completed "records"
;(128-byte sectors) or, if requested to, will fill the
;remainder of the current "record" with 1AH's to
;indicate end of file to CP/M.

;Entry parameters
; DE -> File control block
; B = 0, A = next data character to be output
; B /= 0, fill the current "record" with 1AH's

;Exit parameters
; none.

;Calling sequence
; LXI D,FCB
; MVI B,0 ;Not end of file
; LDA CHAR
; CALL PUTC
; or
; LXI D,FCB
; MVI B,1 ;Indicate end of file
; CALL PUTC

0015 = BWRITESEQ EQU 21 ;Write sequential
001A = BSETDMA EQU 26 ;Set DMA address
005E = BDOS EQU 5 ;BDOS entry point
0080 = PUTCBS EQU 128 ;Buffer size
0000 = PUTCBF DS PUTCBS ;Declare buffer
0090 00 PUTCCF DB 0 ;Char. count (initially "empty")

PUTC:
0081 DS PUSH D ;Save FCB address
0082 FS PUSH PSW ;Save data character
0083 78 MOV A,B ;Check if end of file requested
0084 87 ORA A
0085 C99000 JNZ PUTCEF ;Yes
0088 CDC300 CALL PUTCEA ;No; get address of next free byte
008B 00 STA PUTCCC ;HL -> next free byte
008D 78 MOV A,E ;Current char. count (as well as A)
008E 3C INR A ;Update character count
008F FE80 CPI PUTCBS ;Check if buffer full
0091 CAA400 JZ PUTCBF ;Yes, write buffer
0094 328000 STA PUTCCC ;No, save updated count
0097 D1 POP D ;Dump FCB address for return
0099 C9 RET

Figure 5-19. Write next character to sequential disk file
### Purpose

This function creates a new file of the specified name and type. You must first ensure that no file of the same name and type already exists on the same logical disk, either by trying to open the file (if this succeeds, the file already exists) or by unconditionally erasing the file.

In addition to creating the file and its associated file directory entry, this function also effectively opens the file so that it is ready for records to be written to it.

This function returns a normal directory code if the file creation has completed successfully or a value of 0FFH if there is insufficient disk or directory space.

### Notes

Under some circumstances, you may want to create a file that is slightly more "secure" than normal CP/M files. You can do this by using either lowercase letters or nongraphic ASCII characters such as ASCII NUL (00H) in the file name or type. Neither of these classes of characters can be generated from the keyboard; in the first case, the CCP changes all lowercase characters to uppercase, and in the second, it rejects names with odd characters in them. Thus, computer operators...
cannot erase such a file because there is no way that they can create the same file name from the CCP.

The converse is also true; the only way that you can erase these files is by using a program that can set the exact file name into an FCB and then issue an Erase File function call.

Note that this function cannot accept an ambiguous file name in the FCB.

Figure 5-20 shows a subroutine that creates a file only after it has erased any existing files of the same name.

**Function 23: Rename File**

Function Code: \( C = 17H \)

Entry Parameters: \( DE = \) Address of file control block

Exit Parameters: \( A = \) Directory code

**Example**

```
0017 = B$RENAME EQU 23 ; Rename file
0005 = BDOS EQU 5 ; BDOS entry point

FCB:
  0000 00 DB 0 ; Search on default disk drive
  0001 4F4C444E41 DB 'OLDNAME' ; File name
  0009 544553 DB 'TVP' ; File type
  000C 00000000 DB 0,0,0,0
```

; CF
; Create file
; This subroutine creates a file. It erases any previous file before creating the new one.

; Entry parameters
; \( DE \rightarrow \) File control block for new file

; Exit parameters
; Carry clear if operation successful
; \( A = 0,1,2,3 \)
; Carry set if error \( (A = OFFH) \)

; Calling sequence
; LXI D, FCB
; CALL CF
; JC ERROR

```
0013 = B$ERASE EQU 19 ; Erase file
0016 = B$CREATE EQU 22 ; Create file
0005 = BDOS EQU 5 ; BDOS entry point

CF:
0000 D5 PUSH D ; Preserve FCB pointer
0001 OE13 MVI C, B$ERASE ; Erase any existing file
0003 CD0500 CALL BDOS ; Recover FCB pointer
0006 D1 POP D ; Create (and open new file)
0007 OE16 MVI C, B$CREATE
0009 CD0500 CALL BDOS
000C FEFF CPI OFFH ; Carry set if OK, clear if error
000E 3F RET ; Complete to use Carry set if Error
```

**Figure 5-20.** Create file request
This function renames an existing file name and type to a new name and type. It is unusual in that it uses a single FCB to store both the old file name and type (in the first 16 bytes) and the new file name and type (in the second 16 bytes).

This function returns a normal directory code if the file rename was completed successfully or a value of OFFH if the old file name could not be found.

Notes

The Rename File function only checks that the old file name and type exist; it makes no check to ensure that the new name and type combination does not already exist. Therefore, you should try to open the new file name and type. If you succeed, do not attempt the rename operation. CP/M will create more than one file of the same name and type, and you stand to lose the information in both files as you attempt to sort out the problem.

For security, you can also use lowercase letters and nongraphic characters in the file name and type, as described under the File Create function (code 22, 16H) above.

Never use ambiguous file names in a rename operation; it produces strange effects and may result in files being irreparably damaged. This function will change all occurrences of the old file name to the new name.

Figure 5-21 shows a subroutine that will accept an existing file name and type and a new name and type and rename the old to the new. It checks to make sure that the new file name does not already exist, returning an error code if it does.
Chapter 5: The Basic Disk Operating System

Chapter 5: The Basic Disk Operating System

Rename file

This subroutine renames a file.
It uses the BF (build FCB) subroutine shown in Figure 5.16

Entry parameters

No case-folding of file names occurs new
HL -> old file name (00-byte terminated)
DE -> new file name (00-byte terminated)

Exit parameters

Carry clear if operation successful
A = 0,1,2,3
Carry set if error
A = OFEH if new file name already exists
A = OFFH if old file name does not exist

Calling sequence

LXI M.OLONAME
LXI Il.NEWNAME
CALL RF
JC ERROR

000F = BSOPEN EQU 15 Open file
0017 = BSRENAME EQU 23 Rename file
0005 = BDOS EQU 5 BDOS entry point

0000 0000000000RFFCB: DW 0,0,0,0,0,0,0,0,0,0,0,0,1/2 FCB's long
0010 0000000000 DW 0,0,0,0,0,0,0,0,0,0,0,0
0020 0000000000 DW 0,0,0,0,0,0,0,0,0,0,0,0
0030 0000000000 DW 0,0,0,0,0,0,0,0,0,0,0,0

RF:

0036 D5 PUSH D Save new name pointer
0037 110000 LXI D,RFFCB Build old name FCB
003A CD5D00 CALL BF HL already -> old name
003D E1 POP H Recover new name pointer
003E 111000 LXI D,RFFCB+16 Build new name in second part of file control block
0041 CD5D00 CALL BF
0044 111000 LXI D,RFFCB+16 Experimentally try
0047 0E0F MVI C,BOPEN Try open the new file
0049 CD0500 CALL BDOS To ensure it does not already exist
004C FEFF CPI OFFH If error (flags unchanged)
004E 3EFF MVI A,OFEH Assume error
0050 DB RC Carry set if A was 0,1,2,3

0051 110000 LXI D,RFFCB Rename the file
0054 0E17 MVI C,BSRENAME
0056 CD0500 CALL BDOS
0059 FEFF CPI OFFH Carry set if OK, clear if error
005B 3F CMC Invert to use carry, set if error
005C C9 RET

BF:

Build file control block
This subroutine formats a 00-byte terminated string (presumed to be a file name) into an FCB, setting the disk and the file name and type, and clearing the remainder of the FCB to 0's.

Entry parameters

DE -> file control block (36 bytes)
HL -> file name string (00-byte terminated)

Exit parameters

The built file control block.

Calling sequence

LXI D,FCB
LXI H,FNAME
CALL BF

BF:

RET ;Dummy subroutine : see Figure 5.16.

Figure 5-21. Rename file request
Reset Disk function (code 13, 0DH). The least significant bit of L corresponds to
disk A, while the highest order bit in H maps disk P. The bit corresponding to the
specific logical disk is set to 1 if the disk has been selected or to 0 if the disk is not
currently on-line.

Logical disks can be selected programmatically through any file operation
that sets the drive field to a nonzero value, through the Select Disk function (code
14, 0EH), or by the operator entering an “X:” command where “X” is equal to A,
B,..., P.

Notes
This function is intended for programs that need to know which logical disks
are currently active in the system—that is, those logical disks which have been
selected.

Function 25: Get Current Default Disk

Function Code: C = 19H
Entry Parameters: None
Exit Parameters: A = Current disk
(0 = A, 1 = B,..., F = P)

Example

0019 = B*GETCURDSK
0005 = BDOS
0000 0E19
0002 CD0300
MVI C,B*GETCURDSK
CALL BDOS

Purpose
This function returns the current default disk set by the last Select Disk
function call (code 14, 0EH) or by the operator entering the “X:” command (where
“X” is A, B,..., P) to the CCP.

Notes
This function returns the current default disk in coded form. Register A = 0 if
drive A is the current drive, 1 if drive B, and so on. If you need to convert this to the
corresponding ASCII character, simply add 41H to register A.

Use this function when you convert a file name and type in an FCB to an
ASCII string in order to display it. If the first byte of the FCB is 00H, the current
default drive is to be used. You must therefore use this function to determine the
logical disk letter for the default drive.

Function 26: Set DMA (Read/Write) Address

Function Code: C = 1AH
Entry Parameters: DE = DMA (read/write) address
Exit Parameters: None

Example

001A = B*SETDMA
0005 = BDOS
0000 0E19
0002 CD0500
MVI C,B*SETDMA
CALL BDOS

Purpose
This function returns the current default disk in coded form. Register A = 0 if
drive A is the current drive, 1 if drive B, and so on. If you need to convert this to the
corresponding ASCII character, simply add 41H to register A.

Use this function when you convert a file name and type in an FCB to an
ASCII string in order to display it. If the first byte of the FCB is 00H, the current
default drive is to be used. You must therefore use this function to determine the
logical disk letter for the default drive.
Chapter 5: The Basic Disk Operating System

Purpose

This function sets the BDOS's direct memory access (DMA) address to a new value. The name is an historic relic dating back to the Intel Development System on which CP/M was originally developed. This machine, by virtue of its hardware, could read data from a diskette directly into memory or write data to a diskette directly from memory. The name DMA address now applies to the address of the buffer to and from which data is transferred whenever a diskette Read, Write, or directory operation is performed.

Whenever CP/M first starts up (cold boot) or a warm boot or Reset Disk operation occurs, the DMA address is reset to its default value of 0080H.

Notes

No function call can tell you the current value of the DMA address. All you can do is make a Set DMA function call to ensure that it is where you want it.

Once you have set the DMA address to the correct place for your program, it will remain set there until another Set DMA call, Reset Disk, or warm boot occurs.

The Read and Write Sequential and Random operations use the current setting of the DMA address, as do the directory operations Search First and Search Next.

Function 27: Get Allocation Vector

Function Code: C = 1BH
Entry Parameters: None
Exit Parameters: HL = Address of allocation vector

Example

```
001B = B.GETALVEC EQU 27 ;Get Allocation Vector Address
0005 = BDOS EQU 5 ;BDOS entry point
0000 0E1B MVI C,B.GETALVEC ;Function code
0002 CD0500 CALL BDOS ;HL -> Base address of allocation vector
```

Purpose

This function returns the base, or starting, address of the allocation vector for the currently selected logical disk. This information, indicating which parts of the disk are assigned, is used by utility programs and the BDOS itself to determine how much unused space is on the logical disk, to locate an unused allocation block in order to extend a file, or to relinquish an allocation block when a file is deleted.

Notes

Digital Research considers the actual layout of the allocation vector to be proprietary information.
Function 28: Set Logical Disk to Read-Only Status

Function Code: \( C = 1CH \)
Entry Parameters: None
Exit Parameters: None

Example

```
001C = BSETDSKRO EQU 28 ; Set disk to Read Only
0005 = BDOS EQU 5 ; BDOS entry point
0000 OEIC MVI C.BSETDSKRO
0002 CD0500 CALL BDOS
```

Purpose
This function logically sets the currently selected disk to a Read-Only state. Any attempts to execute a Write Sequential or Write Random function to the selected disk will be intercepted by the BDOS, and the following message will appear on the console:

\[ \text{BDOS Err on X: R/O} \]

where X: is the selected disk.

Notes
Once you have requested Read-Only status for the currently selected logical disk, this status will persist even if you proceed to select other logical disks. In fact, it will remain in force until the next warm boot or Reset Disk System function call.

Digital Research documentation refers to this function code as Disk Write Protect. The Read-Only description is used here because it corresponds to the error message produced if your program attempts to write on the disk.

Function 29: Get Read-Only Disks

Function Code: \( C = 1DH \)
Entry Parameters: None
Exit Parameters: \( HL = \) Read-Only disk map

Example

```
001D = BGETRODSKS EQU 29 ; Get Read Only disks
0005 = BDOS EQU 5 ; BDOS entry point
0000 OE19 MVI C.BGETRODSKS
0002 CD0500 CALL BDOS
```

Purpose
This function returns a bit map in registers H and L showing which logical disks in the system have been set to Read-Only status, either by the Set Logical
Disk to Read-Only function call (code 28, 1CH), or by the BDOS itself, because it detected that a diskette had been changed.

The least significant bit of L corresponds to logical disk A, while the most significant bit of H corresponds to disk P. The bit corresponding to the specific logical disk is set to 1 if the disk has been set to Read-Only status.

**Function 30: Set File Attributes**

- **Function Code:** C = 1EH
- **Entry Parameters:** DE = Address of FCB
- **Exit Parameters:** A = Directory code

**Example**

```
001E  B@SETFAT EQU 30 ;Set File Attribute
0005  BDOS EQU 5 ;BDOS entry point

0000 00 FCB@DISK: DB 0 ;File control block
0001 46494C4544FCB@NAME: DB 'FILENAME' ;File name
0009  D4 FCB@TYP: DB 'T'+80H ;Type with R/O attribute

000A 5950 DB 'YP'
000C 0000000000 DW 0,0,0,0,0,0,0,0,0,0,0,0

0022 0E1E MVI C,B@SETFAT ;Function code
0024 110000 LXI D,FCB ;DE -> file control block
0027 CD0500 CALL BDOS ;MS bits set in file name/type

0A = OFFH if file not found
```

**Purpose**

This function sets the bits that describe attributes of a file in the relevant directory entries for the specified file. Each file can be assigned up to 11 file attributes. Of these 11, two have predefined meanings, four others are available for you to use, and the remaining five are reserved for future use by CP/M.

Each attribute consists of a single bit. The most significant bit of each byte of the file name and type is used to store the attributes. The file attributes are known by a code consisting of the letter “f” (for file name) or “t” (for file type), followed by the number of the character position and a single quotation mark. For example, the Read-Only attribute is f1'.

The significance of the attributes is as follows:

- f1' to f4' Available for you to use
- f5' to f8' Reserved for future CP/M use
- t1' Read-Only File attribute
- t2' System File attribute
- t3' Reserved for future CP/M use

Attributes are set by presenting this function with an FCB in which the unambiguous file name has been preset with the most significant bits set appropriately. This function then searches the directory for a match and changes the matched entries to contain the attributes which have been set in the FCB.
The BDOS will intercept any attempt to write on a file that has the Read-Only attribute set. The DIR command in the CCP does not display any file with System status.

Notes

You can use the four attributes available to you to set up a file security system, or perhaps to flag certain files that must be backed up to other disks. The Search First and Search Next functions allow you to view the complete file directory entry, so your programs can test the attributes easily.

The example subroutines in Figures 5-22 and 5-23 show how to set file attributes (SFA) and get file attributes (GFA), respectively. They both use a bit map in which the most significant 11 bits of the HL register pair are used to indicate the corresponding high bits of the 11 characters of the file name/type combination. You will also see some equates that have been declared to make it easier to manipulate the attributes in this bit map.

```
:SFA
;Set file attributes
;This subroutine takes a compressed bit map of all the
;file attribute bits, expands them into an existing
;file control block and then requests CP/M to set
;the attributes in the file directory.

;Entry parameters
;  DE -> file control block
;  HL = bit map. Only the most significant 11
;  bits are used. These correspond directly
;  with the possible attribute bytes.

;Exit parameters
;  Carry clear if operation successful (A = 0, 1, 2, 3)
;  Carry set if error (A = OFFH)

;Calling sequence
;  LXI D, FCB
;  LXI H, $0000$0000$1100$0000
;  CALL SFA
;  JC ERROR

;File Attribute Equates
8000 = FA$F1 EQU 1000$0000$0000$0000
4000 = FA$F2 EQU 0100$0000$0000$0000 ;Available for use by
2000 = FA$F3 EQU 0010$0000$0000$0000 ;application programs
1000 = FA$F4 EQU 0001$0000$0000$0000
0800 = FA$F5 EQU 0000$0100$0000$0000 ;Reserved for CP/M
4000 = FA$F6 EQU 0000$0010$0000$0000
2000 = FA$F7 EQU 0000$0001$0000$0000
0200 = FA$F8 EQU 0000$0000$1000$0000
0080 = FA$T1 EQU 0000$0000$0100$0000 ;T1' -- read/only file
0080 = FA$T0 EQU FA$T1
0040 = FA$T2 EQU 0000$0000$0100$0000 ;T2' -- system files
0040 = FA$SYS EQU FA$T2
0020 = FA$T3 EQU 0000$0000$0010$0000 ;T3' -- reserved for CP/M
001E = $SETFAT EQU 30 ;Set file attributes
0005 = BDOS EQU 5 ;BDOS entry point
```

Figure 5-22. Set file attributes
Figure 5-22. Set file attributes (continued)

;GFA
;Get file attributes
;This subroutine finds the appropriate file using a
;search for First Name Match function rather than opening
;the file. It then builds a bit map of the file attribute
;bits in the file name and type. This bit map is then ANOed
;with the input bit map, and the result is returned in the
;zero flag. The actual bit map built is also returned in case
;more complex check is required.

;>>> Note: This subroutine changes the CP/M DMA address.

;Entry parameters
;    DE -> File control block
;    HL = Bit map mask to be ANDed with attribute
;          results

;Exit parameters
;    Carry clear, operation successful
;    Nonzero status set to result of AND between
;    input mask and attribute bits set.
;    HL = Unmasked attribute bytes set.
;    Carry set, file could not be found

0000 = B$SETDMA EQU 26 ;Set DMA address
0011 = B$SEARCHF EQU 17 ;Search for first entry to match
0005 = BDOS EQU 5 ;Default DMA address
0080 = EQU 80H ;BDOS entry point

;Calling sequence
;    LXI D, FCB
;    LXI H, 0000$0000$0000$0000B ;Bit map
;    CALL GFA
;    JC ERROR

;File attribute equates
8000 = FASF1 EQU 1000$0000$0000$0000B ;F1' = F5'
4000 = FASF2 EQU 0100$0000$0000$0000B ;Available for use by

Figure 5-23. Get file attributes
Figure 5-23. Get file attributes (continued)

```
2000 = FAST3 EQU 0010#0000#0000#0000B : Application programs
1000 = FAST4 EQU 0001#0000#0000#0000B
0800 = FAST5 EQU 0000#1000#0000#0000B : F6' - F8'
0400 = FAST6 EQU 0000#0100#0000#0000B : Reserved for CP/M
0200 = FAST7 EQU 0000#0010#0000#0000B
0100 = FAST8 EQU 0000#0001#0000#0000B
0080 = FAST9 EQU 0000#0000#1000#0000B : T1' - read/only file
0080 = FAST10 EQU FAST1
0040 = FAST12 EQU 0000#0000#0100#0000B : T2' - system files
0040 = FAST14 EQU FAST2
0020 = FAST13 EQU 0000#0000#0010#0000B : T3' - reserved for CP/M

0000 E5 PUSH H ; Save AND-mask
0001 D5 PUSH D ; Save FCB pointer
0002 0E1A MVI C, B#SETDMA ; Set DMA to default address
0004 0004 LXI D, OFADMA ; DE -> DMA address
0007 CD5000 CALL BDOS

000A D1 POP D ; Recover FCB pointer
000B 0E11 MVI C, B#SEARCHF ; Search for match with name
000D CD5000 CALL BDOS
0010 0EFF CPI OFFH ; Carry set if OK, clear if error
0012 5F CMP ; Invert to use set carry if error
0013 DA4100 JC OFA ; Return if error

0016 87 ADD A ; Multiply by 32 to get offset into DMA buffer
0017 87 ADD A
0018 87 ADD A
0019 87 ADD A
001A 87 ADD A
001B 5F MOV E, A ; Make into a word value
001C 1600 MVI D, 0 ; HL -> DMA address
001E 210000 LXI H, OFADMA
0021 19 DAD D ; HL -> Directory entry in DMA buffer
0022 23 INX H ; HL -> 1st character of file name
0023 EB XCHG ; DE -> 1st character of file name
0024 0E0B MVI C, B#3 ; Count of characters in file name and type
0026 210000 LXI H, 0 ; Clear bit map

0029 1A LDAX D ; Main loop
002A C680 ANI 80H ; Get next character of file name
002C 07 RLC ; Isolate attribute bit
002D 85 ORA L ; Move MS bit into LS bit
002E 5F MOV L, A ; OR in any previously set bits
002F 29 DAD H ; Save result
0030 13 INX D ; Shift HL left one bit for next time
0031 0D DCR C ; Downdate count
0032 C2900 JNZ OFA ; Do back for next character
0035 29 DAD H ; Left justify attribute bits in HL
0036 29 DAD H ; MS attribute bit will already be in
0037 29 DAD H ; bit 11 of HL, so only 4 shifts are
0038 29 DAD H ; necessary
0039 D1 POP D ; Recover AND-mask
003A 7A MOV A, D ; Get MS byte of mask
003B 44 ANA H ; AND with MS byte of result
003C 47 MOV B, A ; Save interim result
003D 7B MOV A, E ; Get LS byte of mask
003E A5 ANA L ; AND with LS byte of result
003F 80 ORA B ; Combine two results to set Z flag
0040 C9 RET ; Error exit
0041 E1 POP H ; Balance stack
0042 C9 RET
```

Chapter 5: The Basic Disk Operating System

Function 31: Get Disk Parameter Block Address

Function Code: \( C = 1FH \)

Entry Parameters: None

Exit Parameters: HL = Address of DPB

**Example**

```
001F = B$GETDPB EQU 31 ;Get Disk Parameter Block
0005 = BDOS EQU 5 ;BDOS entry point

0000 OE1F MVI C, B$GETDPB ;Function code
0002 CD0500 CALL BDOS ;Returns DPB address of logical disk previously selected with a Select Disk function.

Purpose

This function returns the address of the disk parameter block (DPB) for the last selected logical disk. The DPB, explained in Chapter 3, describes the physical characteristics of a specific logical disk—information mainly of interest for system utility programs.

Notes

The subroutines shown in Figure 5-24 deal with two major problems. First, given a track and sector number, what allocation block will they fall into? Conversely, given an allocation block, what is its starting track and sector?

These subroutines are normally used by system utilities. They first get the DPB address using this BDOS function. Then they switch to using direct BIOS calls to perform their other functions, such as selecting disks, tracks, and sectors and reading and writing the disk.

The first subroutine, GTAS (Get Track and Sector), in Figure 5-24, takes an allocation block number and converts it to give you the starting track and sector number. GMTAS (Get Maximum Track and Sector) returns the maximum track and sector number for the specified disk. GDTAS (Get Directory Track and Sector) tells you not only the starting track and sector for the file directory, but also the number of 128-byte sectors in the directory.

Note that whenever a track number is used as an entry or an exit parameter, it is an absolute track number. That is, the number of reserved tracks on the disk before the directory has already been added to it.

GNTAS (Get Next Track and Sector) helps you read sectors sequentially. It adds 1 to the sector number, and when you reach the end of a track, updates the track number by 1 and resets the sector number to 1.

GAB (Get Allocation Block) is the converse of GTAS (Get Track and Sector). It returns the allocation block number, given a track and sector.

Finally, Figure 5-24 includes several useful 16-bit subroutines to divide the HL register pair by DE (DIVHL), to multiply HL by DE (MULHL), to subtract DE from HL (SUBHL — this can also be used as a 16-bit compare), and to shift HL right one bit (SHLR). The divide and multiply subroutines are somewhat primitive, using iterative subtraction and addition, respectively. Nevertheless, they do perform their role as supporting subroutines.
Useful subroutines for accessing the data in the disk parameter block

000E = B$SELDSK EQU 14 ;Select Disk function code
001F = B$GETDPB EQU 31 ;Get DPB address
0005 = BDOS EQU 5 ;BDOS entry point

It makes for easier, more compact code to copy the specific disk parameter block into local variables while manipulating the information.

Here are those variables --

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DPB:</td>
<td>Disk parameter block</td>
</tr>
<tr>
<td>DPBSPRT:</td>
<td>DW 0 128-byte sectors per track</td>
</tr>
<tr>
<td>DPBSS:</td>
<td>DB 0 Block shift</td>
</tr>
<tr>
<td>DPBEM:</td>
<td>DB 0 Block mask</td>
</tr>
<tr>
<td>DPBEM:</td>
<td>DB 0 Extent mask</td>
</tr>
<tr>
<td>DPBMB:</td>
<td>DW 0 Maximum allocation block number</td>
</tr>
<tr>
<td>DPBMD:</td>
<td>DW 0 Number of directory entries - 1</td>
</tr>
<tr>
<td>DPBDB:</td>
<td>DW 0 Directory allocation block number</td>
</tr>
<tr>
<td>DPBCS:</td>
<td>DW 0 Check buffer size</td>
</tr>
<tr>
<td>DPBTBD:</td>
<td>DW 0 Tracks before directory (reserved tracks)</td>
</tr>
<tr>
<td>OPBSZ</td>
<td>EQU $-OPB Disk parameter block size</td>
</tr>
</tbody>
</table>

; GETDPB
; Gets disk parameter block
; This subroutine copies the DPB for the specified logical disk into the local DPB variables above.

; Entry parameters
; A = Logical disk number (A: = 0, B: = 1...)

; Exit parameters
; Local variables contain DPB

GETDPB:

0006 0000 MOV E,A ;Get disk code for select disk
0010 008E MVI C,B$SELDSK ;Select the disk
0012 CD050D CALL BDOS ;Get disk parameter base address
0015 001F MVI C,B$GETDPB
0017 CD050D CALL BDOS ;HL -> DPB
001A 00E0 MVI D,OPB ;Set count
001C 110000 LXI D,OPB ;Get base address of local variables

GETDPB:

000F 5F MOV E,A ;Get disk code for select disk
0010 0E0E MVI C,B$SELDSK ;Select the disk
0012 CD050D CALL BDOS ;Get disk parameter base address
0015 0E1F MVI C,B$GETDPB
0017 CD050D CALL BDOS ;HL -> DPB
001A 00E0 MVI D,OPB ;Set count
001C 110000 LXI D,OPB ;Get base address of local variables

GETDPB:

001F 7E MOV A,M ;Copy DPB into local variables
0020 12 STAX D ;Get byte from DPB
0021 10 INX D ;Store into local variable
0022 23 INX H ;Update local variable pointer
0023 0D DCR C ;Downdate count
0024 C2 F00 JNZ GDPBL ;Loop back for next byte
0027 0F RET

; GTAS
; Get track and sector (given allocation block number)

; This subroutine converts an allocation block into a track and sector number -- note that this is based on 128-byte sectors.

; Note: You must call GETDPB before you call this subroutine

; Entry parameters
; HL = allocation block number

; Exit parameters
; HL = track number
; DE = sector number

; Method:
; In mathematical terms, the track can be derived from:
; Trk = ((allocation block * sec. per all. block) / sec. per trk)
; + tracks before directory

Figure 5-24. Accessing disk parameter block data
Chapter 5: The Basic Disk Operating System

Figure 5-24. (Continued)
GNTAS:

> This subroutine updates the input track and sector by one, incrementing the track and resetting the sector number as required.

Note: You must call GETDPB before you call this subroutine.

Notes: you must check for end of disk by comparing the track number returned by this subroutine to that returned by OMTAS + 1. When equality occurs, the end of disk has been reached.

Entry parameters:
- HL = current track number
- DE = current sector number

Exit parameters:
- HL = updated track number
- DE = updated sector number

ONTAS:

| 0056 E5 | PUSH H          | ;Save track
| 0059 13 | INX D          | ;Update sector
| 005A 2A0000 | LHLD DPBSPT | ;Get sectors per track
| 005D CDC900 | CALL SUBHL | ;HL = HL - DE
| 0060 E1 | POP H          | ;Recall current track
| 0061 D0 | PNC H          | ;Return if updated sector < sec. per trk.
| 0062 23 | INX H          | ;Update track if upd. sec > sec. per trk.
| 0063 110100 | LXI D,1 | ;Reset sector to 1
| 0066 C9 | RET            | |

GAB:

> This subroutine returns an allocation block number given a specific track and sector. It also returns the offset down the allocation block at which the sector will be found. This offset is in units of 128-byte sectors.

Note: You must call GETDPB before you call this subroutine.

Entry parameters:
- HL = track number
- DE = sector number

Exit parameters:
- HL = allocation block number

Method:
The allocation block is formed from:
- \( AB = (\text{sector} + (\text{track} - \text{tracks before directory}) \times \text{sectors per track}) / \log_2(\text{sectors per all. block}) \)

The sector offset within allocation block is formed from:
- \( \text{Offset} = (\text{sector} + (\text{tracks before directory}) \times \text{sectors per track}) / \text{sectors per all. block} - 1 \)

GAB:

| 0067 DS | PUSH D          | ;Save sector
| 0068 EB | XCHG            | ;DE = track
| 0069 2A0000 | LHLD DPBTBD | ;Get no. of tracks before directory
| 006C EB | XCHG            | ;DE = no. of tracks before dir. HL = track
| 006D CDC900 | CALL SUBHL | ;HL = HL - DE
| 0070 EB | XCHG            | ;DE = relative track
| 0071 2A0000 | LHLD DPBSPT | ;Get sectors per track
| 0074 CDA400 | CALL MULHL | ;HL = HL \* DE
| 0077 EB | XCHG            | ;DE = number of sectors

Figure 5-24. (Continued)
### Chapter 5: The Basic Disk Operating System

#### Figure 5-24. (Continued)

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>0078 E1</td>
<td>POP H</td>
<td>Recover sector</td>
</tr>
<tr>
<td>0079 2B</td>
<td>DCX H</td>
<td>Make relative to 0</td>
</tr>
<tr>
<td>007A 19</td>
<td>DAD D</td>
<td>HL = relative sector</td>
</tr>
<tr>
<td>007B 3AC000</td>
<td>LDA DPBBM</td>
<td>Get block mask</td>
</tr>
<tr>
<td>007C 47</td>
<td>MOV B,A</td>
<td>Ready for AND operation</td>
</tr>
<tr>
<td>007D 7D</td>
<td>MOV A.L</td>
<td>Get LS byte of relative sector</td>
</tr>
<tr>
<td>0080 4D</td>
<td>ANA B</td>
<td>AND with block mask</td>
</tr>
<tr>
<td>0081 F5</td>
<td>PUSH PSW</td>
<td>A = sector displacement</td>
</tr>
<tr>
<td>0082 3A0200</td>
<td>LDA DPBBS</td>
<td>Get block shift</td>
</tr>
<tr>
<td>0083 4F</td>
<td>MOV C,A</td>
<td>Make into counter</td>
</tr>
</tbody>
</table>

OABS:       | Shift loop |
           | HL shifted right (divided by 2) |
0086 CDDD00 | CALL SHLR  | Shift again if necessary |
0089 0D  | DCR C       | Recover offset |
008A C28600 | POP PSW    | |
008B F1 | DAD D       | |
008C C9 | RET         | |

### DIVHL
Divides HL by DE using an iterative subtract.

#### Entry parameters
- HL = dividend
- DE = divisor

#### Exit parameters
- BC = quotient
- HL = remainder

### DIVHLS:
Save divisor

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>008D D5</td>
<td>PUSH D</td>
<td>Save divisor</td>
</tr>
<tr>
<td>0090 7B</td>
<td>MOV A,E</td>
<td>Note: 2's complement is formed by</td>
</tr>
<tr>
<td>0091 2F</td>
<td>MOV E,A</td>
<td>Inverting all bits and adding 1.</td>
</tr>
<tr>
<td>0092 5F</td>
<td>MOV A,D</td>
<td>Complement divisor (for iterative</td>
</tr>
<tr>
<td>0093 7A</td>
<td>MOV D,A</td>
<td>ADD later on)</td>
</tr>
<tr>
<td>0094 2F</td>
<td>MOV A,E</td>
<td>Get MS byte</td>
</tr>
<tr>
<td>0095 57</td>
<td>MOV D,A</td>
<td>Complement it</td>
</tr>
<tr>
<td>0096 13</td>
<td>INX D</td>
<td></td>
</tr>
</tbody>
</table>

### DIVHLS:
Make 2's complement

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>0097 010000</td>
<td>LXI B,0</td>
<td>Make 2's complement</td>
</tr>
<tr>
<td>009A 03</td>
<td>INX B</td>
<td>Now, subtract negative divisor until</td>
</tr>
<tr>
<td>009B 19</td>
<td>DAD D</td>
<td>dividend goes negative, counting the number</td>
</tr>
<tr>
<td>009C DA9A00</td>
<td>JC DIVHLS</td>
<td>of times the subtract occurs</td>
</tr>
<tr>
<td>009F 0B</td>
<td>DCX B</td>
<td>Initialize quotient</td>
</tr>
<tr>
<td>00A0 EB</td>
<td>XCHG</td>
<td>Subtract loop</td>
</tr>
<tr>
<td>00A1 E1</td>
<td>POP H</td>
<td></td>
</tr>
<tr>
<td>00A2 19</td>
<td>DAD D</td>
<td></td>
</tr>
<tr>
<td>00A3 C9</td>
<td>RET</td>
<td></td>
</tr>
</tbody>
</table>

### MULHL
Multiply HL = DE using iterative ADD.

#### Entry parameters
- HL = multiplicand
- DE = multiplier

#### Exit parameters
- HL = product |
- DE = multiplier

### MULHL:
Save user register

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>00A4 C5</td>
<td>PUSH B</td>
<td>Check if either multiplicand</td>
</tr>
<tr>
<td></td>
<td></td>
<td>or multiplier is 0</td>
</tr>
</tbody>
</table>
MOV A,H
ORA L, R5
JZ MULHLZ ;Yes, fake product

MOV A,D
ORA E
JC MULHLZ ;Yes, fake product

MOV A,D
CMP H
JZ MULHLZ ;Check which is smaller
C set if D < H, so no exchange

XCHG

MOV B,O
MOV C,E
MOV O,H
MOV E,L
OCX B

MULHLN:
MOV A,B,D
;BC = multiplier
MOV C,E
;DE = multiplicand
MOV D,H
;HL = multiplicand + multiplicand
MOV E,L
;Adjust count as
DCX B ;1 * multiplicand = multiplicand

MULHLA:
MOV A,B
;Check if all iterations completed
MOV A,C
;DE = subtractor
JZ MULHLX ;Countdown on multiplier - 1

MULHLZ:
LXI H,O
;Fake product as either multiplicand
; or multiplier is O

MULHLX:
POP B
;Recover user register
RET

;SUBHL
;Subtract HL - DE

SUBHL:
;Entry parameters
; HL = subtrahend
; DE = subtractor

;Exit parameters
; HL = difference

;SHLR
;Shift HL right one place (dividing HL by 2)

SHLR:
;Entry parameters
; HL = value to be shifted

;Exit parameters
; HL = value/2

SHLR:
;Clear carry

;Get MS byte
;Bit 7 set from previous carry,
;bit 0 goes into carry

;Put shift MS byte back

;Get LS byte
;Bit 7 = bit 0 of MS byte

;Put back into result

Figure 5-24. (Continued)
Chapter 5: The Basic Disk Operating System

**Function 32: Set/Get User Number**

Function Code: \( C = 20H \)

Entry Parameters: \( E = 0FFH \) to get user number, or
\( E = 0 \) to 15 to set user number

Exit Parameters: \( A = \) Current user number if \( E = 0FFH \)

**Example**

<table>
<thead>
<tr>
<th>Hex</th>
<th>Binary</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0020</td>
<td>0000 0020</td>
<td>B#SEGETUN EQU 32 ; Set/Get User Number</td>
</tr>
<tr>
<td>0005</td>
<td>0005 0005</td>
<td>BDOS EQU 5 ; BDOS entry point</td>
</tr>
<tr>
<td>0000</td>
<td>0000 0020</td>
<td>MVI C, B#SEGETUN ; To set user number</td>
</tr>
<tr>
<td>0002</td>
<td>0002 1EEF</td>
<td>MVI E, 15 ; Required user number</td>
</tr>
<tr>
<td>0004</td>
<td>0004 CD0500</td>
<td>CALL BDOS ; To get user number</td>
</tr>
<tr>
<td>0007</td>
<td>0007 0E20</td>
<td>MVI C, B#SEGETUN ; Function code</td>
</tr>
<tr>
<td>0009</td>
<td>0009 1EFF</td>
<td>MVI E, OFFH ; Indicate request to GET</td>
</tr>
<tr>
<td>0008</td>
<td>0008 CD0500</td>
<td>CALL BDOS ; A = Current user no. (0 -- 15)</td>
</tr>
</tbody>
</table>

**Purpose**

This subroutine either sets or gets the current user number. The current user number determines which file directory entries are matched during all disk file operations.

When you call this function, the contents of the \( E \) register specify what action is to be taken. If \( E = 0FFH \), then the function will return the current user number in the \( A \) register. If you set \( E \) to a number in the range 0 to 15 (that is, a valid user number), the function will set the current user number to this value.

**Notes**

You can use this function to share files with other users. You can locate a file by attempting to open a file and switching through all of the user numbers. Or you can share a file in another user number by setting to that number, operating on the file, and then reverting back to the original user number.

If you do change the current user number, make provisions in your program to return to the original number before your program terminates. It is disconcerting for computer operators to find that they are in a different user number after a program. Files can easily be damaged or accidentally erased this way.

**Function 33: Read Random**

Function Code: \( C = 21H \)

Entry Parameters: \( DE = \) Address of FCB

Exit Parameters: \( A = \) Return code

**Example**

<table>
<thead>
<tr>
<th>Hex</th>
<th>Binary</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0021</td>
<td>0021 0021</td>
<td>B#READRAN EQU 33 ; Read Random</td>
</tr>
<tr>
<td>0005</td>
<td>0005 0005</td>
<td>BDOS EQU 5 ; BDOS entry point</td>
</tr>
<tr>
<td>FCB:</td>
<td>00 00</td>
<td>FCB#DISK: DB 0 ; Search on default disk drive</td>
</tr>
<tr>
<td>0001</td>
<td>46494C454E</td>
<td>FCB#NAME: DB 'FILENAME' ; File name</td>
</tr>
<tr>
<td>0009</td>
<td>545950</td>
<td>FCB#TYP: DB 'TYP' ; File type</td>
</tr>
</tbody>
</table>
This function reads a specific CP/M record (128 bytes) from a random file—that is, a file in which records can be accessed directly. It assumes that you have already opened the file, set the DMA address using the BDOS Set DMA function, and set the specific record to be read into the random record number in the FCB. This function computes the extent of the specified record number and attempts to open it and read the correct CP/M record into the DMA address.

The random record number in the FCB is three bytes long (at relative bytes 33, 34, and 35). Byte 33 is the least significant byte, 34 is the middle byte, and 35 the most significant. CP/M uses only the most significant byte (35) for computing the overall file size (function 35). You must set this byte to 0 when setting up the FCB. Bytes 33 and 34 are used together for the Read Random, so you can access from record 0 to 65535 (a maximum file size of 8,388,480 bytes).

This function returns with A set to 0 to indicate that the operation has been completed successfully, or A set to a nonzero value if an error has occurred. The error codes are as follows:

- A = 01 (attempt to read unwritten record)
- A = 03 (CP/M could not close current extent)
- A = 04 (attempt to read unwritten extent)
- A = 06 (attempt to read beyond end of disk)

Unlike the Read Sequential BDOS function (code 20, 14H), which updates the current (sequential) record number in the FCB, the Read Random function leaves the record number unchanged, so that a subsequent Write Random will replace the record just read.

You can follow a Read Random with a Write Sequential (code 21, 15H). This
will rewrite the record just read, but will then update the sequential record number. Or you may choose to use a Read Sequential after the Read Random. In this case, the same record will be reread and the sequential record number will be incremented. In short, the file can be sequentially read or written once the Read Random has been used to position to the required place in the file.

Notes

To use the Read Random function, you must first open the base extent of the file, that is, extent 0. Even though there may be no actual data records in this extent, opening permits the file to be processed correctly.

One problem that is not immediately obvious with random files is that they can easily be created with gaps in the file. If you were to create the file with record number 0 and record number 5000, there would be no intervening file extents. Should you attempt to read or copy the file sequentially, even using CP/M's file copy utility, only the first extent (and in this case, record 0) would get copied. A Read Sequential function would return an "end of file" error after reading record 0. You must therefore be conscious of the type of the file that you try and read.

See Figure 5-26 for an example subroutine that performs Random File Reads and Writes. It reads or writes records of sizes other than 128 bytes, where necessary reading or writing several CP/M records, prereading them into its own buffer when the record being written occupies only part of a CP/M record. It also contains subroutines to produce a 32-bit product from multiplying HL by DE (MLDL—Multiply double length) and a right bit shift for DE, HL (SDLR—Shift double length right).

**Function 34: Write Random**

Function Code: \[ C = 22H \]
Entry Parameters: \[ DE = \text{Address of file control block} \]
Exit Parameters: \[ A = \text{Return code} \]

Example

```
0022 = $WRITERAN EQU 34 ;Write Random
0005 = BDOS EQU 5 ;BDOS entry point

FCB: DB 0 ;File control block
FCB*DISK: DB 0 ;Search on default disk drive
FCB*NAME: DB 'FILENAME' ;File name
FCB*TYPE: DB 'TYP' ;File type
FCB*EXTENT: DB 0 ;Extent
FCB*RESV: DB 0 ;Reserved for CP/M
FCB*RECUSED: DB 0 ;Records used in this extent
FCB*ABUSED: DB 0 ;Allocation blocks used
FCB*SEQREC: DB 0 ;Sequential rec. to read/write
FCB*RANREC: DB 0 ;Random rec. to read/write
FCB*RANRECO: DB 0 ;Random rec. overflow byte (MS)

0024 D204 RANRECN: DW 1234 ;Example random record number
```

;Record will be written from
;address set by prior
;SETDMA call
This function writes a specific CP/M record (128 bytes) into a random file. It is initiated in much the same way as the companion function, Read Random (code 33, 21H). It assumes that you have already opened the file, set the DMA address to the address in memory containing the record to be written to disk, and set the random record number in the FCB to the specified record being written. This function also computes the extent in which the specified record number lies and opens the extent (creating it if it does not already exist). The error codes returned in A by this call are the same as those for Read Random, with the addition of error code 05, which indicates a full directory.

Like the Read Random (but unlike the Write Sequential), this function does not update the logical extent and sequential (current) record number in the FCB. Therefore, any subsequent sequential operation will access the record just written by the Read Random call, but these functions will update the sequential record number. The Write Random can therefore be used to position to the required place in the file, which can then be accessed sequentially.

In order to use the Write Random, you must first open the base extent (extent 0) of the file. Even though there may be no data records in this extent, opening permits the file to be processed correctly.

As explained in the notes for the Read Random function, you can easily create a random file with gaps in it. If you were to create a file with record number 0 and record number 5000, there would be no intervening file extents.

Figure 5-25 shows an example subroutine that creates a random file (CRF) but avoids this problem. You specify the number of 128-byte CP/M records in the file. The subroutine creates the file and then writes zero-filled records throughout. This makes it easier to process the file and permits standard CP/M utility programs to copy the file because there is a data record in every logical record position in the file. It is no longer a “sparse” file.

Figure 5-26 shows a subroutine that ties the Read and Write Random functions together. It performs Random Operations (RO). Unlike the standard BDOS functions that operate on 128-byte CP/M records, RO can handle arbitrary record size from one to several thousand bytes. You specify the relative record number of your record, not the CP/M record number (RO computes this). RO also prereads a CP/M record when your logical record occupies part of a 128-byte record, either because your record is less than 128 bytes or because it spans more than one
Chapter 5: The Basic Disk Operating System

Chapter 5: The Basic Disk Operating System

**Create random file**

This subroutine creates a random file. It erases any previous file before creating the new one, and then writes O-filled records throughout the entire file.

**Entry parameters**
- DE = file control block for new file
- HL = Number of 128-byte CP/M records to be zero-filled.

**Exit parameters**
- Carry clear if operation successful (A = 0, 1, 2, 3)
- Carry set if error (A = OFFH)

**Calling sequence**
- LXI D, FCB
- CALL CRF
  - JC ERROR

**Figure 5-25.** Create random file
128-byte sector. The subroutine suppresses this preread if you happen to use a record size that is some multiple of 128 bytes. In this case, your records will fit exactly onto a 128-byte record, so there will never be some partially occupied 128-byte sector.

This example also contains subroutines to produce a 32-bit product from multiplying HL by DE (MLDL—Multiply double length) and a right bit shift for DE, HL (SDLR—Shift double length right).

This subroutine reads or writes a random record from a file. The record length can be other than 128-bytes. This subroutine computes the start CP/M record (which is 128 bytes), and, if reading, performs a random read and moves the user-specified record into a user buffer. If necessary, more CP/M records will be read until the complete user-specified record has been input. For writing, if the size of the user-specified record is not an exact multiple of CP/M records, the appropriate sectors will be preread. It is not necessary to preread when the user-specified record is an exact CP/M record, nor when subroutine is processing CP/M records entirely spanned by a user-specified record.

Entry parameters:
- HL -> parameter block of the form:
  - DB 0:OFFH when reading, OOH for write
  - DW FCB:Pointer to FCB
  - DW RECSZ:User record number
  - DW RECSI:User record size
  - DW BUFFER:Pointer to buffer of
    - RECSI bytes in length

Exit parameters:
- A = 0 if operation completed (and user record copied into user buffer)
- 1 if attempt to read unwritten CP/M record
- 3 if CP/M could not close an extent
- 4 if attempt to read unwritten extent
- 5 if CP/M could not create a new extent
- 6 if attempt to read beyond end of disk

Calling sequence:
- LXI H,PAPMRS ;HL -> parameter block
- CALL RO
- DJNZ ERROR ;Check if error

0021 = FCBE$RANREC EQU 33 ;Offset of random record no. in FCB
001A = B$SETDMA EQU 26 ;Set the DMA address
0021 = B$READRAN EQU 33 ;Read random record
0026 = B$WRITERANZ EQU 40 ;Write random record with zero-fill previously unallocated allocation
  blocks
0005 = BDDS EQU 5 ;BDDS entry point
0000 00 ;Parameter block image
0001 0000 ;Parameter block image
0003 0000 ;Parameter block image
0005 0000 ;Parameter block image
0007 0000 ;Parameter block image
0009 = ROPLB EQU $-ROPB ;Parameter block length
0009 0000 ;Parameter block length

Figure 5-26. Read/Write variable length records randomly
Chapter 5: The Basic Disk Operating System

Figure 5-26. (Continued)
00DE A0  ANAL  B  ;Form logical AND
00DF 320E00  STA   ROMECR  ;Save back in flag

;Recover the double length byte offset within the file
;of the start of the user record. Shift 7 places right
;to divide by 128 and get the CP/M record number for
;the start of the user record.
00E2 EI    POP    H       ;Recover user rec. byte offset
00E3 DI    POP    D       ;Count for shift right

ROS:
00E6  CDF101  CALL  SDLR  ;DE, HL = DE, HL / 2
00E9  0D    DCR    C       
00EA  C2E600  JNZ  ROS

00ED 7A    MOV    A, D  ;Error if DE still NZ after
00EE  B3    ORA    E       
00EF  C2AC01  JNZ  ROERO

00F2  EB    XCHG  
00F3  2A0100  LHL    RORCB  ;Get pointer to FCB
00F6  012100  LI   B, FCBEPLANREC  ;Offset of random record no. in FCB
00F9  0D    DAD    B       ;HL -> ran. rec. no. in FCB
00FA  220C00  SHLD  RORNP  ;Save record number pointer
00FD  73    MOV    H, E  ;Store LS byte
00FE  23    INX    H       
00FF  72    MOV    H, D  ;Store MS byte

0100  OE1A  MVI    C, B, SETDMA  ;Set DMA address to local buffer
0102  11F000  LI   D, ROBUF
0105  CD0500  CALL  BDOS

0108  3A0E00  LDA   'ROMEWR  ;Bypass preread if exact sector write
010B  B7    ORA    A       
010C  C21F01  JNZ  ROMNF

010F  2A0100  LHL    RORCB  ;Get pointer to FCB
0112  EB    XCHG  ;DE -> FCB
0113  OE21  MVI    C, B, READRAN  ;Read random function
0115  CD0500  CALL  BDOS

0118  FE05  CPI    5  ;Check if error code < 5
011A  DCAF01  CC    ROCIE  ;Yes, check if ignorable error
011B  B7    ORA    A  ;(i.e. error reading unwritten part
011C  C0    RNI    ;of file for write operation preread)

ROMNF:
011F  2A0700  LHL    ROUB  ;Move next fragment
0122  EB    XCHG  ;Get pointer to user buffer
0123  2A0900  LHL    RORFRP  ;DE -> user buffer
0126  3A0800  LDA   ROFRPL  ;HL -> start of user rec. in local buffer
0129  4F    MOV    C, A  ;Get fragment length
012A  3A0000  LDA   ROREAD  ;Ready for MOVE
012D  B7    ORA    A       
012E  C23201  JNZ  RORD1  ;Yes, so leave DE, HL unchanged
0131  EB    XCHG  ;Writing, so swap source and destination

RORD1:
0132  CDFE01  CALL  MOVE  ;DE -> fragment local -> user buffer
0135  3A0000  LDA   ROREAD  ;Writing -> fragment user -> local buffer
0138  B7    ORA    A  ;Check if writing
0139  CA3D01  JZ    ROWR1  ;Writing, so leave HL -> user buffer
013C  EB    XCHG  ;HL -> next byte in user buffer

ROWR1:
0140  220700  SHLD  ROUB  ;Save updated user buffer pointer

0140  3A0000  LDA   ROREAD  ;Check if reading

---

Figure 5-26. (Continued)
Chapter 5: The Basic Disk Operating System

Figure 5-26. (Continued)

0143 B7 ORA A ;Yes, bypass write code
0144 C25001 JNZ RORD3
0147 0E28 MVI C,B\text{WRITERANZ} ;Write random
0149 2A0100 LHLD ROFCB ;Get address of FCB
014C EB MVI D,E
014D CALL BDOS
RORD3: ;Compute residual length of user record as yet unmoved. ;If necessary (because more data needs to be transferred) ;more CP/M records will be read. In this case ;the start of the fragment will be offset 0. The fragment ;length depends on whether the user record finishes within ;the next sector or spans it. If the residual length of the ;user record is > 128, the fragment length will be set to ;128.

0150 2A0500 LHLD ROURL ;Get residual user rec. length
0153 3A0800 LDA ROFRDL ;Get fragment length just moved
0156 SF MOVLDE,A,E ;Make into a word value
0157 1600 MVI D,0
0159 CD\text{DEA}1 CALL SUBHL ;Compute ROURL - ROFRDL
015C 7C MOVA,H
015D B5 ORAL
015E CB RZ ;Return when complete USER ; record has been transferred
015F 220500 SL\text{DH} ROURL ;Save downarded residual rec. length
0162 4D MOVLCO,L ;Assume residual length < 128
0163 118000 LXI D,128 ;Check if residual length is < 128
0165 6C CD\text{DEA}1 CALL SUBHL ;HL = HL - DE
0166 F\text{A}6E01 JM ROLT128 ;negative if < 128
0168 0E80 MVI C,128 ;=> 128, so set frag.length to 128

ROURL:

0168 79 MOV A,C ;Fragment length now is either 128
016F 320800 STA ROFRDL ; if more than 128 bytes left to input ; in user record, or just the right ; number of bytes (< 128) to complete ; the user record.

0172 210F00 LXI H,ROBUF ;All subsequent CP/M records will start ; at beginning of buffer
0175 220900 SL\text{DL} ROFRP
0178 2A0C00 LHLD RO\text{RMP} ;Update random record number in FCB
017B 5E MOVEL,M,H ;Increment the random record number
017C 23 INX H ;HL -> MS byte of record number
017D 76 MOVD,M,D ;Get MS byte
017E 13 INXD,D ;Update record number itself
017F 7A MOVA,D
0180 B3 ORAE
0181 C29701 JNZ RO\text{RMP} ;No, so save record number
0184 3E06 MVI A,6 ;Indicate "seek past end of disk" ;ROURL:
0186 C9 RET ;Return to user

RORMP:

0187 72 MOVM, D ;Save record number
0188 2B D\text{CX} H ;HL -> LS byte
0189 73 MOVM,E
018A 3A0E00 LDA RO\text{MCRR} ;If writing, check if preread required
018B 87 ORA A ;Check if exact CP/M record write
018E C21F01 JNZ ROM\text{NFO} ;Yes, go move next fragment
0191 3A0000 LDA RO\text{READ} ;If reading, perform read unconditionally
0194 B7 ORA A
0195 C2A001 JNZ RORD2
0198 3A0800 LDA ROFRDL ;For writes, bypass preread if
019B FE80 CPI 126 ;whole CP/M-record is to be overwritten
019D C1F01 JZ ROM\text{NFO} ; (fragment length = 128)
RORD2:

01A0 0E21 MVI C,B\text{READRAN} ;Read the next CP/M record
01A2 2A0100 LHLD ROFCB ; in sequence

(Continued)
Figure 5-26. (Continued)
Chapter 5: The Basic Disk Operating System

Figure 5-26. (Continued)
Function 35: Get File Size

Function Code: \( C = 23H \)

Entry Parameters: \( DE = \) Address of FCB

Exit Parameters: Random record field set in FCB

Example

```
0023 = B*GETFSIZ   EQU 35 ;Get Random File LOGICAL size
0005 = BDOS       EQU 5  ;BDOS entry point

FCB:           DB 0 ;File control block
0000 00 FCB*DISK: DB 0 ;Search on default disk drive
0001 46494C454EFCB*NAME: DB 'FILENAME' ;File name
0009 545950 FCB*TYPE: DB 'TYPE' ;File type
000C 00 FCB*EXTENT: DB 0 ;Extent
000D 0000 FCB*RESV: DB 0,0 ;Reserved for CP/M
000F 00 FCB*REUSED: DB 0 ;Records used in this extent
0010 0000000000FCB*ABUSED: DB 0,0,0,0,0,0,0,0 ;Allocation blocks used
0018 0000000000
0020 00 FCB*SEQREC: DB 0 ;Sequential rec. to read/write
0021 0000 FCB*RANREC: DW 0 ;Random rec. to read/write
0023 00 FCB*RANREC2: DB 0 ;Random rec. overflow byte (MS)

0024 0E23 MVI C,B*GETFSIZ ;Function code
0026 110000 LXI D,FCB ;DE -> file control block
0029 CD0500 CALL BDOS ;Get random record number
002C 2A2100 LHLD FCB*RANREC ;i.e. the record number of the last record
```

Purpose

This function returns the virtual size of the specified file. It does so by setting the random record number (bytes 33-35) in the specified FCB to the maximum 128-byte record number in the file. The virtual file size is calculated from the record address of the record following the end of the file. Bytes 33 and 34 form a 16-bit value that contains the record number, with overflow indicated in byte 35. If byte 35 is 01, this means that the file has the maximum record count of 65,536.

If the function cannot find the file specified by the FCB, it returns with the random record field set to 0.

You can use this function when you want to add data to the end of an existing file. By calling this function first, the random record bytes will be set to the end of file. Subsequent Write Random calls will write out records to this preset address.

Notes

Do not confuse the virtual file size with the actual file size. In a random file, if you write just a single CP/M record to record number 1000 and then call this function, it will return with the random record number field set in the FCB to 1000—even though only a single record exists in the file.

For sequential files, this function returns the number of records in the file. In this case, the virtual and actual file sizes coincide.

Function 36: Set Random Record Number

Function Code: \( C = 24H \)

Entry Parameters: \( DE = \) Address of FCB

Exit Parameters: Random record field set in FCB
Example

0024 = B$SETTRANREC EQU 36 ; Set Random Record Number
0005 = BDOS EQU 5 ; BDOS entry point
0000 00 FCB = File control block
0001 46494C454E FCB$DISK = Search on default disk drive
0009 545950 FCB$NAME = File name
000C 00 FCB$EXTENT = File type
000D 00 FCB$RESV = Extent
000F 00 FCB$RECUSED = Records used in this extent
0010 0000000000 FCB$ABUSED = Allocation blocks used
0018 0000000000
0020 00 FCB$SEGREC = Sequential rec. to read/write
0021 0000 FCB$RANREC = Random rec. to read/write
0023 00 FCB$RANRECO = Random rec. overflow byte (MS)

... file opened and read or written sequentially...

MVI C,B$SETTRANREC ; Function code
LXI D,FCB ; DE -> file control block
CALL BDOS ; Get random record number
LHLD FCB$RANREC ; that corresponds to the sequential progress down the file.

Purpose

This function sets the random record number in the FCB to the correct value for the last record read or written sequentially to the file.

Notes

This function provides you with a convenient way to build an index file so that you can randomly access a sequential file. Open the sequential file, and as you read each record, extract the appropriate key field from the data record. Make the BDOS Set Random Record request and create a new data record with just the key field and the random record number. Write the new data record out to the index file.

Once you have done this for each record in the file, your index file provides a convenient method, given a search key value, of finding the appropriate CP/M record in which the data lies.

You can also use this function as a means of finding out where you are currently positioned in a sequential file—either to relate a CP/M record number to the position, or simply as a place-marker to allow a repositioning to the same place later.

Function 37: Reset Logical Disk Drive

Function Code: C = 25H
Entry Parameters: DE = Logical drive bit map
Exit Parameters: A = 00H

Example

0025 = B$RESETD EQU 37 ; Reset Logical Disks
0005 = BDOS EQU 5 ; BDOS entry point
This function resets individual disk drives. It is a more precise version of the Reset Disk System function (code 13,ODH), in that you can set specific logical disks rather than all of them.

The bit map in DE shows which disks are to be reset. The least significant bit of E represents disk A, and the most significant bit of D, disk P. The bits set to 1 indicate the disks to be reset.

Note that this function returns a zero value in A in order to maintain compatibility with MP/M.

Notes
Use this function when only specific diskettes need to be changed. Changing a diskette without requesting CP/M to log it in will cause the BDOS to assume that an error has occurred and to set the new diskette to Read-Only status as a protective measure.

**Function 40: Write Random with Zero-fill**

Function Code: \( C = 28H \)
Enter Parameters: DE = Address of FCB
Exit Parameters: A = Return Code

Example

```assembly
0028 = B@WRITERANZ EQU 40 ;Write Random with Zero-Fill
0029 = BDOS EQU 5 ;BDOS entry point

FB: ;File control block
0000 00 FCB@DISK: DB 0 ;Search on default disk drive
0001 4649444546F430DB 'FILENAME' ;File name
0009 54454950 DB 'TYP' ;File type
000C 00 FCB@EXTENT: DB 0 ;Extent
000D 0000 FCB@RESV: DB 0,0 ;Reserved for CP/M
000F 00 FCB@RECUSED: DB 0 ;Records used in this extent
0010 0000000000FCB@ABUSED: DB 0,0,0,0,0,0,0,0 ;Allocation blocks used
0018 00000000 DB 0,0,0,0,0,0,0,0
0020 00 FCB@SEQREC: DB 0 ;Sequential rec. to read/write
0021 0000 FCB@RANREC: DW 0 ;Random rec. to read/write
0023 00 FCB@RANREC0: DB 0 ;Random rec. overflow byte (MS)
0024 D204 RANRECNO: DW 1234 ;Example random record number

0026 2A2400 LHLD RANRECNO ;Record will be written from
0029 222100 ShLD FCB@RANREC ;address set by prior
002C 0E28 MVI C,B@WRITERANZ ;SETDM call
002E 110000 LXI D,FCB ;Get random record number
0031 CD0500 CALL BDOS ;Set up file control block
```

;DE = Bit map of disks to reset
;Bits are 1 if disk to be reset
;Bits 15 14 13 ... 2 1 0
;Disk P O N ... C B A
This function is an extension to the Write Random function described previously. In addition to performing the Write Random, it will also fill each new allocation block with 00H's. Digital Research added this function to assist Micro­soft with the production of its COBOL compiler—it makes the logic of the file handling code easier. It also is an economical way to completely fill a random file with 00H's. You need only write one record per allocation block; the BDOS will clear the rest of the block for you.

Notes Refer to the description of the Write Random function (code 34).
This chapter takes a closer look at the Basic Input/Output System (BIOS). The BIOS provides the software link between the Console Command Processor (CCP), the Basic Disk Operating System (BDOS), and the physical hardware of your computer system. The CCP and BDOS interact with the parts of your computer system only as logical devices. They can therefore remain unchanged from one computer system to the next. The BIOS, however, is customized for your particular type of computer and disk drives. The only predictable part of the BIOS is the way in which it interfaces to the CCP and BDOS. This must remain the same no matter what special features are built into the BIOS.
The BIOS Components

A standard BIOS consists of low-level subroutines that drive four types of physical devices:

- **Console**: CP/M communicates with the outside world via the console. Normally this will be a video terminal or a hard-copy terminal.

- **"Reader" and "punch"**: These devices are normally used to communicate between computer systems—the names "reader" and "punch" are just historical relics from the early days of CP/M.

- **List**: This is a hard-copy printer, either letter-quality or dot-matrix.

- **Disk drives**: These can be anything from the industry standard single-sided, single-density, 8-inch floppy diskette drives to hard disk drives with capacities of several hundred megabytes.

The BIOS Entry Points

The first few instructions of the BIOS are all jump (JMP) instructions. They transfer control to the 17 different subroutines in the BIOS. The CCP and the BDOS, when making a specific request of the BIOS, do so by transferring control to the appropriate JMP instruction in this BIOS jump table or jump vector. The BIOS jump vector always starts at the beginning of a 256-byte page, so the address of the first jump instruction is always of the form xx00H, where "xx" is the page address. Location 0000H to 0002H has a jump instruction to the second entry of the BIOS jump vector—so you can always find the page address of the jump vector by looking in location 0002H.

Figure 6-1 shows the contents of the BIOS jump vector along with the page-relative address of each jump. The labels used in the jump instructions have been adopted by convention.

The following sections describe the functions of each of the BIOS's main subroutines. You should also refer to Digital Research's manual *CP/M 2.0 Alteration Guide* for their description of the BIOS routines.

Bootstrap Functions

There are two bootstrap functions. The cold bootstrap loads the entire CP/M operating system when the system is either first turned on or reset. The warm bootstrap reloads the CCP whenever a program branches to location 0000H.
Chapter 6: The Basic Input/Output System

BOOT: "Cold" Bootstrap

The BOOT jump instruction is the first instruction executed in CP/M. The bootstrap sequence must transfer control to the BOOT entry point in order to bring up CP/M. In general, a PROM receives control either when power is first applied or after you press the RESET button on the computer. This reads in the CP/M loader on the first sector of the physical disk drive chosen to be logical disk A. This CP/M loader program reads the binary image of the CCP, BDOS, and BIOS into memory at some predetermined address. Then it transfers control to the BOOT entry point in the BIOS jump vector.

This BOOT routine must initialize all of the required computer hardware. It sets up the baud rates for the physical console (if this has not already been done during the bootstrap sequence), the "reader," "punch," and list devices, and the disk controller. It must also set up the base page of memory so that there is a jump at location 0000H to the warm boot entry point in the BIOS jump vector (at xx03H) and a jump at location 0005H to the BDOS entry point.

Most BOOT routines sign on by displaying a short message on the console, indicating the current version of CP/M and the computer hardware that this BIOS can support.

The BOOT routine terminates by transferring control to the start of the CCP + 6 bytes (the CCP has its own small jump vector at the beginning). Just before the BOOT routine jumps into the CCP, it sets the C register to 0 to indicate that logical disk A is to be the default disk drive. This is what causes "A>" to be the CCP's initial prompt.

The actual CCP entry point is derived from the base address of the BIOS. The CCP and BDOS together require 1E00H bytes of code, so the first instruction of the CCP starts at BIOS – 1E00H.

Figure 6-1. Layout of the standard BIOS jump vector

| xx00H | JMP BOOT | ;"Cold" (first time) bootstrap |
| xx03H | JMP WBOOT | ;"Warm" bootstrap |
| xx06H | JMP CONST | ;Console input status |
| xx09H | JMP CONIN | ;Console input |
| xx0CH | JMP CONOUT | ;Console output |
| xx0FH | JMP LIST | ;List output |
| xx12H | JMP PUNCH | ;"Punch" output |
| xx15H | JMP READER | ;"Reader" input |
| xx18H | JMP HOME | ;Home disk heads (to track 0) |
| xx1BH | JMP SELDSK | ;Select logical disk |
| xx1EH | JMP SETTRK | ;Set track number |
| xx21H | JMP SETSEC | ;Set sector number |
| xx24H | JMP SETDMA | ;Set DMA address |
| xx27H | JMP READ | ;Read (128-Byte) sector |
| xx2AH | JMP WRITE | ;Write (128-Byte) sector |
| xx2DH | JMP LISTST | ;List device output status |
| xx30H | JMP SECTRAN | ;Sector translate |
WBOOT: "Warm" Bootstrap

Unlike the "cold" bootstrap entry point, which executes only once, the WBOOT or warm boot routine will be executed every time a program terminates by jumping to location 0000H, or whenever you type a CONTROL-C on the console as the first character of an input line.

The WBOOT routine is responsible for reloading the CCP into memory. Programs often use all of memory up to the starting point of the BDOS, overwriting the CCP in the process. The underlying philosophy is that while a program is executing, the CCP is not needed, so the program can use the memory previously occupied by the CCP. The CCP occupies 800H (2048) bytes of memory — and this is frequently just enough to make the difference between a program that cannot run and one that can.

A few programs that are self-contained and do not require the BDOS’s facilities will also overwrite the BDOS to get another 1600H (5632) bytes of memory. Therefore, to be really safe, the WBOOT routine should read in both the CCP and the BDOS. It also needs to set up the two JMPs at location 0000H (to WBOOT itself) and at location 0005H (to the BDOS). Location 0003H should be set to the initial value of the IOBYTE if this is implemented in the BIOS.

As its last act, the WBOOT routine sets register C to indicate which logical disk is to be selected (C = 0 for A, 1 for B, and so on). It then transfers control into the CCP at the first instruction in order to restart the CCP. Again, the actual address is computed based on the knowledge that the CCP starts 1E00H bytes lower in memory than the base address of the BIOS.

Character Input/Output Functions

Character input/output functions deal with logical devices: the console, "reader," “punch,” and list devices. Because these logical devices can in practice be connected by software to one of several physical character I/O devices, many BIOS’s use CP/M’s IOBYTE features to assign logical devices to physical ones.

In this case, each of the BIOS functions must check the appropriate bit fields of the IOBYTE (see Figure 4-2 and Table 4-1) to transfer control to the correct physical device driver (program that controls a physical device).

CONST: Console Input Status

CONST simply returns an indicator showing whether there is an incoming character from the console device. The convention is that A = 0FFH if a character is waiting to be processed, A = 0 if one is not. Note that the zero flag need not be set to reflect the contents of the A register — it is the contents that are important.

CONST is called by the CCP whenever the CCP is in the middle of an operation that can be interrupted by pressing a keyboard character.
The BDOS will call CONST if a program makes a Read Console Status function call (B$CONST, code 11, 0BH). It is also called by the console input BIOS routine, CONIN (described next).

**CONIN: Console Input**

CONIN reads the next character from the console to the A register and sets the most significant (parity) bit to 0.

Normally, CONIN will call the CONST routine until it detects A = 0FFH. Only then will it input the data character and mask off the parity bit.

CONIN is called by the CCP and by the BDOS when a program executes a Read Console Byte function (B$CONIN, code 1).

**CONOUT: Console Output**

CONOUT outputs the character (in ASCII) in register C to the console. The most significant (parity) bit of the character will always be 0.

CONOUT must first check that the console device is ready to receive more data, delaying if necessary until it is, and only then sending the character to the device.

CONOUT is called by the CCP and by the BDOS when a program executes a Write Console Byte function (B$CONOUT, code 2).

**LIST: List Output**

LIST is similar to CONOUT except that it sends the character in register C to the list device. It too checks first that the list device is ready to receive the character.

LIST is called by the CCP in response to the CONTROL-P toggle for printer echo of console output, and by the BDOS when a program makes a Write Printer Byte or Display String call (B$LISTOUT and B$PRINTS, codes 5 and 9).

**PUNCH: "Punch" Output**

PUNCH sends the character in register C to the "punch" device. As mentioned earlier, the "punch" is rarely a real paper tape punch. In most BIOS's, the PUNCH entry point either returns immediately and is effectively a null routine, or it outputs the character to a communications device, such as a modem, on your computer.

PUNCH must check that the "punch" device is indeed ready to accept another character for output, and must wait if it is not.

Digital Research's documentation states that the character to be output will always have its most significant bit set to 0. This is not true. The BDOS simply transfers control over to the PUNCH entry point in the BIOS; the setting of the most significant bit will be determined by the program making the BDOS function request (B$PUNOUT, code 4). This is important because the requirement of a zero
would preclude being able to send pure binary data via the BIOS PUNCH
function.

**READER: “Reader” Input**

As with the PUNCH entry point, the READER entry point rarely connects to
a real paper tape reader.

The READER function must return the next character from the reader device
in the A register, waiting, if need be, until there is a character.

Digital Research’s documentation again says that the most significant bit of
the A register must be 0, but this is not the case if you wish to receive pure binary
information via this function.

READER is called whenever a program makes a Read “Reader” Byte function
request (BSREADIN, code 3).

---

**Disk Functions**

All of the disk functions that follow were originally designed to operate on the
128-byte sectors used on single-sided, single-density, 8-inch floppy diskettes that
were standard in the industry at the time. Now that CP/M runs on many different
types of disks, some of the BIOS disk functions seem strange because most of the
new disk drives use sector sizes other than 128 bytes.

To handle larger sector sizes, the BIOS has some additional code that makes
the BDOS respond as if it were still handling 128-byte sectors. This code is referred
to as the *blocking/deblocking* code. As its name implies, it blocks together several
128-byte “sectors” and only writes to the disk when a complete *physical* sector has
been assembled. When reading, it reads in a physical sector and then deblocks it,
handing back several 128-byte “sectors” to the BDOS.

To do all of this, the blocking/deblocking code uses a special buffer area of the
same size as the physical sectors on the disk. This is known as the host disk buffer
or HSTBUF. Physical sectors are read into this buffer and written to the disk from it.

In order to optimize this blocking/deblocking routine, the BIOS has code in it
to reduce the number of times that an actual disk read or write occurs. A side effect
is that at any given moment, several 128-byte “sectors” may be stored in the
HSTBUF, waiting to be written out to the disk when HSTBUF becomes full. This
sometimes complicates the logic of the BIOS disk functions. You cannot simply
select a new disk drive, for example, when the HSTBUF contains data destined for
another disk drive. You will see this complication in the BIOS only in the form of
added logical operations; the BIOS disk functions rarely trigger immediate physi-
cal operations. It is easier to understand these BIOS functions if you consider that
they make *requests* — and that these requests are satisfied only when it makes sense to do so, taking into account the blocking/deblocking logic.

**HOME: Home Disk**

HOME sets the requested track and sector to 0.

**SELDSK: Select Disk**

SELDSK does not do what its name implies. It does not (and must not) physically select a logical disk. Instead, it returns a pointer in the HL register pair to the disk parameter header for the logical disk specified in register C on entry. C = 0 for drive A, 1 for drive B, and so on. SELDSK also stores this code for the requested disk to be used later in the READ and WRITE functions.

If the logical disk code in register C refers to a nonexistent disk or to one for which no disk parameter header exists, then SELDSK must return with HL set to 0000H. Then the BDOS will output a message of the form

"BDOS Err on X: Select"

Note that SELDSK not only does not select the disk, but also does not indicate whether or not the requested disk is physically present — merely whether or not there are disk tables present for the disk.

SELDSK is called by the BDOS either during disk file operations or by a program issuing a Select Disk request (B$SELDSK, code 14).

**SETRTK: Set Track**

SETRTK saves the requested disk track that is in the BC register pair when SETTRK gets control. Note that this is an absolute track number; that is, the number of reserved tracks before the file directory will have been added to the track number relative to the start of the logical disk.

The number of the requested track will be used in the next BIOS READ or WRITE function (described later in this chapter).

SETRTK is called by the BDOS when it needs to read or write a 128-byte sector. Legitimate track numbers are from 0 to OFFFFH (65,535).

**SETSEC: Set Sector**

SETSEC is similar to SETTRK in that it stores the requested sector number for later use in BIOS READ or WRITE functions. The requested sector number is handed to SETSEC in the A register; legitimate values are from 0 to 0FFH (255).

The sector number is a logical sector number. It does not take into account any sector skewing that might be used to improve disk performance.

SETSEC is called by the BDOS when it needs to read or write a 128-byte sector.
SETDMA: Set DMA Address

SETDMA saves the address in the BC register pair in the requested DMA address. The next BIOS READ or WRITE function will use the DMA address as a pointer to the 128-byte sector buffer into which data will be read or from which data will be written.

The default DMA address is 0080H. SETDMA is called by the BDOS when it needs to READ or WRITE a 128-byte sector.

READ: Read Sector

READ reads in a 128-byte sector provided that there have been previous BIOS function calls to

- SELDSK — "select" the disk
- SETDMA — set the DMA address
- SETTRK — set the track number
- SETSEC — set the sector number.

Because of the blocking/deblocking code in the BIOS, there are frequent occasions when the requested sector will already be in the host buffer (HSTBUF), so that a physical disk read is not required. All that is then required is for the BIOS to move the appropriate 128 bytes from the HSTBUF into the buffer pointed at by the DMA address.

Only during the READ function will the BIOS normally communicate with the physical disk drive, selecting it and seeking to read the requested track and sector. During this process, the READ function must also handle any hardware errors that occur, trying an operation again if a "soft," or recoverable, error occurs.

The READ function must return with the A register set to 00H if the read operation is completed successfully. If the READ function returns with the A register set to 01H, the BDOS will display an error message of the form

**BDOS Err on X: Bad Sector**

Under these circumstances, you have only two choices. You can enter a CARRIAGE RETURN, ignore the fact that there was an error, and attempt to make sense of the data in the DMA buffer. Or you can type a CONTROL-C to abort the operation, perform a warm boot, and return control to the CCP.

As you can see, CP/M's error handling is not particularly helpful, so most BIOS writers add more sophisticated error recovery right in the disk driver. This can include some interaction with the console so that a more determined effort can be made to correct errors or, if nothing else, give you more information as to what has gone wrong. Such error handling is discussed in Chapter 9.

If you are working with a hard disk system, the BIOS driver must also handle the management of bad sectors. You cannot simply replace a hard disk drive if one or two sectors become unreadable. This bad sector management normally requires
that a directory of "spare" sectors be put on the hard disk before it is used to store data. Then, when a sector is found to be bad, one of the spare sectors is substituted in its place. This is also discussed in Chapter 9.

**WRITE: Write Sector**

WRITE is similar to READ but with the obvious difference that data is transferred from the DMA buffer to the specified 128-byte sector. Like READ, this function requires that the following function calls have already been made:

- SELDSK — "select" the disk
- SETDMA — set the DMA address
- SETTRK — set the track number
- SETSEC — set the sector number.

Again, it is only in the WRITE routine that the driver will start to talk directly to the physical hardware, selecting the disk unit, track, and sector, and transferring the data to the disk.

With the blocking/deblocking code, the BDOS optimizes the number of disk writes that are needed by indicating in register C the type of disk write that is to be performed:

- 0 = normal sector write
- 1 = write to file directory sector
- 2 = write to sector of previously unused allocation block.

Type 0 occurs whenever the BDOS is writing to a data sector in an already used allocation block. Under these circumstances, the disk driver must preread the appropriate host sector because there may be previously stored information on it.

Type 1 occurs whenever the BDOS is writing to a file directory sector — in this case, the BIOS must not defer writing the sector to the disk, as the information is too valuable to hold in memory until the HSTBUF is full. The longer the information resides in the HSTBUF, the greater the chance of a power failure or glitch, making file data already physically written to the disk inaccessible because the file directory is out of date.

Type 2 occurs whenever the BDOS needs to write to the first sector of a previously unused allocation block. Unused, in this context, includes an allocation block that has become available as a result of a file being erased. In this case, there is no need for the disk driver to preread an entire host-sized sector into the HSTBUF, as there is no data of value in the physical sector.

As with the READ routine, the WRITE function returns with A set to 00H if the operation has been completed successfully. If the WRITE function returns with A set to 01H, then the BDOS will display the same message as for READ:

**BDOS Err on X: Bad Sector**
You can see now why most BIOS writers add extensive error-recovery and user-interaction routines to their disk drivers.

For hard disk systems, some disk drivers are written so that they automatically "spare out" a failing sector, writing the data to one of the spare sectors on the disk.

**LISTST: List Status**

As you can tell from its position in the list of BIOS functions, the LISTST function was a latecomer. It was added when CP/M was upgraded from version 1.4 to version 2.0.

This function returns the current status of the list device, using the IOBYTE if necessary to select the correct physical device. It sets the A register to OFFH if the list device can accept another character for output or to OOH if it is not ready.

Digital Research's documentation states that this function is used by the DESPOOL utility program (which allows you to print a file "simultaneously" with other operations) to improve console response during its operation, and that it is acceptable for the routine always to return OOH if you choose not to implement it fully.

Unfortunately, this statement is wrong. Many other programs use the LISTST function to "poll" the list device to make sure it is ready, and if it fails to come ready after a predetermined time, to output a message to the console indicating that the printer is not ready. If you ever make a call to the BDOS list output functions, Write Printer Byte and Print String (codes 5 and 9), and the printer is not ready, then CP/M will wait forever — and your program will have lost control so it cannot even detect that the problem has occurred. If LISTST always returns a OOH, then the printer will always appear not to be ready. Not only does this make nonsense out of the LISTST function, but it also causes a stream of false "Printer not Ready" error messages to appear on the console.

**SECTRAN: Sector Translate**

SECTRAN, given a logical sector number, locates the correct physical sector number in the sector translate table for the previously selected (via SELDSK) logical disk drive.

Note that both logical and physical sector numbers are 128-byte sectors, so if you are working with a hard disk system, it is not too efficient to impose a sector interlace at the 128-byte sector level. It is better to impose the sector interlace right inside the hard disk driver, if at all; in general, hard disks spin so rapidly that CP/M simply cannot take advantage of sector interlace.

The BDOS hands over the logical sector number in the BC register pair, with the address of the sector translate table in the DE register pair. SECTRAN must return the physical sector number in HL.

If SECTRAN is to be a null routine, it must move the contents of BC to HL and return.
Calling the BIOS Functions Directly

As a general rule, you should not make direct calls to the BIOS. To do so makes your programs less transportable from one CP/M system to the next. It precludes being able to run these programs under MP/M, which has a different form of BIOS called an extended I/O system, or XIOS.

There are one or two problems, however, that can only be solved by making direct BIOS calls. These occur in utility programs that, for example, need to make direct access to the CP/M file directory, or need to access some “private” jump instructions which have been added to the standard BIOS jump vector.

If you really do need direct access to the BIOS, Figure 6-2 shows an example subroutine that does this. It requires that the A register contain a BIOS function code indicating the offset in the jump vector of the jump instruction to which control is to be passed.

```
; Equates for use with BIOS subroutine
;
0003 = WBOOT EQU 03H ;Warm boot
0006 = CONST EQU 06H ;Console status
0009 = CONIN EQU 09H ;Console input
000C = CONOUT EQU 0CH ;Console output
000F = LIST EQU 0FH ;Output to list device
0012 = PUNCH EQU 12H ;Output to punch device
0015 = READER EQU 15H ;Input from reader
0018 = HOME EQU 18H ;Home selected disk to track 0
001B = SELDSK EQU 1BH ;Select disk
001E = SETTRK EQU 1EH ;Set track
0021 = SETSEC EQU 21H ;Set sector
0024 = SETDMA EQU 24H ;Set DMA address
0027 = READ EQU 27H ;Read 128-byte sector
002A = WRITE EQU 2AH ;Write 128-byte sector
002D = LISTST EQU 2DH ;Return list status
0030 = SECTRAN EQU 30H ;Sector translate
;
;Add further "private" BIOS codes here
;
; This subroutine transfers control to the appropriate
; entry in the BIOS Jump Vector, based on a code number
; handed to it in the L register.
;
; Entry parameters
;
L = Code number (which is in fact the page-relative
; address of the correct JMP instruction within
; the jump vector)
;
; All other registers are preserved and handed over to
; the BIOS routine intact.
;
; Exit parameters
```

Figure 6-2. BIOS equates
This routine does not CALL the BIOS routine, therefore when the BIOS routine RETURNS, it will do so directly to this routine's caller.

Calling sequence

```
0000 F5
0001 3A0200
0004 67
0005 F1
0006 E9
```

```
BIOS:
PUSH   PSW ;Save user's A register
LDA    0002H ;Get BIOS JMP vector page from warm boot JMP
MOV    H,A ;HL -> BIOS JMP vector entry
POP    PSW ;Recover user's A register
PCHL   ;Transfer control into the BIOS routine
```

**Figure 6-2.** BIOS equates (continued)

```
<table>
<thead>
<tr>
<th>Line Numbers</th>
<th>Functional Component or Routine</th>
</tr>
</thead>
<tbody>
<tr>
<td>0072-0116</td>
<td>BIOS Jump Vector</td>
</tr>
<tr>
<td>0120-0270</td>
<td>Initialization Code</td>
</tr>
<tr>
<td>0275-0286</td>
<td>Display Message</td>
</tr>
<tr>
<td>0289-0310</td>
<td>Enter CP/M</td>
</tr>
<tr>
<td>0333-0364</td>
<td>CONST - Console Status</td>
</tr>
<tr>
<td>0369-0393</td>
<td>CONIN - Console Input</td>
</tr>
<tr>
<td>0397-0410</td>
<td>CONOUT - Console Output</td>
</tr>
<tr>
<td>0414-0451</td>
<td>LISTST - List Status</td>
</tr>
<tr>
<td>0456-0471</td>
<td>LIST - List Output</td>
</tr>
<tr>
<td>0476-0492</td>
<td>PUNCH - Punch Output</td>
</tr>
<tr>
<td>0496-0511</td>
<td>READER - Reader Input</td>
</tr>
<tr>
<td>0516-0536</td>
<td>IOBYTE Driver Select</td>
</tr>
<tr>
<td>0540-0584</td>
<td>Device Control Tables</td>
</tr>
<tr>
<td>0589-0744</td>
<td>Low-level Drivers for Console, List, etc.</td>
</tr>
<tr>
<td>0769-0824</td>
<td>Disk Parameter Header Tables</td>
</tr>
<tr>
<td>0831-0878</td>
<td>Disk Parameter Blocks</td>
</tr>
<tr>
<td>0881-0907</td>
<td>Other Disk data areas</td>
</tr>
<tr>
<td>0910-0955</td>
<td>SELDSK - Select Disk</td>
</tr>
<tr>
<td>0958-0964</td>
<td>SETTRK - Set Track</td>
</tr>
<tr>
<td>0967-0973</td>
<td>SETSEC - Set Sector</td>
</tr>
<tr>
<td>0978-0984</td>
<td>SETDMA - Set DMA Address</td>
</tr>
<tr>
<td>0987-1025</td>
<td>Sector Skew Tables</td>
</tr>
<tr>
<td>1026-1037</td>
<td>SECTRAN - Logical to Physical Sector translation</td>
</tr>
<tr>
<td>1041-1056</td>
<td>HOME - Home to Track 0</td>
</tr>
<tr>
<td>1059-1154</td>
<td>Debloking Algorithm data areas</td>
</tr>
<tr>
<td>1157-1183</td>
<td>READ - Read 128-byte sector</td>
</tr>
<tr>
<td>1185-1204</td>
<td>WRITE - Write 128-byte sector</td>
</tr>
<tr>
<td>1206-1278</td>
<td>Debloking Algorithm</td>
</tr>
<tr>
<td>1381-1432</td>
<td>Buffer Move</td>
</tr>
<tr>
<td>1435-1478</td>
<td>Debloking subroutines</td>
</tr>
<tr>
<td>1481-1590</td>
<td>8&quot; Floppy Physical Read/Write</td>
</tr>
<tr>
<td>1595-1681</td>
<td>5 1/4&quot; Floppy Physical Read/Write</td>
</tr>
<tr>
<td>1685-1764</td>
<td>WBOOT - Warm Boot</td>
</tr>
</tbody>
</table>
```

**Figure 6-3.** Functional Index to Figure 6-4
Example BIOS

The remainder of this chapter is devoted to an example BIOS listing. This actual working BIOS shows the overall structure and interface to the individual BIOS subroutines.

Unlike most BIOS’s, this one has been written specifically to be understood easily. The variable names are uncharacteristically long and descriptive, and each block of code has commentary to put it into context.

Each source line has been sequentially numbered (an infrequently used option that Digital Research’s Assembler, ASM, permits). Figure 6-3 contains a functional index to the BIOS as a whole so that you can find particular functions in the listing in Figure 6-4 by line number.

```
0001 <= Line Number ; Figure 6-4.
0002
0003******************************************************************************
0004
0005 ; Simple BIOS Listing
0006******************************************************************************
0007
0008
0009
0010 3030 = VERSION EQU '00' ;Equates used in the sign on message
0011 3730 = MONTH EQU '07'
0012 3531 = DAY EQU '15'
0013 3238 = YEAR EQU '82'
0014
0015******************************************************************************
0016
0017 ; This BIOS is for a computer system with the following
0018 ; hardware configuration:
0019
0020 ; - 8080 CPU
0021 ; - 64K bytes of RAM
0022 ; - CRT/keyboard controller that transfers data as though it were a serial port (but requires
0023 ; - no baud rate generator or USART programming).
0024 ; - A serial port, used for both list and "reader"/ "punch" devices. The serial port chip is an
0025 ; - Intel 8251A with an 8253 baud rate generator.
0026 ; - Two 5 1/4" mini-floppy, double-sided, double-density drives. These drives use 512-byte sectors.
0027 ; - These are used as logical disks A1 and B1.
0028 ; - Two 8" standard diskette drives (128-byte sectors).
0029 ; - These are used as logical disks C1 and D1.
0030 ; - Two intelligent disk controllers are used, one for each diskette type. These controllers access memory
0031 ; - directly, both to read the details of the operations they are to perform and also to read
0032 ; - and write data from and to the diskettes.
0033 ; -******************************************************************************
0034
0035 ; Equates for defining memory size and the base address and
0036 ; length of the system components.
```

Figure 6-4. Simple BIOS listing
160  The CP/M Programmer's Handbook

Figure 6-4. (Continued)
Figure 6-4. (Continued)
Figure 6-4. (Continued)
Figure 6-4. (Continued)
Figure 6-4. (Continued)
CALL GetList$Status;Return A = zero or nonzero
; according to status, then convert
; to return parameter convention
ORA A
; If 0, cannot accept data for output
MVI A,OFFH
; Otherwise return A = OFFH to
SET flags to reflect status
RZ
; If Z, cannot accept data for output
DATA A = OFFH TO RET
; indicate can accept data for output

Digital Research's documentation indicates
that you can always return with A = OOH
(Cannot accept data) if you do not wish to
implement the LISTST routine. This is NOT TRUE.
If you do not wish to implement
the LISTST routine,
always return with A = OFFH (Can accept data).
The LIST driver will then take care of things rather
than potentially hanging the system.

Reader input
; Entered directly from BIOS JMP vector,
; inputs the next data character from the reader device
; into the A register

Figure 6-4. (Continued)
The appropriate device is selected according to bits 3,2 of IOBYTE.

; to bits 3,2 of IOBYTE.

LDA IOBYTE

; Get I/O redirection byte

RRC

; Move bits 3,2 to 2,1

CALL SelectRoutine*21

; Select correct READER routine

These routines return directly to READER's caller.

; Entry to low-level driver, HL points to the appropriate control table.

; TerminalTable:

Figure 6-4. (Continued)
The following routines are "called" by SelectRoutine to perform the low-level input/output:

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0576</td>
<td>FBEF 02  DB  Terminal@Data@Port</td>
</tr>
<tr>
<td>0577</td>
<td>FBF0 01  DB  Terminal@Output@Ready</td>
</tr>
<tr>
<td>0578</td>
<td>FBF1 02  DB  Terminal@Input@Ready</td>
</tr>
<tr>
<td>0579</td>
<td></td>
</tr>
<tr>
<td>0580</td>
<td>Communication@Table;</td>
</tr>
<tr>
<td>0581</td>
<td>FBF2 ED  DB  Communication@Status@Port</td>
</tr>
<tr>
<td>0582</td>
<td>FBF3 EC  DB  Communication@Data@Port</td>
</tr>
<tr>
<td>0583</td>
<td>FBF4 01  DB  Communication@Output@Ready</td>
</tr>
<tr>
<td>0584</td>
<td>FBF5 02  DB  Communication@Input@Ready</td>
</tr>
<tr>
<td>0585</td>
<td></td>
</tr>
<tr>
<td>0586</td>
<td></td>
</tr>
<tr>
<td>0587</td>
<td></td>
</tr>
<tr>
<td>0588</td>
<td></td>
</tr>
<tr>
<td>0589</td>
<td></td>
</tr>
<tr>
<td>0590</td>
<td></td>
</tr>
<tr>
<td>0591</td>
<td></td>
</tr>
<tr>
<td>0592</td>
<td>FBF6 21EAF8 LXI H,Teletype@Table</td>
</tr>
<tr>
<td>0593</td>
<td>JMP Input@Status</td>
</tr>
<tr>
<td>0594</td>
<td>Note use of JMP, Input@Status</td>
</tr>
<tr>
<td>0595</td>
<td>will execute the RETURN.</td>
</tr>
<tr>
<td>0596</td>
<td></td>
</tr>
<tr>
<td>0597</td>
<td></td>
</tr>
<tr>
<td>0598</td>
<td></td>
</tr>
<tr>
<td>0599</td>
<td>FBF8C 21EEF8 LXI H,Terminal@Table</td>
</tr>
<tr>
<td>0600</td>
<td>JMP Input@Status</td>
</tr>
<tr>
<td>0601</td>
<td>Note use of JMP, Input@Status</td>
</tr>
<tr>
<td>0602</td>
<td>will execute the RETURN.</td>
</tr>
<tr>
<td>0603</td>
<td></td>
</tr>
<tr>
<td>0604</td>
<td></td>
</tr>
<tr>
<td>0605</td>
<td></td>
</tr>
<tr>
<td>0606</td>
<td></td>
</tr>
<tr>
<td>0607</td>
<td></td>
</tr>
<tr>
<td>0608</td>
<td>F908 3EFF MVI A,OFFH</td>
</tr>
<tr>
<td>0609</td>
<td></td>
</tr>
<tr>
<td>0610</td>
<td></td>
</tr>
<tr>
<td>0611</td>
<td></td>
</tr>
<tr>
<td>0612</td>
<td></td>
</tr>
<tr>
<td>0613</td>
<td>F90B 21EAF8 LXI H,Teletype@Table</td>
</tr>
<tr>
<td>0614</td>
<td>JMP Output@Status</td>
</tr>
<tr>
<td>0615</td>
<td>Note use of JMP, Output@Status</td>
</tr>
<tr>
<td>0616</td>
<td>will execute the RETURN.</td>
</tr>
<tr>
<td>0617</td>
<td></td>
</tr>
<tr>
<td>0618</td>
<td></td>
</tr>
<tr>
<td>0619</td>
<td></td>
</tr>
<tr>
<td>0620</td>
<td></td>
</tr>
<tr>
<td>0621</td>
<td></td>
</tr>
<tr>
<td>0622</td>
<td></td>
</tr>
<tr>
<td>0623</td>
<td></td>
</tr>
<tr>
<td>0624</td>
<td></td>
</tr>
<tr>
<td>0625</td>
<td></td>
</tr>
<tr>
<td>0626</td>
<td></td>
</tr>
<tr>
<td>0627</td>
<td></td>
</tr>
<tr>
<td>0628</td>
<td>F91D 3EFF MVI A,OFFH</td>
</tr>
<tr>
<td>0629</td>
<td></td>
</tr>
<tr>
<td>0630</td>
<td></td>
</tr>
<tr>
<td>0631</td>
<td></td>
</tr>
<tr>
<td>0632</td>
<td></td>
</tr>
<tr>
<td>0633</td>
<td>F920 21EAF8 LXI H,Teletype@Table</td>
</tr>
<tr>
<td>0634</td>
<td>JMP Input@Data</td>
</tr>
<tr>
<td>0635</td>
<td>Note use of JMP, Input@Data</td>
</tr>
<tr>
<td>0636</td>
<td>will execute the RETURN.</td>
</tr>
<tr>
<td>0637</td>
<td></td>
</tr>
<tr>
<td>0638</td>
<td>F926 21EEF8 LXI H,Terminal@Table</td>
</tr>
<tr>
<td>0639</td>
<td>JMP Input@Data</td>
</tr>
<tr>
<td>0640</td>
<td>will execute the RETURN.</td>
</tr>
<tr>
<td>0641</td>
<td></td>
</tr>
<tr>
<td>0642</td>
<td>F92C E67F CALL Input@Data</td>
</tr>
<tr>
<td>0643</td>
<td>ANI 7FH</td>
</tr>
<tr>
<td>0644</td>
<td></td>
</tr>
<tr>
<td>0645</td>
<td></td>
</tr>
<tr>
<td>0646</td>
<td></td>
</tr>
<tr>
<td>0647</td>
<td></td>
</tr>
<tr>
<td>0648</td>
<td></td>
</tr>
<tr>
<td>0649</td>
<td></td>
</tr>
<tr>
<td>0650</td>
<td></td>
</tr>
<tr>
<td>0651</td>
<td>F935 3EIA MVI A,1AH</td>
</tr>
</tbody>
</table>

Figure 6-4. (Continued)
The CP/M Programmer's Handbook

These are the general purpose low-level drivers. On entry, HL points to the appropriate control table. For output, the C register contains the data to be output.

In the use of JMP. Output will execute the RET.

Note use of JMP. Output Data will execute the RET.

Dummy output, always discards.

Set above; Move HL to point to input data mask

Set above; Return with next data character in A.

Save control table pointer

Get input status in zero flag

Recover control table pointer

Wait until incoming data

HL -> data port

Set above; Input Data Port

Input to A from correct data port

F944 21F2F8
LXI H, Teletype Table
JMP Output Data

F947 C370F9
JMP Output Data

Communication Output:

Output Status Port.

DB 00

LXI H, Terminal Table
JMP Output Data

F950 00
DB 00

INX H

INX H

INX H

ANI M

RET

Figure 6-4. (Continued)
### Disk Definition Table

The drivers perform the following functions:

- **SELDISK**: Select a specified disk and return the address of the appropriate disk parameter header.
- **SETTRK**: Set the track number for the next read or write.
- **SECTSEC**: Set the sector number for the next read or write.
- **SECTRAN**: Translate a logical sector number into a physical.
- **HOME**: Set the track to 0 so that the next read or write will be on Track 0.

In addition, the high-level drivers are responsible for making the 5 1/4" floppy diskettes that use a 512-byte sector appear to CP/M as though they used a 128-byte sector. They do this by using what is called blocking/deblocking code, described in more detail later in this listing.

#### Disk Parameter Tables

As discussed in Chapter 3, these describe the physical characteristics of the disk drives. In this example BIOS, there are two types of disk drives: standard single-sided, single-density 8" diskettes, and double-sided, double-density 5 1/4" diskettes.

The standard 8" diskettes do not need to use the blocking/deblocking code, but the 5 1/4" drives do. Therefore an additional byte has been prefixed to the disk parameter block to tell the disk drivers each logical disk's physical diskette type, and whether or not it needs deblocking.

#### Disk Parameter Headers

- **Logical Disk A**: (5 1/4" Diskette)
- **Logical Disk B**: (5 1/4" Diskette)

---

Figure 6-4. (Continued)
Figure 6-4. (Continued)
Disk work areas

These are used by the BDOS to detect any unexpected change of diskettes. The BDOS will automatically set such a changed diskette to read-only status.

Disk allocation vectors

These are used by the BDOS to maintain a bit map of which allocation blocks are used and which are free. One byte is used for eight allocation blocks, hence the expression of the form (allocation blocks/8)+1.

The BDOS will automatically detect any unexpected change of diskette to read-only status.

Figure 6-4. (Continued)
The CP/M Programmer's Handbook

Floppy$e$Skewtable:  .Standard 8"

DB   00,01,02,03 DB 04,05,06,07 DB 08,09,10,11 DB 12,13,14,15 DB 16,17,18,19

Figure 6-4. (Continued)
Data written to or read from the mini-floppy drive is transferred via a physical buffer that is actually 512 bytes long (it was declared at the front of the BIOS and holds the "one-time" initialization code used for the cold boot procedure).

The blocking/deblocking code attempts to minimize the amount of actual disk I/O by storing the disk, track, and physical sector currently residing in the Physical Buffer. If a read request is for a 128-byte CP/M "sector" that already is in the physical buffer, then no disk access occurs.

![Figure 6-4](Continued)
174 The CP/M Programmer’s Handbook

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Operation</th>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FBEA 00</td>
<td>Selected Disk Type</td>
<td>DB 0</td>
<td></td>
<td>Set by SELSDK to indicate either 8&quot; or 5 1/4&quot; floppy</td>
</tr>
<tr>
<td>FBEF 00</td>
<td>Unallocated Disk</td>
<td>DB 0</td>
<td></td>
<td>Nonzero to indicate an error</td>
</tr>
<tr>
<td>FBF0 00</td>
<td>Unallocated Disk Type</td>
<td>DB 0</td>
<td></td>
<td>Indicates 8&quot; or 5 1/4&quot; floppy</td>
</tr>
<tr>
<td>FBF7 00</td>
<td>Disk Error Flags</td>
<td>DB 0</td>
<td></td>
<td>Flags used inside the deblocking code</td>
</tr>
<tr>
<td>FBF8 00</td>
<td>Read Operation</td>
<td>DB 0</td>
<td></td>
<td>Nonzero if a physical sector must be read into the disk buffer</td>
</tr>
<tr>
<td>FBF9 00</td>
<td>Disk Type</td>
<td>DB 0</td>
<td></td>
<td>Indicates 8&quot; or 5 1/4&quot; floppy</td>
</tr>
</tbody>
</table>

Figure 6.4. (Continued)
Chapter 6: The Basic Input/Output System

On arrival here, the BOOS will have set register C to indicate whether this write operation is to an already allocated allocation block (which means a preread of the sector may be needed), to the directory (in which case the data will be written to the disk immediately), or to the first 128-byte sector of a previously unallocated allocation block (in which case no preread is required). Only writes to the directory take place immediately. In all other cases, the data will be moved from the DMA address into the disk buffer, and only written out when circumstances force the transfer. The number of physical disk operations can therefore be reduced considerably.

Figure 6-4. (Continued)
Figure 6-4. (Continued)
Chapter 6: The Basic Input/Output System 177

[Image of text]

Figure 6-4. (Continued)
The controllers are "hard-wired" to monitor certain locations. There are two "smart" disk controllers on this system, one for the 8" floppy diskette drives, and one for the 5 1/4" mini-diskette drives.
have the capability for control tables to be
only has
02H
These are
0043H.
valid
Command Block. If the most significant
OlH
number = 0 or
sector.
BIOS. no further details
back
AND
first byte of
BIOS 3EOI A. Floppy'Read' Code
of disk operations can
the
locat
retry logic -- reads and writes are attempted ten
times before the controller returns an error.
controller returns
The disk control table layout is shown below. Note that the
controllers have the capability for control tables to be
chained together so that a sequence of disk operations can
be initiated. In this BIOS this feature is not used. However,
the controller requires that the chain pointers in the
disk control tables be pointed back to the main control bytes
in order to indicate the end of the chain.

| 0040 = | Disk Control Block | EQU 40H |
| 0041 = | Command Block | EQU 41H |
| 0043 = | Disk Status Block | EQU 43H |
| 0045 = | Disk Control Block | EQU 45H |
| 0046 = | Command Block | EQU 46H |

Floppy Disk Control Tables

```asm
FD40 00 Floppy Command: DB 0 ; Command
FD01 00 Floppy Write Code: DB 0 ; Unit (drive) number = 0 or 1
FD42 00 Floppy Unit: DB 0 ; Head/number = 0 or 1
FD43 00 Floppy Track: DB 0 ; Track number
FD44 00 Floppy Sector: DB 0 ; Sector number
FD45 000 Floppy Byte Count: DW 0 ; Number of bytes to read/write
FD47 000 Floppy DMA Address: DW 0 ; Transfer address
FD49 000 Floppy Next Status Block: DW 0 ; Pointer to next status block
FD4B 000 Floppy Next Control Location: DW 0 ; Pointer to next control byte

Write %0#Deblock: ; Write contents of disk buffer to correct sector.
FD4D 3E02 MVI A, Floppy Write Code ; Get write function code
FD4F C354FD JNP Common %0#Deblock ; Go to common code
FD50 01 %0#Deblock: ; Read previously selected sector
FD52 3E01 MVI A, Floppy Read Code ; Get read function code
FD54 3240FD STA Floppy Command ; Set command function code
FD55 218000 LXI H, 128 ; Set up nondeblocked command table
FD56 2245FD SHLD Floppy Byte Count ; Bytes per sector
FD57 AF XRA A ; 18" floppy only has head 0
FD59 E242FD STA Floppy Head ;
FD5A 3AEAFB LDA Selected Disk ; 8" Floppy controller only has information
FD63 E101 ANI 01H ; Turn into 0 or 1
FD64 E601 STA Floppy Unit ; Set unit number
```

Figure 6-4. (Continued)
The disk controller can accept chained disk control tables, but in this case, they are not used, so the "Next" pointers must be pointed back at the initial control bytes in the base page.

The sector must be converted into a head number and sector number. Sectors 0 - 8 are head 0, 9 - 17 are head 1. Assume head 0.

Note: This is a single byte value.

The sector must be converted into a head number and sector number.
Chapter 6: The Basic Input/Output System

### Disk Control Table Images for Warm Boot

<table>
<thead>
<tr>
<th>Line</th>
<th>Assembly Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1640</td>
<td>FDD0 3C</td>
<td>INR A</td>
</tr>
<tr>
<td>1641</td>
<td>FDD1 3244FD</td>
<td>STA Floppy$Sector</td>
</tr>
<tr>
<td>1642</td>
<td></td>
<td>; (physical sectors start at 1)</td>
</tr>
<tr>
<td>1643</td>
<td>FDD4 210002</td>
<td>LXI H,Physical$Sector$Size</td>
</tr>
<tr>
<td>1644</td>
<td>FDD7 2244FD</td>
<td>SLD Floppy$Byte$Count</td>
</tr>
<tr>
<td>1645</td>
<td>FDDA 213F6</td>
<td>LXI H,Disk$Buffer</td>
</tr>
<tr>
<td>1646</td>
<td>FDDD 2247FD</td>
<td>SLD Floppy$DMA$Address</td>
</tr>
<tr>
<td>1647</td>
<td></td>
<td>; disk buffer</td>
</tr>
<tr>
<td>1648</td>
<td></td>
<td>; As only one control table is in use, close the status and busy chain pointers back to the main control bytes.</td>
</tr>
<tr>
<td>1650</td>
<td></td>
<td>; chain pointers back to the main control bytes.</td>
</tr>
<tr>
<td>1652</td>
<td>FDE0 214300</td>
<td>LXI H,Disk&gt;Status$Block</td>
</tr>
<tr>
<td>1654</td>
<td>FDE3 2249FD</td>
<td>SLD Floppy$Next$Status$Block</td>
</tr>
<tr>
<td>1655</td>
<td>FDE6 214500</td>
<td>LXI H,Disk$Control$5</td>
</tr>
<tr>
<td>1656</td>
<td>FDE9 224BF0</td>
<td>SLD Floppy$Next$Control$Location</td>
</tr>
<tr>
<td>1657</td>
<td>FDEC 2140FD</td>
<td>LXI H,Floppy$Command</td>
</tr>
<tr>
<td>1659</td>
<td>FDF2 2446FD</td>
<td>SLD Command$Block$5</td>
</tr>
<tr>
<td>1661</td>
<td>FDF2 214500</td>
<td>LXI H,Disk$Control$5</td>
</tr>
<tr>
<td>1662</td>
<td>FDF5 3680</td>
<td>MVI M,SOH</td>
</tr>
<tr>
<td>1663</td>
<td></td>
<td>; Wait<em>For</em>Disk*Complete;</td>
</tr>
<tr>
<td>1665</td>
<td></td>
<td>; Wait until Disk Status Block indicates</td>
</tr>
<tr>
<td>1666</td>
<td></td>
<td>; operation complete, then check</td>
</tr>
<tr>
<td>1667</td>
<td></td>
<td>; if any errors occurred,</td>
</tr>
<tr>
<td>1668</td>
<td>FDF7 7E</td>
<td>MOV A,M</td>
</tr>
<tr>
<td>1669</td>
<td>FDF8 87</td>
<td>ORA A</td>
</tr>
<tr>
<td>1670</td>
<td>FDF9 C2F7FD</td>
<td>JNZ Wait<em>For</em>Disk*Complete;</td>
</tr>
<tr>
<td>1671</td>
<td></td>
<td>; Operation still not yet done</td>
</tr>
<tr>
<td>1672</td>
<td>FDFC 3A4300</td>
<td>LDA Disk&gt;Status$Block</td>
</tr>
<tr>
<td>1673</td>
<td>FDFE FE80</td>
<td>CPI BOH</td>
</tr>
<tr>
<td>1675</td>
<td>FE01 D09FE</td>
<td>JC Disk&gt;Error</td>
</tr>
<tr>
<td>1676</td>
<td>FE04 AF</td>
<td>XRA A</td>
</tr>
<tr>
<td>1677</td>
<td>FE08 C9</td>
<td>RET</td>
</tr>
<tr>
<td>1679</td>
<td>FE09 3E01</td>
<td>MVI A,1</td>
</tr>
<tr>
<td>1680</td>
<td>FE0B 3D06FB</td>
<td>STA Disk&gt;Error$Flag</td>
</tr>
<tr>
<td>1681</td>
<td>FE0E C9</td>
<td>RET</td>
</tr>
<tr>
<td>1682</td>
<td></td>
<td>; Boot<em>Control</em>Part1:</td>
</tr>
<tr>
<td>1683</td>
<td></td>
<td>; Boot<em>Control</em>Part2:</td>
</tr>
<tr>
<td>1688</td>
<td>FE0F 01</td>
<td>DB 1</td>
</tr>
<tr>
<td>1689</td>
<td>FE10 00</td>
<td>DB 0</td>
</tr>
<tr>
<td>1690</td>
<td>FE11 00</td>
<td>DB 0</td>
</tr>
<tr>
<td>1691</td>
<td>FE12 00</td>
<td>DB 0</td>
</tr>
<tr>
<td>1692</td>
<td>FE13 02</td>
<td>DB 2</td>
</tr>
<tr>
<td>1693</td>
<td>FE14 0110</td>
<td>DW Bw512</td>
</tr>
<tr>
<td>1694</td>
<td>FE16 0E00</td>
<td>DW CCP$Entry$Entry</td>
</tr>
<tr>
<td>1695</td>
<td>FE18 4300</td>
<td>DW Disk&gt;Status$Block</td>
</tr>
<tr>
<td>1696</td>
<td>FE1A 4500</td>
<td>DW Disk$Control$5</td>
</tr>
<tr>
<td>1697</td>
<td></td>
<td>Boot<em>Control</em>Part1</td>
</tr>
<tr>
<td>1698</td>
<td>FE1C 01</td>
<td>DB 1</td>
</tr>
<tr>
<td>1699</td>
<td>FE1D 00</td>
<td>DB 0</td>
</tr>
<tr>
<td>1700</td>
<td>FE1E 01</td>
<td>DB 1</td>
</tr>
<tr>
<td>1701</td>
<td>FE1F 00</td>
<td>DB 0</td>
</tr>
<tr>
<td>1702</td>
<td>FE20 01</td>
<td>DB 1</td>
</tr>
<tr>
<td>1703</td>
<td>FE21 0006</td>
<td>DW 3w512</td>
</tr>
<tr>
<td>1704</td>
<td>FE23 00F0</td>
<td>DW CCP$Entry$Entry + (8w512)</td>
</tr>
<tr>
<td>1705</td>
<td>FE25 4300</td>
<td>DW Disk&gt;Status$Block</td>
</tr>
<tr>
<td>1706</td>
<td>FE27 4500</td>
<td>DW Disk$Control$5</td>
</tr>
<tr>
<td>1707</td>
<td></td>
<td>Boot<em>Control</em>Part2</td>
</tr>
<tr>
<td>1708</td>
<td></td>
<td>; Boot<em>Control</em>Part1:</td>
</tr>
<tr>
<td>1709</td>
<td></td>
<td>; Boot<em>Control</em>Part2:</td>
</tr>
<tr>
<td>1710</td>
<td></td>
<td>; Boot<em>Control</em>Part1:</td>
</tr>
<tr>
<td>1711</td>
<td>WBOOT:</td>
<td>; Warm boot entry</td>
</tr>
<tr>
<td>1712</td>
<td></td>
<td>; On warm boot, the CCP and BDOS must be reloaded</td>
</tr>
<tr>
<td>1713</td>
<td></td>
<td>; into memory. In this BIOS, only the 5 1/4&quot; diskettes will be used. Therefore this code</td>
</tr>
</tbody>
</table>
The CP/M Programmer's Handbook

The CP/M Programmer's Handbook

WarmSBootSErrorMessage:

FE67 FE6A

DOA~76172 DB CR.LF.'Warm Boot Error - retrying...'.CR.LF,

182

; is hardware specific to the controller. Two
; prefabricated control tables are used.

; Execute first read of warm boot
; Load drive 0, track 0,
; head 0, sectors 2 to 8
; Execute second read
; Load drive 0, track 0,
; head 1, sectors 1 - 3

; Set up base page and enter CCP

; WarmSBootRead:
; On entry, DE -> control table image
; This control table is moved into
; the main disk control table and
; then the controller activated.

; HL -> actual control table
; Tell the controller its address
; Move the control table image
; into the control table itself
; Set byte count

; WarmSBootMove:
; Get image byte
; Store into actual control table
; Update pointers

; Count down on byte count
; Continue until all bytes moved

; Activate controller

; Get status byte
; Check if complete

; Yes, check for errors

; Yes, an error occurred

; Restart warm boot

; Of simple BIOS listing

Figure 6-4. (Continued)
This chapter describes how to build a version of CP/M with your own BIOS built into it. It also shows you how to put CP/M onto a floppy disk and how to write a bootstrap loader to bring CP/M into memory.

The manufacturer of your computer system plays a significant role in building a new CP/M system. Several of CP/M's utility programs may be modified by manufacturers to adapt them to individual computer systems. Unfortunately, not all manufacturers customize these programs. You should therefore invest some time in studying the documentation provided with your system to see what and how much customizing may have already been done. You should also assemble and print out listings of all assembly language source files from your CP/M release diskette.

It is impossible to predict the details of customization and special procedures that the manufacturer may have installed on your particular system. Therefore, this chapter describes first the overall mechanism of building a CP/M system, and
second the details of building a CP/M system around the example BIOS shown in
the previous chapter as Figure 6-4.

### The Major Steps

Building a new CP/M system consists of the following major steps:

- Create a new or modified BIOS with the appropriate device drivers in it. Assemble this so that it will execute at the top end of memory (by using an `origin` statement (ORG) to set the location counter).
- Create new versions of the CCP and BDOS with all addresses in the instructions changed so that they will be correctly located in memory just below the new BIOS. Digital Research provides a special utility called MOVCPM to do this.
- Create or modify a CP/M bootstrap loader that will be loaded by the firmware that executes when you first switch on your computer (or press the `reset` button). Normally, the CP/M bootstrap loader executes in the low-address end of memory. The exact address and the details of any hardware initialization that it must perform will depend entirely on your particular computer system.
- Using Digital Research standard utility programs, bring the bootstrap loader, the CCP and BDOS, and the BIOS together in the low part of memory. Then write this new version of CP/M onto a disk in the appropriate places. Again, depending on the design of your computer system, you may be able to use the standard utility program, SYSGEN, to write the entire CP/M `image` onto disk. Otherwise you may have to write a special program to do this.

When CP/M is already running on your computer system and you want to add new features to the BIOS, all you need to do is change the BIOS and rebuild the system. The CCP and BDOS will need to be moved down in memory if the changes expand the BIOS significantly. If this happens, you will have to make minor changes in the bootstrap loader so that it reads the new CP/M image into memory at a lower address and transfers control to the correct location (the first instruction of the BIOS jump vector).

### Building Your First System

The first time that you build CP/M, it is a good idea to make no changes to the BIOS at all. Simply reassemble the BIOS source code and proceed with the system build. Then, if the new system does not run, you know that it must be something in the procedure you used rather than any new features or modification to the BIOS
source code. Changes in the BIOS could easily obscure any problems you have with the build procedure itself.

The Ingredients

To build CP/M, you will need the following files and utility programs:

- The assembly language source code for your BIOS. Check your CP/M release diskette for a file with a name like CBIOS.ASM (Customized Basic Input/Output System). Some manufacturers do not supply you with the source code for their BIOS; it may be sold separately or not released at all. If you cannot get hold of the source code, the only way that you can add new features to the BIOS is by writing the entire BIOS from scratch.

- The source code for the CP/M bootstrap loader. This too may be on the release diskette or available separately from your computer's manufacturer.

- The Digital Research assembler, which converts source code into machine language in hexadecimal form. This program, called ASM.COM, will be on your CP/M release diskette. Equivalent assemblers, such as Digital Research's macro-assemblers MAC and RMAC or Microsoft's M80, can also be used.

- The Digital Research utility called MOVCPM, which prepares a memory image of the CCP and BDOS with all addresses adjusted to the right values.

- The Digital Research debugging utility, called DDT (Dynamic Debugging Tool), or the more enhanced version for the Z80 CPU chip, ZSID (Z80 Symbolic Interactive Debugger). DDT is used to read in the various program files and piece together a memory image of the CP/M system.

- The Digital Research utility program SYSGEN. This writes the composite memory image of the bootstrap, CCP, BDOS, and BIOS onto the disk. SYSGEN was designed to work on floppy disk systems. If your computer uses a hard disk, you may have a program with a name like PUTCPM or WRITECPM that performs the same function.

The Ultimate Goal

In Figure 6-4, lines 0044 to 0065, you can see the equates that define the base addresses for the CCP, the BDOS, and the BIOS. Figure 7-1 shows how the top of memory will look when this version of CP/M has been loaded into memory.

Life would be simple if you could build this image in memory at the addresses shown and write the image out to disk. Building this image, however, would probably overwrite the version of CP/M that you were operating since it too lives at the top of memory. Therefore, the goal is to create a replica of this image lower down in memory, but with all the instruction addresses set to *execute* at the addresses shown in Figure 7-1.
Using SYSGEN to Write CP/M to Disk

The SYSGEN utility writes a memory image onto a specified logical disk. It can use a memory image that you arrange to be in memory before you invoke SYSGEN, or you can direct SYSGEN to read in a disk file that contains the image. You can also use SYSGEN to transport an existing CP/M system from one diskette to another by directing it to load the CP/M image from one diskette into memory and then to write that image out to another diskette.

Check the documentation supplied by your computer’s manufacturer to make sure that you can use SYSGEN on your system. SYSGEN, as released by Digital Research, is constructed to run on 8-inch, single-sided, single-density diskettes. If your system does not use these standard diskettes, SYSGEN must be customized to your disk system.

When SYSGEN loads a CP/M image into memory, it will place the bootstrap, CCP, BDOS, and BIOS at the predetermined addresses shown in Figure 7-2, regardless of where this CP/M originated.
Chapter 7: Building a New CP/M System

Currently executing version of CP/M

$0E400H$ (approximate)

BIOS = 2304 (900H) bytes (this will vary from version to version)

$0F80H$

BDOS = 3584 (0E00H) bytes

$1180H$

CCP = 2048 (800H) bytes

$0980H$

Bootstrap = 128 (80H) bytes

$0900H$

SYSGEN = xxx (xxxH) bytes

$0100H$

$0000H$

Figure 7-2. SYSGEN's memory layout
You can see that the *relative* arrangement between the components has not changed; the whole image has simply been moved down in memory well below the currently executing version of CP/M. The bootstrap has been added to the picture just beneath the CCP.

The SYSGEN utility writes this image onto a floppy diskette starting at sector 1 of track 0 and continuing to sector 26 on track 1. Refer back to Figure 2-2 to see the layout of CP/M on a standard 8-inch, single-sided, single-density diskette.

If you request SYSGEN to read the memory image from a file (which you do by calling SYSGEN with the file name on the same line as the SYSGEN call), then SYSGEN presumes that you have previously created the correct memory image and saved it (with the SAVE command). SYSGEN then skips over the first 16 sectors of the file so as to avoid overwriting itself.

Here is an example of how to use SYSGEN to move the CP/M image from one diskette to another:

```
A>SYSGEN<CR>
SYSGEN VER 2.0
SOURCE DRIVE NAME (OR RETURN TO SKIP) A
SOURCE ON A: THEN TYPE RETURN <cr>
FUNCTION COMPLETE
DESTINATION DRIVE NAME (OR RETURN TO REBOOT) B
DESTINATION ON B: THEN TYPE RETURN <cr>
FUNCTION COMPLETE
DESTINATION DRIVE NAME (OR RETURN TO REBOOT)<cr>
A>
```

As you can see, SYSGEN gives you the choice of specifying the source drive name or typing CARRIAGE RETURN. If you enter a CARRIAGE RETURN, SYSGEN assumes that the CP/M image is already in memory. Note that you need to call up SYSGEN only once to write out the same CP/M image to more than one disk.

A larger than standard BIOS can cause difficulties in using SYSGEN. The standard SYSGEN format only allows for six 128-byte sectors to contain the BIOS, so if your BIOS is larger than 768 (300H) bytes, it will be a problem. The CP/M image will not fit on the first two tracks of a standard 8-inch diskette.

Nowadays it is rare to find an 8-inch floppy diskette system where you must load CP/M from a single-sided, single-density diskette. Most systems now use double-sided or double-density diskettes as the normal format, but can switch to single-sided, single-density diskettes to interchange information with other computer systems.

Because there is no "standard" format for 8-inch, double-sided and double-density diskettes, you probably won't be able to read diskettes written on systems of a different make or model. Therefore, you need only be concerned about using a disk layout that will keep your disks compatible with other machines that are exactly the same as yours.

This is also true if you have 5 1/4-inch diskettes. There is no industry standard for these either, so your main consideration is to place the file directory in the same
place as it will be on diskettes written by other users of your model of computer. You must also be sure to use the same sector skewing. Otherwise, you will get a garbled version whenever you try to read files originating on other systems.

With the higher capacity diskettes, you can reserve more space to hold the CP/M image on the diskette. For example, in the case of the BIOS shown in Figure 6-4, the CP/M image is written to a 5 1/4-inch, double-sided, double-density diskette using 512-byte sectors. Figure 7-3 shows the layout of this diskette. Note that the bootstrap loader is placed in a 512-byte sector all by itself. Doing so makes the bootstrap code and warm boot code in the BIOS much simpler.

The memory image must be altered to reflect the fact that the bootstrap now occupies an entire 512-byte sector. Rather than change all of the addresses, the bootstrap is loaded into memory 384 (180H) bytes lower, so that it ends at the same address as before. Figure 7-4 shows the revised memory image.

**Writing a PUTCPM Utility**

Because the example system uses 5 1/4-inch floppy diskettes with 512-byte sectors, the standard version of SYSGEN cannot be used to write the CP/M image onto a diskette. You will have to use a functional replacement provided by your computer's manufacturer or develop a small utility program to do the job.

![Figure 7-3. Disk layout for example BIOS on 5 1/4-inch diskettes](image-url)
Figure 7-4. Addresses for example BIOS image

Figure 7-5 shows an example of such a program. It is written in a general-purpose way, so that you may be able to use it for your system by changing the equates at the front of the program to reflect the specifics of your disk drives.

Note that there are two problems to be solved. First, the area of the disk on which the CP/M image resides cannot be accessed by the BDOS, as it is outside the file system area on the disk. Second, it is rare to write the CP/M image onto the disk with any kind of sector skewing; to do so would slow down the loading process. In any case, skewing would be redundant, since the loader is doing no processing other than reading the disk and can therefore read the disk without skewing.
Chapter 7: Building a New CP/M System

This program writes out the CP/M cold boot loader, CCP, BDOS, and BIOS to a floppy diskette. It runs under CP/M as a normal transient program.

| 3130 | Version         | EQU '01' |
| 3730 | Month           | EQU '07' |
| 3432 | Day             | EQU '24' |
| 3238 | Year            | EQU '82' |

The actual PUTCPMF5.COM program consists of this code, plus the BOOTF5.HEX, CCP, BDOS, and BIOS.

When this program executes, the memory image should look like this:

<table>
<thead>
<tr>
<th>Component</th>
<th>Base Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIOS</td>
<td>IF80H</td>
</tr>
<tr>
<td>BDOS</td>
<td>1180H</td>
</tr>
<tr>
<td>CCP</td>
<td>0980H</td>
</tr>
<tr>
<td>BOOS</td>
<td>0780H</td>
</tr>
</tbody>
</table>

The components are produced as follows:

- BIOS.HEX: By assembling source code
- BDOS: From a CPMinn.COM file output
- CCP: By MOVCPM and SAVEd on disk
- BOOTF5.HEX: By assembling source code

The components are pieced together using DDT with the following commands:

```
DDT CPMinn.COM
IPUTCPMF5.HEX R (Reads in this program)
IBOOTF5.HEX R680 (Reads in BOOT at 0780H)
IBIOS.HEX R2980 (Reads in BIOS at IF80H)
GO 880 (Exit from DDT)
SAVE 40 PUTCPMF5.COM (Create final .COM file)
```

The actual layout of the diskette is as follows:

<table>
<thead>
<tr>
<th>Track 0</th>
<th>Sector</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1 2 3 4 5 6 7 8 9</td>
</tr>
<tr>
<td>Head</td>
<td>+-------------------</td>
</tr>
<tr>
<td>0</td>
<td>Boot</td>
</tr>
<tr>
<td></td>
<td>+-------------------</td>
</tr>
<tr>
<td>1</td>
<td>BDOS</td>
</tr>
<tr>
<td></td>
<td>+-------------------</td>
</tr>
<tr>
<td>10</td>
<td>11 12 13 14 15 16 17 18</td>
</tr>
</tbody>
</table>

Equates for defining memory size and the base address and length of the system components:

- Memory$Size EQU 64 ;Number of Kbytes of RAM
- The BIOS Length must match that declared in the BIOS.
- 0900 = BIOS$Length EQU 0900H
- 0200 = Boot$Length EQU 512
- 0800 = CCP$Length EQU 0800H ;Constant
- 0E00 = BDOS$Length EQU 0E00H ;Constant
- 1F00 = Length$In$Bytes EQU CCP$Length + BDOS$Length + BIOS$Length
- 0780 = Start$Image EQU 980H - Boot$Length ;Address of CP/M image
- 2100 = Length$Image EQU Length$In$Bytes + Boot$Length

Figure 7-5. Example PUTCPM
Disk characteristics

These equates describe the physical characteristics of the floppy diskette so that the program can move from one sector to the next, updating the track and resetting the sector when necessary.

Controller characteristics

On this computer system, the floppy disk controller can write multiple sectors in a single command. However, in order to produce a more general example it is shown only reading one sector at a time.

Cold boot characteristics

Cold boot characteristics

Data structures are shown before the executable code.

Disk control tables

The command table track and DMA Address can also be used as working storage and updated as the load process continues. The sector in the command table cannot be used directly as the disk controller requires it to be the sector number on the specified head (1-9) rather than the sector number on track. Hence a separate variable must be used.

Figure 7-5. (Continued)
Chapter 7: Building a New CP/M System

Figure 7-5. (Continued)
DDT, the Digital Research debug program, is used to read files of type ".COM" and ".HEX" into memory. Understanding the internal structure of these file types is important, both to understand what DDT can do and to understand how the MOVCPM utility can effectively change a machine code file so that it can be executed at a new address in memory.

".COM" File Structure

A COM file is a memory image. It is a replica of the bit patterns that are to be created when the file is loaded into memory. COM files are normally designed to load at location 100H upwards. No internal structure to the file requires this, however, so if you know what the contents of a COM file are, there is nothing to preclude you from loading it into memory starting at some address other than 100H.

As you may recall from the description of the CCP in Chapter 4, the SAVE command built into the CCP allows you to create a COM file by specifying the number of 256-byte "pages" of memory and the name of the file. The CCP will write out an exact image of memory from location 100H up.
".HEX" File Structure

HEX files are output by the assembler. They contain an ASCII character representation of hexadecimal values. For example, the contents of a single byte of memory with the binary value 10101111 would be represented by two ASCII characters, A F, in a HEX file.

The HEX file has a higher level structure than just a series of ASCII characters however. Each line of ASCII characters is terminated by CARRIAGE RETURN/LINE FEED. The overall structure is shown in Figure 7-6.

The most important aspect of a HEX file is that each line contains the address at which the data bytes are loaded. Each line is processed independently, so the load addresses of succeeding lines need not be in order.

DDT can read in a HEX file at an address different from the address where the code must be in order to execute. For example, you can read in the HEX file of the BIOS at the correct place for the memory image (shown in Figure 7-4). There are two ways of using DDT to read in a COM or HEX file. You can specify the name of the file on the same command line with DDT. For example:

```
A>DDT B:XYZ.HEX<cr>
```

The advantage of this method of loading a file is that you can specify which logical disk is to be searched for the file. The second way of using DDT is to load DDT first, and then, when it has given its prompt, specify the file name and request that DDT load it like this:

```
-Ifilename.typ<cr> <- Enter the file name and type
-R<cr> <- Read in the file
```

The "I" command initializes the default file control block in the base page (at location 005CH) with the file name and type; it does not set up the logical disk. If you need to do this, you must set the first byte of the default FCB manually like this:

```
-Ifilename.typ<cr> <- Specify file name
-S5C<cr> <- "S"et location 5C
005C 00 02<cr> <- Was 00, you enter 02<cr>
005D 41 .<cr> <- Enter "." to terminate
-R<cr> <- Read in the file
```

Location 005CH should be set to 01H for Drive A, 02H for B, and so on.

The "R" command will read in HEX files to the execution addresses specified in each line of the HEX file, so be careful—if you forget to put an ORG (origin)
The CP/M Programmer's Handbook

The Check sum formed by adding up all of the values 04, 01, 58, 00, 64, 00, 01 and 80 and then subtracting their sum from OOH

Data bytes to be loaded at the specified address

Record (line) type, normally 00

Load address for the data bytes on this line

Number of data bytes on this line (ASM uses 10H bytes)

Beginning of line marker (colon)

NOTE: HEX files do not have embedded blank characters; the example above is shown with gaps between individual fields only for clarity.

Figure 7-6. Example line from HEX file

statement at the front of the assembly language source code, reading in the resultant HEX file will overwrite location 0000H on up, destroying the contents of the base page. Similarly, if you were trying to read in the HEX file for a BIOS, there is an excellent chance that you will overwrite the currently executing CP/M system.

DDT reacts to the file type you enter as part of the file name. For file types other than .HEX, DDT loads the file starting at location 0100H on up.

The "R" command can also be used to read files into memory at different addresses. You do this by typing a hexadecimal number immediately after the R, with no intervening punctuation. For HEX files, the number that you enter is added to the address in each line of the HEX file and the sum is used as the address into which the data bytes are loaded. The data bytes themselves are not changed, just the load address.

For COM files, the number that you enter is added to 0100H and the sum is used as the starting address for loading the file.

The sum is performed as 16-bit, unsigned arithmetic with any carry ignored, so you can load a BIOS HEX file into low memory by using the "R" command with what is called an "offset value."

If a HEX file has been assembled to execute at address "exec," and you need to use DDT to read in this file to address "load," you need to solve the following equation:

\[ \text{offset} = \text{load} - \text{exec} \]

DDT's "H" command performs hexadecimal arithmetic. It calculates and displays the sum of and difference between two hexadecimal values. For example,
Chapter 7: Building a New CP/M System

the BIOS in Figure 6-4 has been assembled to execute at location 0F600H, but needs to be loaded into memory at location 1F80H. Here is how to compute the correct offset for the "R" command:

```
-H1F80,F600<cr>
1580,2980
```

Thus, to read in the BIOS HEX file called FIG6-4.HEX at location 1F80H, you would enter the following commands to DDT:

```
-FIG6-4.HEX<cr>
-R2980<cr>
```

In this way, using DDT, you can read in the HEX files for both the BIOS and the bootstrap loader.

---

**The CP/M Bootstrap Loader**

The bootstrap loader is brought into memory by PROM-based firmware in the computer system. It loads in the CCP, BDOS, and BIOS and then transfers control to the cold boot entry point in the BIOS—the first jump instruction in the BIOS jump vector.

The bootstrap loader is a stand-alone program; it cannot make use of any CP/M functions because no part of CP/M is in memory when the bootstrap loader is needed. The firmware in the PROM that loaded the bootstrap may contain some subroutines that can be used by the bootstrap, but this will vary from system to system.

Figure 7-7 shows the bootstrap code for the example BIOS (from Figure 6-4). This code has been written in a general way, so that you can adapt it to your system. The disk controller on the example system can in fact read in multiple sectors from the disk, but for generality the code shown reads in only one sector at a time. This considerably increases the time it takes to load CP/M, but does make the bootstrap loader more general.

Note that almost the first thing that the bootstrap does is to output to the console a sign-on message. Not only does this confirm the version number, but it shows that the bootstrap has been successfully loaded.

The PROM-based code has been designed to load the CP/M bootstrap into location 100H, allowing the code to be debugged as though it were a normal transient program, albeit with minor changes to the address at which it loads the CP/M image from disk. Clearly, this feature is not very helpful if CP/M is being brought up for the first time on a computer system. It helps a great deal, however, if you need to modify the bootstrap or add the capability to boot your system from a new type of disk drive.
Example CP/M cold bootstrap loader

This program is written out to track 0, head 0, sector 1 by the PUTCPMF5 program. It is loaded into memory at location 100H on up by the PROM-based bootstrap mechanism that gets control of the CPU on power up or system reset.

Version EQU '01'; Equates used in the sign-on message
Month EQU '07'
Day EQU '24'
Year EQU '82'

Debug EQU 0; Set nonzero to debug as normal

The actual layout of the diskette is as follows:

Track 0 Sector 1 2 3 4 5 6 7 8 9
Head +--------------------------+
0 |Boot|...............|CCP|...............|BDOS|...............|
1 |.............|BIOS|...............|
10 |11|12|13|14|15|16|17|18|

Equates for defining memory size and the base address and length of the system components.

MemorySize EQU 64; Number of Kbytes of RAM

The BIOS Length must match that declared in the BIOS.

0900 = BIOS$Length EQU 0900H
0800 = CCP$Length EQU 0800H; Constant
0E00 = BDOS$Length EQU 0E00H; Constant

Length$In$K EQU (CCP$Length + BDOS$Length + BIOS$Length) / 1024 + 1
Length$In$Bytes EQU CCP$Length + BDOS$Length + BIOS$Length

IF NOT Debug
ENDIF
IF Debug
CCP$Entry EQU 3980H; Read into a lower address.

This address is chosen to be above the area into which DDT initially loads and the 980H makes the addresses similar to the SYSGEN values so that the memory image can be checked with DDT.

ENDIF

Disk characteristics

These equates describe the physical characteristics of the floppy diskette so that the program can move from one sector to the next, updating the track and resetting the sector when necessary.

First$Sector$on$Track EQU 1
Last$Sector$on$Track EQU 18
Last$Sector$on$Head$0 EQU 9
Sector$Size EQU 512

Controller characteristics

Figure 7-7. Example CP/M cold bootstrap loader
Chapter 7: Building a New CP/M System

On this computer system, the floppy disk controller can read multiple sectors in a single command. However, in order to produce a more general example it is shown only reading one sector at a time.

0001 = Sectors$Per$Read EQU 1

Cold boot characteristics

0000 = Start$Track EQU 0 ;Initial values for CP/M image
0002 = Start$Sector EQU 2
0010 = Sectors$To$Read EQU (Length$In$Bytes + Sector$Size - 1) / Sector$Size

Cold Boot Loader:

0100 DDRO 100H
0100 C34001 JMP Main$Code ;Enter main code body
00DD = CR EQU ODH ;Carriage return
00DA = LF EQU OAH ;Line feed

Signon$Message:

0103 000A43502F DB CR,LF,"CP/M Bootstrap Loader"
0105 01 DB Debug
0106 02 DB (Debug)'
0107 ENDF
0110 000A DB CR,LF
0111 5665727369 DB 'Version '
0112 3033 DW Version
0114 20 DB /
0115 27 DW Month
0116 3037 DB /
0117 3234 DW Day
0119 2F DB /
0120 3532 DW Year
0122 0000 DB CR,LF,0

Disk Control Tables

0045 = Disk$Control$5 EQU 45H ;5 1/4" control byte
0046 = Command$Block$5 EQU 46H ;Command table pointer
0043 = Disk$Status EQU 43H ;Completion status

The command table track and DMA$Address can also be used as working storage and updated as the load process continues. The sector in the command table cannot be used directly as the disk controller requires it to be the sector number on the specified head (1 -- 9) rather than the sector number on track. Hence a separate variable must be used.

0132 02 Sector: DB Start$Sector
0133 01 Command$Table: DB 01H ;Command -- read
0134 00 Unit: DB 0 ;Unit (drive) number = 0 or 1
0135 00 Head: DB 0 ;Head number = 0 or 1
0136 00 Track: DB Start$Track ;Used as working variable
0137 00 Sector$On$Head: DB 0 ;Converted by low-level driver
0138 0002 Byte$Count: DW Sector$Size * Sectors$Per$Read
0139 0000 DMA$Address: CCP$Entry
013A 4300 Next$Status: DW Disk$Status ;Pointer to next status block
013B 4300 Next$Control: DW Disk$Control$5 ;Pointer to next control byte
013E 4500 ;if commands are chained.

Main$Code:

0140 310001 LXI SP,Cold$Boot$Loader ;Stack grows down below code

Figure 7-7. (Continued)
Figure 7.7. (Continued)
Chapter 7: Building a New CP/M System

Figure 7-7. (Continued)

In this case, the bootstrap code must be loaded at location 0780H, not the normal 0980H, because the bootstrap takes a complete 512-byte sector (200H). The same principle applies in determining the offset value to be used with DDT’s “R” command to read the bootstrap HEX file, namely:

offset = load address − execution address.

In this case, the values are the following:

0680H = 0780H − 0100H

Using MOVCPM to Relocate the CCP and BDOS

MOVCPM builds a CP/M memory image at the correct locations for SYSGEN, but with the instructions modified to execute at a specific address. Inside MOVCPM is not only a complete replica of CP/M, but also enough
information to tell MOVCPM which bytes of which instructions need be changed whenever the execution address of the image needs to be moved.

MOVCPM, as released from Digital Research, contains the bootstrap and BIOS for an Intel MDS-800 computer along with the generic CCP and BDOS. Unless you have an MDS-800, all you use is the CCP and BDOS. Some manufacturers have customized MOVCPM to include the correct bootstrap and BIOS for their own computers; consult their documentation to see if this applies to your computer system.

When you invoke MOVCPM, you have the following options:

- **MOVCPM<cr>**
  MOVCPM will relocate its built-in copy of CP/M to the top of available memory and will then transfer control to this new image of CP/M. Unless your manufacturer has included the correct BIOS into MOVCPM, using this option will cause an immediate system crash.

- **MOVCPM nn<cr>**
  This is similar to the option above, except that MOVCPM assumes that $nnK$ bytes of memory are available and will relocate the CP/M image to the top of that before transferring control. Again, this will crash the system unless the correct BIOS has been installed into MOVCPM.

- **MOVCPM * * <cr>**
  MOVCPM will adjust all of the internal addresses inside the CP/M image so that the image could execute at the top of available memory, but instead of actually putting this image at the top of memory, MOVCPM will leave it in low memory at the correct place for SYSGEN to write it onto a disk. The SAVE command could also preserve the image on a disk.

- **MOVCPM nn *<cr>**
  MOVCPM proceeds as above for the "* *" option except that the CP/M image is modified to execute at the top of $nnK$.

MOVCPM has a fundamental problem. The $nn$ value indicates that the top of available memory is computed, assuming that your BIOS is small—less that 890 (380H) bytes. If your BIOS is larger (as is the case with the example in Figure 6-4), then you will have to reduce the value of "$nn$" artificially.

Figure 7-8 shows the relationship between the size of the BIOS and the "$nn$" value to use with MOVCPM. It also shows, for different lengths of BIOS, the BIOS base address, the offset value to be used in DDT to read in the BIOS to location 1F80H (preparatory to using SYSGEN or PUTCPM to write it out), and also the base addresses for the CCP and the BDOS. The base address of the BDOS indicates how much memory is available for loading transient programs, as the CCP can be overwritten if necessary.

The numbers in Figure 7-8 are based on the assumption that you have 64K of memory in your computer system. If this is not the case, then proceed as follows:
Chapter 7: Building a New CP/M System

1. Convert the amount of memory in your system to hex. Remember that 1K is 1024 bytes.
2. Determine the length of your BIOS in hex.
3. Locate the line in Figure 7-8 that shows a BIOS length equal to or greater than the length of your BIOS.
4. Using the “H” command in DDT, compute the BIOS Base Address using the formula:
   \[ \text{Memory in system} - \text{BIOS length from Figure 7-8} \]
5. Find the line in Figure 7-8 that shows the same BIOS Base Address as the result of the computation above. Use this line to derive the other relevant numbers.

It is helpful to use DDT to examine a CP/M image in memory to check that all of the components are correctly placed, and, in the case of the CCP and BDOS, correctly relocated.

Figure 7-9 shows an example console dialog in which DDT is used first to examine the memory image produced by MOVCPM and second to examine the image built into the PUTCPMF utility shown in Figure 7-5.

<table>
<thead>
<tr>
<th>BIOS Length</th>
<th>BIOS Base</th>
<th>DDT Offset</th>
<th>MOVCPM ‘nn’</th>
<th>CCP Base</th>
<th>BDOS Base</th>
</tr>
</thead>
<tbody>
<tr>
<td>600</td>
<td>FA00</td>
<td>2580</td>
<td>64</td>
<td>E400</td>
<td>E000</td>
</tr>
<tr>
<td>300</td>
<td>E400</td>
<td>2800</td>
<td>63</td>
<td>E000</td>
<td>E800</td>
</tr>
<tr>
<td>1200</td>
<td>EE00</td>
<td>3180</td>
<td>62</td>
<td>D000</td>
<td>E400</td>
</tr>
<tr>
<td>1600</td>
<td>EA00</td>
<td>3580</td>
<td>61</td>
<td>D800</td>
<td>E000</td>
</tr>
<tr>
<td>1E00</td>
<td>E200</td>
<td>3980</td>
<td>59</td>
<td>D400</td>
<td>DC00</td>
</tr>
<tr>
<td>12200</td>
<td>DE00</td>
<td>4180</td>
<td>58</td>
<td>C800</td>
<td>D400</td>
</tr>
<tr>
<td>12600</td>
<td>DA00</td>
<td>4580</td>
<td>57</td>
<td>C800</td>
<td>D000</td>
</tr>
<tr>
<td>12E00</td>
<td>D200</td>
<td>4980</td>
<td>56</td>
<td>C200</td>
<td>CC00</td>
</tr>
<tr>
<td>3200</td>
<td>CE00</td>
<td>5180</td>
<td>55</td>
<td>B400</td>
<td>CC00</td>
</tr>
<tr>
<td>3600</td>
<td>CA00</td>
<td>5580</td>
<td>54</td>
<td>B800</td>
<td>BC00</td>
</tr>
<tr>
<td>3A00</td>
<td>C600</td>
<td>5980</td>
<td>53</td>
<td>C000</td>
<td>C400</td>
</tr>
<tr>
<td>3E00</td>
<td>C200</td>
<td>5D80</td>
<td>52</td>
<td>B000</td>
<td>B000</td>
</tr>
<tr>
<td>4200</td>
<td>BE00</td>
<td>6180</td>
<td>51</td>
<td>A800</td>
<td>B400</td>
</tr>
<tr>
<td>4600</td>
<td>BA00</td>
<td>6580</td>
<td>50</td>
<td>A000</td>
<td>B000</td>
</tr>
<tr>
<td>4A00</td>
<td>B600</td>
<td>6780</td>
<td>49</td>
<td>A400</td>
<td>A000</td>
</tr>
<tr>
<td>4E00</td>
<td>B200</td>
<td>6980</td>
<td>48</td>
<td>A400</td>
<td>A000</td>
</tr>
<tr>
<td>5200</td>
<td>AE00</td>
<td>7180</td>
<td>47</td>
<td>A000</td>
<td>A800</td>
</tr>
<tr>
<td>5600</td>
<td>A600</td>
<td>7380</td>
<td>46</td>
<td>9000</td>
<td>9000</td>
</tr>
<tr>
<td>5A00</td>
<td>A200</td>
<td>7580</td>
<td>45</td>
<td>8800</td>
<td>9800</td>
</tr>
<tr>
<td>5E00</td>
<td>9E00</td>
<td>7880</td>
<td>44</td>
<td>8000</td>
<td>9000</td>
</tr>
<tr>
<td>6200</td>
<td>9A00</td>
<td>8180</td>
<td>43</td>
<td>8400</td>
<td>8C00</td>
</tr>
<tr>
<td>6600</td>
<td>9F00</td>
<td>8580</td>
<td>42</td>
<td>8800</td>
<td>8800</td>
</tr>
<tr>
<td>6A00</td>
<td>9C00</td>
<td>8980</td>
<td>41</td>
<td>8000</td>
<td>8800</td>
</tr>
</tbody>
</table>

Apart from the MOVCPM ‘nn’ value all other values are in hexadecimal

Figure 7-8. CP/M addresses for different BIOS lengths
Call up MOVCPM requesting a ‘63K’ system and the image to be left in memory.

CONSTRUCTING 63K CP/M vers 2.2
READY FOR “SYSGEN” OR 
“SAVE 34 CPMS3.COM”

Save the image from location 100H UP. By convention, the file name is CPMMn.COM, so in this case it will be CPMS3.COM

A>Movecpm 63 <cr>
A>Save 34 cpms3.com<cr>
A>dtt cpms3.com<cr>

Display memory to show the first few bytes of the CCP. Note the two JMP (C3H) instructions, followed by 7FH, OOH, 2OH’s, and the Digital Research Copyright notice. These identify the code as being the CCP. Note that the first JMP instruction is to 35CH into the CCP -- you can therefore infer the base address of the CCP. In this case the JMP is to location E35C, therefore this version of the CCP has been configured to execute based at E000H.

Display the first few bytes of the BDOS. Note the JMP instruction at 1186. This is the instruction to which control is transferred by the JMP in location 5.

Displaying further up in the BDOS identifies it unambiguously -- there are some ASCII error messages.

Display the first few bytes of the BIOS. Notice the BIOS JMP vector -- the series of C3H instructions. Normally the first instruction in the vector can be used to infer the base address of the BIOS; in this case it is F600H. But there is no rule that says that the cold boot code must be close to the BIOS JMP vector -- so this is only a rough guide.

Figure 7-9. Using DDT to check CP/M images
In contrast, load DDT and request that it load the PUTCPMF5.COM program.

A>dtt_putcpmf5.com
DDT VERS 2.2
NEXT PC
2900 0100

Display the special bootstrap loader that starts at location 0780H (compared to the MDS-800 bootstrap which is at 0980H). Note the sign-on message.

-780.7af

0780 C3 40 01 00 0A 43 50 2F 4D 20 42 6F 6F 74 73 74 .8...CP/M Bootst
0790 72 61 70 20 4C 6F 61 64 65 72 00 0A 56 65 73 74 73 rap Loader..Vers
07A0 69 6F 6E 20 30 31 20 37 32 35 20 32 34 2F 38 32 00 01 01 07/24/82.

Confirm that the CCP is loaded in the correct place. Check the address of the first JMP instruction (0E35CH).

-780.9bf

0980 C3 5C E3 C3 58 E3 7F 00 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 ...

1180 00 16 00 00 09 85 C1 11 E8 99 E8 A5 E8 AB E8 B1 ................

Confirm that the BIOS is also in place.

-1180.11bf

IFSO 37 2F 31 35 2F 38 32 OD OA OA 53 69 6D 70 6C 65 7/15/82 ...Simple
IFSO 38 00 00 43 50 2F 4D 20 32 2E 32 2E 30 30 20 30 8 ...CP/M 2.2.00 0
IFSO 37 2F 31 35 2F 38 32 00 0A 0A 53 69 6D 70 6C 65 7/15/82 ...Simple

Using DDT to check CP/M images (continued)

Figure 7-9

Putting it all Together

Figure 7-10 shows an annotated console dialog for the complete generation of a new CP/M system. Note that the following file names appear in the dialog:

BIOS1.ASM
PUTCPMF5.ASM
BOOTF5.ASM

Figure 6-4
Figure 7-5
Figure 7-7
Figure 7-10. Console dialog for system build

C:\<assembler5.exe> (cr)
CP/M ASSEMBLER - VER 2.0
02E4
004H USE FACTOR
END OF ASSEMBLY

Assemble the CP/M Bootstrap Loader, with the source code and HEX file on drive C:, no listing output.

C:\<assembler5.exe> (cr)
CP/M ASSEMBLER - VER 2.0
01DB
003H USE FACTOR
END OF ASSEMBLY

Assemble the PUTCPMFS program (that writes CP/M onto the disk), with the source code and HEX file on drive C:, no listing output.

C:\<assembler5.exe> (cr)
CP/M ASSEMBLER - VER 2.0
0E6C
011H USE FACTOR
END OF ASSEMBLY

Assemble the BIOS with the source code and HEX file on drive C:, no listing output.

Start piecing the CP/M image together. Load DDT and ask it to read in the file previously SAVED after a MOVCPM 63.

C:\ddt cpm63.com (cr)
DDT VERS 2.2
NEXT PC
2300 0100

Indicate the file name of PUTCPMFS.HEX, and read in without any offset (i.e., it will load at 100H because of the ORG 100H it contains). -iputcpmfs.hex (cr)

C:\ddt cpm63.com (cr)
DDT VERS 2.2
NEXT PC
2300 0100

Indicate the file name of BOOTFS.HEX and read in with an offset of 680H to make it load at 780H on up (it contains ORG 100H too).

-ibootfs.hex (cr)
-r680 (cr)
NEXT PC
2300 0100

Indicate the file name of the BIOS HEX file, and read it in with an offset of 2980 such that it will load at 1FB0H (it contains an ORG OF600H).

-ibios1.hex (cr)
-r2980 (cr)
NEXT PC
27EE 0000

Exit from DDT by going to location 0000H and executing a warm boot.

-g0 (cr)

Save the complete CP/M image on disk. Saving 40 256-byte pages from location 100H to 2900H.

C:\save 40 putcpmfs.com (cr)
Figure 7-10. Console dialog for system build (continued)
This chapter describes ways in which you can enhance your BIOS to make CP/M easier to use, faster, and more versatile.

Get a standard BIOS working on your computer system, and then install the additional features. Although you can write an enhanced BIOS from the outset, it will take considerably longer to get it functioning correctly.

A complete listing of an enhanced BIOS is included at the end of this chapter. It is quite large: approximately 4500 lines of source code, with extensive comments and long variable names to make it more understandable.

The sections that follow describe the main concepts embodied in the enhanced BIOS listing.
BIOS Enhancements

BIOS enhancements fall into two classes: those that add new capabilities and those that extend existing features.

Some enhancements are normally accompanied by utility programs that allow you to select the enhancement option from the console. For example, when the BIOS is enhanced to include a real time clock, you need a utility program to set the clock to the correct time. Other enhancements will not require supporting utilities. For example, if the disk drivers are improved to read and write data faster, the enhancement is "transparent." As a user, you are aware of the results of the enhancement but not of the enhancement itself.

Viewed at its simplest, the BIOS deals with two broad classes of input/output:

Character input/output
This includes the console, auxiliary, and list devices.

Disk input/output
This can accommodate several types of floppy and hard disks.

Enhancements in these areas do not fundamentally change the way that the BDOS and CCP interact with these devices. Instead, enhancements improve the way in which the device drivers deal with the devices. They can improve the speed of manipulating data, the way of handling external devices, or the user's control over the behavior of the system.

The example enhanced BIOS has capabilities not found in standard CP/M systems. These can be grouped in several main categories:

Character input/output
This area probably benefits most from enhancement. This is partly because such a wide range of peripheral devices needs to be supported and partly because this is the most visible area of interaction between you and your computer. Any improvements here will therefore be immediate and obvious to you as a user.

Error handling
CP/M's error handling is, at best, startling in its simplicity. Enhanced error handling gives you more information about the nature of the failure, and then gives you the options of retrying the operation, ignoring the error, or aborting the program. This topic is covered in detail in Chapter 9.

System date and time
This is the ability to maintain a time-of-day clock and the current date. It allows your programs to set and access the date and time. In addition, your system can react to the passing of time, and you can move certain operations into the time domain. For example, you can set upper limits on the
number of seconds, or milliseconds, that each operation should take, and arrange for emergency action if the operation takes too long.

**Logical-to-physical device assignment**

CP/M’s logical-to-physical device assignment is primitive. With enhancements, you can use any character input/output device as the system console, and output data to several devices at the same time.

**Disk input/output**

CP/M only knows about the 128-byte sector. Even with the deblocking routines shown in Figure 6-4, overall disk performance can be slow. Performance can be improved dramatically by “track buffering” (in which entire tracks are read and written at one time) or by using a memory disk (that is, using large areas of RAM as though they were a disk). These have a cost, though, in increased memory requirements.

**Public files**

CP/M’s user number system needs improvements to function well in conjunction with large hard disks.

---

**Preserving User-Settable Options**

A by-product of adding features to the BIOS is that many of these features have options that you can alter, either from the console using a utility program or from within one of your programs.

Each of these options, once set according to your preferences, or to the requirements of your hardware, do not normally change from day to day. Therefore, the BIOS should be designed so that options set by the user can be “frozen” or preserved on the disk by using a utility program, FREEZE. All of the variables recording these options are gathered into a single area and then this area is written out to the disk.

This area is called the *configuration block*. In practice, there are two configuration blocks: one short term and the other long term. The short term block is not preservable — you can set options within it, but they cannot be preserved after you switch your computer off. The system date, for example, is normally set each time you turn your computer on, and therefore is kept in the short term block. The baud rate for your printer, on the other hand, is kept in the long term block so that it can be saved permanently.

An extra BIOS entry point, CBS$Get$Address, has been built into the enhanced BIOS so that utility programs can locate variables in both configuration blocks. For example, when a utility needs to know where the date is kept in memory, it calls CBS$Get$Address using a code number (specific for date) in a register. CBS$Get$Address returns the address of the date in memory. If a new version of the BIOS is produced with the date in a different location, CBS$Get$Address will still hand the correct, although different, address back to the utility program.
Two other variables that CB$Get$Address can access pertain to the configuration block itself. One is the relative address of the start of the long term configuration block. The other is the length of the long term block. These are used by the FREEZE utility when it needs to preserve the long term block on a disk. FREEZE must (1) read in the sectors containing the long term block from the CP/M BIOS image on the reserved area of the disk, (2) copy the current RAM-resident version of the long term block over the disk image version, and then (3) write the sectors back onto the disk.

Figure 8-1 shows how the long term block appears on disk and in memory. The
size of the CCP and BDOS do not change, even if the BIOS does. Therefore, the sector containing the start of the BIOS will not change. The formula (using decimal numbers)

\[
\text{BIOS Start Sector} + \text{INT}(\text{Relative LTB Address} / 128)
\]

then gives the start sector number to be read in. The number of sectors to read is calculated as follows:

\[
(\text{Long Term Block Length} + 127) / 128
\]

The relative address and length can be used to locate the long term block in the BIOS executing in RAM.

---

**Character Input/Output**

The character I/O drivers shown in the example BIOS, Figure 8-10, have been enhanced to have the following features:

- A single set of driver subroutines controlling all character devices
- Preservation of option settings
- Flexible redirection of input/output between logical and physical devices
- Interrupt-driven input drivers, to get user “type-ahead” capability
- Support of several different protocols to avoid loss of data during high-speed output to printers or other operations
- Forced input of characters into the console input stream, allowing automatic commands at system start-up
- Conversion of terminal function keys into useful character strings
- Ability to recognize “escape sequences” output to the console and to take special action as a result
- Ability to read the current time and date as though they were typed on the console
- “Timeout” signaling when the printer is busy for too long.

Each of these features is discussed in the following sections, as an introduction to the actual code example.

**Single Set of Driver Subroutines**

In the following examples, only a single set of subroutines is used to process the input and output for all of the physical devices in the system.

This is made possible by grouping all of the individual device’s characteristics
into a table called the **device table**. For example, in order to get a character from the current console device, the address of its device table will be handed over to the subroutines. These in turn will use the appropriate values from the device table when they need to access a port number or any unique attribute of that device.

In our example, the drivers assume that all of the physical devices use serial input/output. To support a device with parallel input/output, you would need to extend the device table to include a field that would enable the drivers to detect whether they were operating on a serial or parallel device. You would probably also have to add different device initialization and input/output routines more suited to the problems of dealing with a parallel port.

The device table structure consists of a series of equate (EQU) instructions. These define the relative offset of each field in the table. Each definition is expressed by referencing the *preceding* field so that you can insert additional fields without revising the definitions for all the other fields.

Individual instances of device tables are then defined as a series of define byte (DB) and define word (DW) lines. The drivers are given the base address of the device table whenever they need to do something with a device. By adding the base address to the relative address (defined by the equate), the drivers can determine the actual address in memory that contains the required value. The detailed contents of the device table are described later in this chapter.

### Permanent Setting of Options

About the only options that need preserving in the long term configuration block are the values used to initialize the hardware chips. Other options can be set during automatic execution of the command file when CP/M is first loaded.

### Redirection of Input/Output Between Devices

As you recall, the BDOS only "knows about" the **logical** devices console, reader, punch, and list. Using the IOBYTE at location 0003H in conjunction with the STAT utility, you can redirect the BDOS to assign the logical devices to specific physical devices. However, the redirection provided by CP/M is rather primitive. It permits only four physical devices per logical device. Input and output of a logical device must always come from the same physical device. Output data can only be sent to a single destination, or (using the CONTROL-P toggle) to the console and the list device.

The system in Figure 8-10 supports up to 16 physical devices. Any one of these devices can act as the console, reader, punch, or list device. Input can come from any single device. Output can be sent to any or all of the devices. Each logical device's input and output are separate— that is, console input can come from physical device X while the output can be sent to physical devices Y and Z.

Device redirection can be done dynamically, either from within a program or by using a system utility program. For example, if you have some special input
device, your program can momentarily switch over to reading input from this device as though it were the console, and then revert back to reading data from the “real” console.

This redirection scheme is achieved by defining a 16-bit word, called the redirection word, in the long term configuration block for each of the following logical devices:

- Console input
- Console output
- Auxiliary (reader/punch) input
- Auxiliary (reader/punch) output
- List input (printers need to send data, too)
- List output.

Each bit in a given redirection word is assigned to a physical device. For input, the drivers use the device corresponding to the first 1 bit that they find in the redirection word. For output, the drivers send the character to be output to all of the devices for which the corresponding bit is set.

The example code does not select a different driver for each bit set — it selects a specific device table and then hands over the base address of this table to the common driver used for all character operations.

**Interrupt-Driven Input Drivers**

With a standard CP/M BIOS, character data is read from the hardware chips only when control is transferred to the CONIN or READER subroutines. If this character data arrives faster than the BIOS can handle, data overrun occurs and incoming characters are lost.

By using interrupts, the hardware can transfer control to the appropriate interrupt service routine whenever an incoming character arrives. This routine reads the data character and places it into a buffer area to wait for the next CONIN or READER call, which will get the character from the buffer and feed it into the incoming data stream.

User programs and the CCP are “unaware” of this process, perceiving only that data characters are available. However, users will become aware of the process; they will be able to enter data characters from the keyboard before the program is ready for them. This gives the technique its other name—“type-ahead.” Although this technique does not alter the speed of execution of any programs running under CP/M, it does create the illusion of greater speed, since pauses while a program accepts data vanish completely. The user can enter data at a rate convenient to the tasks or thoughts at hand, without regard to the rate at which the program can accept that data.
The example contains the code necessary to handle arriving characters under interrupt control. In order to be of general applicability, the code assumes a "flat" interrupt structure: that is, all character input interrupts cause control to be transferred to the same address in memory. The address is determined by the actual hardware interrupt architecture.

The simplest interrupt schemes use the restart (RST) instructions built into the 8080 CPU chip. In the RST scheme, the external hardware interrupts what the CPU chip is doing and forces one of the eight RST instructions into the processor. Each RST instruction causes the processor to execute what is, in effect, a CALL instruction to a predetermined address in memory.

In more complicated systems, a specific interrupt controller chip (such as the Intel 8259A) will be used. In addition to providing very sophisticated (and complicated) prioritization of interrupts, the interrupt controller can transfer control to a different address depending on which physical device causes the interrupt. It does this by forcing the CPU to execute a CALL instruction to a different address for each device.

In both architectures, it is the responsibility of the BIOS writer to initialize all the hardware chips so that an interrupt occurs under the correct circumstances. The BIOS writer also must plant instructions at the correct places in memory to receive control from an RST instruction or from the fake CALL instruction emitted by the interrupt controller.

Some hardware requires that the interrupt service subroutine inform it as soon as the interrupt has been serviced and the character has been input. The example drivers provide for this.

This section deals with using interrupts for the input drivers, not the output drivers. All of today's microcomputers can output data much faster than external peripherals can handle. After the first few minutes of output, the computer will fill any reasonably sized buffer — and from this point there is no advantage in having a buffered output system. The computer still must slow down to the peripheral's data rate for each character, although now it is waiting to put the character in the output buffer rather than out to the peripheral.

One exception to this is where you have a large amount of "spare" memory and a "slow" printer (which most of them are). Increasing numbers of systems have more than 64K of RAM. The 8080 or Z80 can't address more than this, but a "bank switched" memory system can switch blocks of memory in and out of that 64K address space.

Using this trick, you can access memory "unknown" to CP/M, store some characters in it, switch back to the normal 64K memory, and return control to the caller of the BIOS output routine. When the physical device is ready to accept another output data character from the CPU, it will generate an interrupt. The interrupt service routine then will access the "secret" buffer, output the characters to the device, and switch back to the normal memory.

For example, if you have a printer that prints at 80 characters per second and
From the point of view of software, interrupt-driven input drivers are divided into two major groups: the interrupt service routine that reads the characters and stacks them in a buffer, and the non-interrupt routines that get the characters from the buffer and handle the other BIOS functions such as returning console status.

The input character buffer serves as a transfer mechanism between the two groups of subroutines, although the device table also plays an important role.

The example code uses a circular buffer, as shown in Figure 8-2.

The drivers start putting data into the beginning of the buffer. When the last character in the buffer has been reached, the drivers reset to the beginning of the buffer and start over. This, of course, assumes that the non-interrupt drivers have been getting data from the front of the buffer, thus creating space for additional incoming data.

Each device table contains the address of the input buffer, a "put" pointer (for the interrupt service routine), and a "get" pointer (for the non-interrupt service routine). It also contains two character counts: the total number of characters and the number of control characters in the input buffer. You can see how the put and
get pointers operate asynchronously. The put pointer is used every time an incoming character generates an interrupt. The get pointer is used for each CONIN call.

The get and put pointers are only single-byte values and are more accurately described as “relative offsets.” That is, they contain a value which, when converted to a word and added to the base address of the buffer, will point directly to the appropriate position inside the buffer.

By making the buffer a binary number of characters long — 32 characters, for example — a programming trick can be used to make the buffer appear circular. The device tables contain a mask value formed from the buffer's length minus one (length − 1). Whenever the get or put pointers are incremented by one (to “point” to the next character position), the updated value is ANDed with this (length − 1) mask. In this example, if the get value goes from 31 (the relative address of the last character in the buffer) to 32 (which would be “off the end”), the masking operation will reset it to zero (the relative address of the first character of the buffer). This avoids having to compare pointers to know when to reset them.

It is also simpler to use a count of the number of characters in the buffer, rather than comparing the get and put pointers, to distinguish between an empty and a full buffer. To support different serial protocols, the driver must be able to react when the buffer is within five characters of being full and when it drops below half empty. Both of these conditions are much easier to detect using a simple count that is incremented as a character is put into the buffer and decremented as a character is retrieved from the buffer.

The count of control characters is used to deal with a class of programs that incessantly “gobble” characters, thereby rendering any type-ahead useless. An example is Microsoft’s BASIC interpreter. When it is interpreting a program, you can enter a CONTROL-C from the keyboard and the interpreter will come to an orderly stop. It does this by constantly making calls to CONST (console status). If it ever detects an incoming character, it makes a call to CONIN to input the character. A character that is not CONTROL-C is discarded without further ado. Thus, any characters that are input are consumed, destroying the effect of type-ahead.

To deal with this problem, the CONST routine shown in the example can be told to “lie” about the console’s status. In this mode, CONST will only indicate that characters are waiting in the input buffer if a control character is received. It uses the control character count to determine whether there are control characters in the buffer; this count is incremented by the interrupt service routine when it detects one, and decremented by the CONIN routine when it gets a control character from the buffer.

**Protocol Support**

In this context, a protocol is a scheme to avoid loss of data that would otherwise occur if a device sent data faster than the receiving device could handle
it. For example, protocols are used to prevent the CPU sending data out to a printer faster than the printer can print the characters and move the paper. The drivers also support input protocols, indicating to a transmitting device when the input buffer gets close to being full.

Two basic methods are used to implement protocols. The first uses the control lines found in the normal RS-232C serial interface cables. For data being output by the computer, the data terminal ready (DTR) signal is used, and for incoming data, the request to send (RTS) signal. These signals conform to the electrical standards for the RS-232C interface; they are considered true when they are at some positive voltage between +3 and +12 volts, and false when they are between −3 and −12 volts.

The second method uses ASCII control characters instead of control signals. Two separate protocols are supported by this method. One uses the ASCII characters XON and XOFF. Before the sending device (the computer or some peripheral device) sends a data character, it checks to see if an XOFF character has been received. If so, the sender will wait for an XON character. The receiving device will only send an XON when it is ready to receive more data.

The second protocol uses the characters ETX (end of transmission) and ACK (acknowledge). This method is normally used only when transmitting data from the computer to a buffered printer. A message length (usually half the printer’s buffer size) is defined. When this number of characters has been output, the computer will send an ETX character. No further output will occur until the computer receives an ACK character from the printer.

The example drivers support the DTR high-to-send, the XON/XOFF, and the ETX/ACK protocols for output data. For input, they support RTS high-to-receive and XON/XOFF.

The input protocols are invoked when the input buffer gets within five characters of being full. Then the drivers output an XOFF character or lower the RTS signal voltage, or do both. Only when the input buffer has been emptied to 50% capacity will the drivers send XON or raise the RTS line, or both.

As an emergency measure, if the input buffer becomes completely full, notwithstanding protocols, the drivers will output a predetermined character (defined in the device table) each time they discard an incoming character. This is normally the ASCII BEL (bell) character. When you type too far ahead, the terminal will start beeping to tell you that data is being dropped.

**Forced Input Into the Console Stream**

All application languages provide a means of reading data from the console keyboard. This makes the console input stream a useful gateway to the system. A simple enhancement to the CONIN/CONST routines makes it easy to “fool” the system into acting as if data had been input from the keyboard when in fact the data is coming in from a character string in memory.
In the enhanced BIOS, both CONIN and CONST are extended to check a pointer in the long term configuration block, as shown in Figure 8-3. If this pointer is pointing at a nonzero byte, then that byte is returned as though it had come from the console keyboard. The forced input pointer is then moved up one byte in memory. The process of forcing input continues until a zero byte is encountered.

Forced input serves several purposes. It can be used to force a command or commands into the system when the system first starts up. In conjunction with a utility program, it can allow the user to enter several CP/M commands on a single command line, injecting the characters as each of the commands is executed. It also makes possible the features described in the next two sections.

**Support of Terminal Function Keys**

Many terminals on the market today have special function keys on their keyboards. When you press one of these keys, the terminal will emit several characters, the first of which is normally the ASCII ESC (escape) character. The remaining one or two characters identify the specific function key that was pressed.

For these function keys to be of any practical use, an applications program must detect the incoming escape sequence and take appropriate action. The problem is that not all terminal manufacturers support the ANSI standard escape sequences.
The example drivers avoid this problem by providing a general-purpose method, shown in Figure 8-4, of detecting escape sequences and of substituting a user-defined character string that is injected into the console input stream as though it had been entered from the keyboard.

This scheme permits function keys to be used very flexibly, even for off-the-shelf programs that have not been designed specifically to accept function key input.

There is, however, one stumbling block. When an ESCAPE character is received, the program must detect whether this is the start of a function key sequence or the user pressing the ESCAPE key on the terminal's keyboard. In the former case, the

Figure 8-4. CONIN decodes terminal function keys
driver must wait to determine whether a function key string must be substituted for the escape sequence. In the latter case, the driver must input the ESCAPE character as it would other incoming data characters.

This recognition can only be done by moving into the time domain. When the CONIN routine (the non-interrupt routine) gets an ESCAPE character from the input buffer, it delays for approximately 90 milliseconds, enough time for a terminal-generated character sequence to arrive. CONIN then checks the input buffer to see if it contains at least two characters. If it does, the driver checks for a match in a function key table in the long term configuration block. If the characters match a defined function key, then the string associated with the function key will be injected into the console stream by pointing the forced input pointer at it. If the characters do not match anything in the function key table, then the ESCAPE and subsequent characters are handed over as normal data characters.

If after the 90-millisecond delay no further characters have arrived, the ESCAPE character is handed over as a normal character, on the basis that it must have been a manually entered ESCAPE character rather than part of a terminal-generated sequence.

The example drivers show the necessary code and tables for function keys that emit three characters. You could modify them easily for two-character sequences, or, if you are fortunate enough to have a keyboard that uses all eight bits of a byte, to recognize single incoming characters.

Processing Output Escape Sequences

The output side of the console driver, the CONOUT routine, can also be enhanced to recognize escape sequences. It uses a vectored JMP instruction to keep track of the current state of affairs. The CONOUT driver gets an address from the vector and transfers control to it. Normally this vector is set to direct control to the output byte routine. However, if an ESCAPE character is detected in the output stream, the vector is changed to transfer control to a routine that will recognize the character following the ESCAPE. If recognition does not occur, the driver will output an ESCAPE followed by the character that arrived after it.

If the second character is recognized, then the driver can transfer control to the correct escape-sequence processor. This processor can then take whatever action is appropriate. It must also make sure that when all processing is finished, the console output vector is set to process normal output characters again.

This technique is described in more practical detail in the next section, where it is used to preset and read the date and time. You can easily extend the recognition tables in the long term configuration block to perform any special processing that you need, ranging from altering the I/O redirection words to changing any other variable in the system or programming special hardware in your computer.

Be careful not to embed any pure binary values in the sequence of characters going out to the CONOUT routine. If you attempt to send a value of 09H (the TAB
Chapter 8: Writing an Enhanced BIOS

character) out via the BDOS, it will gratuitously expand the tab out to some number of blanks. If you need to send out a bit pattern, such as the I/O redirection word, split it up into a series of 7-bit long values. Then send it out with each byte having the most significant bit set to 1. A value of 09H will then become 89H, preventing the BDOS from expanding it to blanks.

Reading Date and Time From Console

For the moment, set aside the question of how the date and time get into the system. Since the date and time are stored in the short term configuration block (there being no need to save them from one work session to the next), all that the BIOS needs to be able to do is recognize a request from an applications program to read either the date or the time and then set the forced input pointer to the appropriate string in memory. Both the date and time strings are terminated by a LINE FEED followed by a 00 byte.

This sequence of events is shown in Figure 8-5.

You can see that the characters “ESC d” output to CONOUT cause it to point the forced input pointer at the date in memory. Subsequent calls to CONIN bring the characters in the date into the program as though they were being entered on the keyboard.

![Figure 8-5](image_url)
“Watchdog” Timeout on Printer

There is no provision in CP/M to deal with a hardware device that for one reason or another is permanently unavailable. Unless special steps are taken in the drivers, the system will screech to a halt in a loop, reading status and testing for the peripheral to be ready.

The example enhancement code shows a scheme, using a real time clock, that can detect when a device such as a printer fails to come ready for more than 30 seconds. On detecting this situation, the code outputs a message to all of the console devices that are not also being used as printers. This type of output is needed to avoid “deadly embraces” where a printer not being ready generates a message that cannot be output because the printer is not ready.

The code that performs the timing function is known as a watchdog timer. Each time the real time clock “ticks,” the interrupt service routine checks the watchdog count. If the count is nonzero, it is decremented. If the watchdog timer reaches zero, exceeding the time allowed, the drivers will display a message on the console indicating that the printer has been busy for too long. The user then has the option of making the printer ready and trying again to output data, ignoring the error and carrying on, or aborting the program by doing a BDOS System Reset (function 0).

Although sending an error message to the console sounds simple, it is complicated if console output is directed to the offending printer itself. The drivers attempt to solve this problem by sending the message only to those devices being used as consoles and not as printers. If all consoles are being used as printer devices as well, the driver will send the message to device 0 — normally the main console.

Keeping Time and Date

CP/M does not have provision for keeping the current time and date in the system. The example enhancement shows how to keep the time of day and the current date in the short term configuration block by using escape sequences output to the console (1) to set them to the correct values and (2) to “read” them from the console input stream.

The example presupposes that the system has a hardware chip that can be programmed to generate an interrupt every 1/60th of a second (16.666 milliseconds). This provides a divide-down counter to measure seconds elapsed. Of course, if your computer has a true real time clock that you can read and get the current time in hours, minutes, and seconds, your code will be very simple. You still will need to have the clock generate a periodic interrupt, however, in order to use the watchdog feature for timing printer and disk operations.

Actual time is kept as ASCII characters, using another ASCII control table to determine when “carry and reset to zero” should occur. By changing two bytes in this table, the time can be kept in 12- or 24-hour format.
The date is simply stored as a string. The example code does not attempt to make sure that the date is valid, nor to update when midnight rolls around. This could be done easily by the BIOS — but it would take a fairly large amount of code.

**Watchdog Timer**

Having a periodic source of interrupts also opens the door to building in an emergency or watchdog timer. This is nothing more than a 16-bit counter. Each time the real time clock interrupts, or ticks, the interrupt service routine checks the watchdog count. If it is already at zero, nothing more happens — the watchdog is not in use. If it is nonzero, the routine decrements the count by one. If this results in a zero value, the interrupt service routine CALLs a predetermined address. This will be the address of some emergency interrupt service routine that can then take special action, such as investigating the cause of the timeout.

The watchdog routine has a non-interrupt-level subroutine associated with it. Calling this set watchdog subroutine provides a means of setting the count to a predetermined number of real time clock “ticks” and setting the address to which control should be transferred if the count reaches zero.

Having called the set watchdog subroutine, the driver can then sit in a status loop, with interrupts enabled, waiting for some event to occur. If the event happens before the watchdog count hits zero, the driver must call the set watchdog routine again to set the count back to zero, thereby disabling the watchdog mechanism.

The watchdog timer can be used to detect printers that are busy for too long or disk drives that take too long to complete an action either because of a hardware failure or because the user has not loaded the disk into the drive.

**Data Structures**

As already stated, each character I/O device has its own device table that describes all of its unique characteristics.

The other major data structure is the configuration blocks — both short and long term.

This section describes each field in these data structures.

**Device Table**

Figure 8-6 shows the contents of a device table. More correctly, it shows a series of equates that define the offsets of each field in the device table. The drivers are given the base address of a specific device table. They then access each field by adding the required offset to this base address.

The first part of the device table is devoted to the physical aspect of the device, defining which port numbers are to be used to communicate with it. The drivers need to know several different port numbers since each one is used for a particular
The drivers use a device table for each physical device they service. The equates that follow are used to access the various fields within the device table.

0000 = DTSStatusPort EQU 0  ; Device status port number
0001 = DTSDataPort EQU DTSStatusPort+1  ; Device data port number
0002 = DTSOutputReady EQU DTSDataPort+1  ; Output ready status mask
0003 = DTSInputReady EQU DTSOutputReady+1  ; Input ready status mask
0004 = DTSDRSReady EQU DTSInputReady+1  ; DTR ready to send mask
0005 = DTSResetIntPort EQU DTSDRSReady+1  ; Port number used to reset an interrupt
0006 = DTSResetIntValue EQU DTSResetIntPort+1  ; Value output to reset interrupt
0007 = DTSDetectErrorPort EQU DTSResetIntValue+1  ; Port number for error detect
0008 = DTSDetectErrorValue EQU DTSDetectErrorPort+1  ; Mask for detecting error (parity etc.)
0009 = DTSResetErrorPort EQU DTSDetectErrorValue+1  ; Port number to reset error
000A = DTSResetErrorValue EQU DTSResetErrorPort+1  ; Value output to reset error
000B = DTRTSControlPort EQU DTSResetErrorValue+1  ; Control port for lowering RTS
000C = DTRTSControlValue EQU DTRTSControlPort+1  ; Value, when output, to drop RTS
000D = DTSRaiseRTSValue EQU DTRTSControlValue+1  ; Value, when output, to raise RTS
000E = DTSStatus EQU DTSRaiseRTSValue+1  ; Device logical status (incl. protocols)
000F = DTSStatus2 EQU DTSStatus+1  ; Secondary status byte
0010 = DTSFakeTypeahead EQU DTSStatus2+1  ; Requests InputStatus to return "Data Ready" when control characters are in input buffer
0011 = DTEtxCount EQU DTSFakeTypeahead+1  ; No. of chars. sent in Etx protocol
0012 = DTEtxMessageLength EQU DTEtxCount+2  ; Specified message length
0013 = DTBufferBase EQU DTEtxMessageLength+2  ; Address of input buffer
0014 = DTBufferBase+2 EQU DTBufferBase+2  ; Offset for putting chars. into buffer
0015 = DTGetOffset EQU DTBufferBase+2  ; Offset for getting chars. from buffer
0016 = DTBufferLengthMask EQU DTGetOffset+1  ; Length of buffer - 1
Note: Buffer length must always be a binary number; e.g. 32, 64, or 128,
This mask then becomes:
32 -> 31 (0001'1111)
64 -> 63 (0011'1111)
128-> 127 (0111'1111)

Figure 8-6. Device table equates
function. Depending upon your hardware, each port number could be different; however, with standard Intel or Zilog chips, you will often find that the same port number is used for several functions. The drivers also need to know what bit patterns to expect when they read some ports and what values to output to ports in order to obtain particular results.

The layout of the device table and the manner in which the equates are declared are designed to make it easy for you to change the contents of the table to meet your own special requirements. The fields in this first section of the device table are discussed in the sections that follow.

**DT$Status$Port** The driver reads this port to determine whether the hardware chip has incoming data ready to be input to the computer or whether the chip is capable of accepting another data character for output to the physical device.

**DT$Data$Port** The driver reads from this port to access the next data character from the physical device. The driver also writes to this port to output the next data character to the device.

If your computer hardware requires that the input data port be a different number from the output data port, you will have to alter the coding in the device table equates as well as make the necessary changes in the input and output subroutines in the body of the code.

**DT$Output$Ready** This is the bit mask that the driver will AND with the current device status (obtained by reading the DT$Status$Port) to see whether the device is ready to accept another output character. It assumes that the device is ready if the result of the AND instruction is nonzero. You may have to change some JNZ (jump...
nonzero) instructions to \texttt{JZ} (jump zero) instructions if your hardware device uses inverted logic, with bits in the status byte set to 0 to indicate that the device can accept another character for output.

Note that this status check relates only to the output chip—it is completely separate from the question of whether the peripheral itself is ready to accept data.

\textbf{DT$\text{Input\$Ready}$} This is the bit mask that the driver will AND with the current device status to see if there is an incoming data character. The drivers again presume that if the result of the AND is nonzero, then an incoming data character is waiting to be read from the data port. You will need to make changes similar to those for the output subroutines described in the previous section if your hardware uses inverted logic (0 bit means incoming data).

\textbf{DT$\text{DTR\$Ready}$} DTR stands for \textit{data terminal ready}. It refers to one of the control lines connected from the actual peripheral device to the I/O chip (via several other integrated circuits). The drivers, as an option, will only output data to the device when the DTR signal is at a positive voltage. If the peripheral, in order to stop the flow of data characters being output to it, lowers the DTR signal to a negative voltage, the drivers will wait. Once DTR goes positive again, the drivers will resume sending data. Many hard-copy devices use this scheme to give themselves a chance to print out data received from the computer. They may have to lower DTR for several seconds, while they perform paper movement, for example.

The value in this field is a bit mask that the drivers use on the device status to determine the state of the data-terminal-ready control signal.

\textbf{DT$\text{Reset\$Int\$Port}$} Since the input side of the drivers uses interrupts, when an incoming character is ready to be input by the CPU, the hardware generates an interrupt signal, and control is transferred to the interrupt service routine. This routine "services" the interrupt by reading the incoming data character, saving it in memory, and then transferring control back to whatever was being executed when the interrupt occurred.

The more complicated interrupt controller chips (such as the Intel 8259A) must be told as soon as a given interrupt has been serviced so that they can permit servicing of any lower priority interrupts that may be waiting.

This field contains the port number that will be used to "reset" the interrupt, or more correctly, to indicate the end of the previous interrupt’s servicing.

\textbf{DT$\text{Reset\$Int\$Value}$} This is the value that will be output to the \texttt{DT$\text{Reset\$Int\$Port}$} to tell the hardware that the previous interrupt service has been completed.

\textbf{DT$\text{Detect\$Error\$Port}$} Before the driver attempts to read any incoming data from the \texttt{DT$\text{Data\$Port}$}, it checks to see if any hardware errors have occurred. It does so by reading status from this port.
DT$Detect$Error$Value  The status byte that is input from the DT$Detect$Error$Port is ANDed with this value. If the result is nonzero, the driver assumes that an error has occurred.

DT$Reset$Error$Port  If an error has occurred, the driver outputs an error reset value to this port number.

DT$Reset$Error$Value  This is the value that will be output to the DT$Reset$Error$Port to reset an error.

DT$RTS$Control$Port  The drivers use this port number to control the request-to-send line if the RTS protocol option is selected.

DT$Drop$RTS$Value  This value is output to the RTS control port to lower the RTS line so that some external device will stop sending data to the computer.

DT$Raise$RTS$Value  This value is output to raise the RTS line so that the external device will resume sending data to the computer.

DT$Status  This is the first of two status bytes. It contains bit flags that are set to a 1 bit to indicate the following conditions:

DT$Output$Suspend  Because of protocol, the device is currently suspended from receiving any further output characters.

DT$Input$Suspend  Because of protocol, the device has been requested not to send any more input characters.

DT$Output$DTR  The driver will maintain DTR-high-to-send protocol for output data.

DT$Output$Xon  The driver will maintain XON/XOFF protocol for output data.

DT$Output$EtX  The driver will maintain ETX/ACK protocol for output data.

DT$Input$RTS  The driver will maintain RTS-high-to-receive protocol for input data.

DT$Input$Xon  The driver will maintain XON/XOFF protocol for input data.

DT$Status$2  This is another status byte, also with the following bit flag:

DT$Fake$Typeahead  CONST will “lie” about the availability of incoming console characters. It
will only indicate that data is waiting if there are control characters other than CARRIAGE RETURN, LINE FEED, or TAB in the input buffer.

**DT$Etx$Count**  This value is only used for ETX/ACK protocol. It is a count of the number of characters sent in the current message. When this count reaches the defined message length, then the driver will send an ETX character and suspend any further output.

**DT$Etx$Message$Length**  This value is the defined message length for the ETX/ACK protocol. It is used to reset the DT$Etx$Count.

**DT$Buffer$Base**  This is the address of the first byte of the device’s input buffer.

**DT$Put$Offset**  This byte contains the relative offset indicating where the next incoming character is to be “put” in the input buffer. This byte must then be converted into a word value and added to the DT$Buffer$Base address to get the absolute memory location.

**DT$Get$Offset**  This byte contains the relative offset indicating where the next character is to be “got” in the input buffer.

**DT$Buffer$Length$Mask**  This byte contains the length of the buffer minus one. The length of the buffer must always be a binary number (8, 16, 32, 64...). Therefore, one less than the length forms a mask value. Both the get and put offsets, after being incremented, are masked with this value. When the offset reaches the end of the buffer, this masking operation will “automatically” reset the offset to zero.

**DT$Character$Count**  This is a count of the total number of characters in the buffer. It is incremented by the interrupt service routine each time a character is placed in the buffer, and decremented by the CONIN routine each time it gets a character from the buffer.

CONST uses this value to determine whether any characters are available for input.

**DT$Stop$Input$Count**  When the interrupt service routines detect that the DT$Character$Count is equal to this value (normally buffer length minus five), the drivers will invoke the selected input protocol, lowering RTS or sending XOFF, to shut off the incoming data stream.

**DT$Resume$Input$Count**  When the CONIN routine detects that the DT$Character$Count has become equal to this value, the drivers will again invoke the selected input protocol, either raising RTS or sending XON to resume receiving input data.

**DT$Control$Count**  This is a count of the number of control characters in the input buffer. CARRIAGE RETURN, LINE FEED, and TAB characters are not included in this count.
It is incremented by the interrupt service routine and decremented by CONIN. CONST uses the count when the DT$Fake$Typeahead mode is active; it will only indicate that characters are waiting in the input buffer if the control count is nonzero.

**DT$Function$Delay**  This is the number of clock ticks that should be allowed to elapse after the first character of an incoming escape sequence has been detected. It allows time for the remaining characters in the escape sequence to arrive, assuming that these are being emitted by a terminal at maximum baud rate. Normally, this will correspond to a delay of approximately 90 milliseconds.

**DT$Initialize$Stream**  This is the address of the first byte of a string. This string has the following format:

```
DB ppH  Port number
DB nnH  Number of bytes to be output
DB vvH,vvH... Initialization bytes to be output to the specified port number
```

This sequence can be repeated as many times as is necessary, with a "port" number of 00H acting as a terminator.

### Disk Input/Output

The example drivers show three main disk I/O enhancements:

- Full track buffering
- Using memory as an ultra-fast disk
- Improved error handling.

### Full Track Buffering

The 5 1/4" diskettes used in the example system are double-sided. Each side has a separate read/write head in the disk drive. The disk controller is fast enough that, if so commanded, it can read in a complete track's worth of data from one side of the diskette in a single revolution of the diskette.

The drivers have been modified to do just this. The main disk buffer has been dramatically enlarged to accommodate nine 512-byte sectors.

In the earlier standard BIOS, CP/M was configured for tracks of 18 512-byte sectors. The data from each head on a given track was laid "end-to-end" to create the illusion of a single surface with twice as much data on it. For track buffering, performance would be reduced if each read required two revolutions of the diskette, and so in this BIOS the tables and the low-level driver logic have been changed. Each surface is separated, with even numbered tracks on head 0, odd on head 1.
The track number given to the low-level drivers serves two purposes. The least significant bit identifies the head number. When the track number is shifted one bit right, the result is the physical track number to which the head assembly must be positioned.

The deblocking algorithm has also been modified by deleting references to sectors. The code is now concerned only with whether the correct disk and track are in the buffer. If this is true, the correct sector must, by definition, be in the buffer.

The deblocking code no longer takes any note when the BDOS indicates that it is writing to an unallocated allocation block — knowledge it used to bypass a sector preread in the standard BIOS. The track size in this enhanced BIOS is much larger than an allocation block, and so the question is meaningless; the whole track must be preread to write just a single sector.

This enhancement really excels when the BDOS is doing directory operations, which always involve a series of sequential reads. The entire directory can be brought into memory, updated, and written back in just two disk revolutions.

One point to watch out for is what is known as “deferred writes.” Imagine a program instructed to write on a sector on track 20. The drivers will read in track 20, copy the contents of the designated sector into the track buffer, and return to the program without actually writing the data to the disk. The program could “write” to all of the sectors on this track without any actual disk writes. During all this time, this data would exist only in memory and not on the disk drive, so if a power failure occurred, several thousand bytes of data would be lost. Writing to the directory is an exception. The drivers always physically write to the disk when the BDOS indicates that it is writing to a directory sector.

In reality, the increased risk is small. Most programs are constantly reading and writing files, so that the track buffer will be written out frequently in order to read in another track. When programs end, they close output files. This in turn triggers directory writes that force data tracks onto the disk.

If high security is a requirement for your computer, you could extend the watchdog routine to include another separate timer. You could preset this timer for, say, a ten-second delay each time you write into the track buffer but do not write the buffer to the disk. When the count expires, it would set a flag that could be tested by all of the BIOS entry points. If set, they would initiate a write of the track buffer to the disk.

Using Memory as an Ultra-Fast Disk

As you can see from the preceding section, increased performance tends to go hand in hand with increased memory requirements. This is certainly true with a “memory disk,” commonly called a RAM-disk or M-disk. In fact, to have an M-disk with reasonable storage capacity, your computer must have at least 128K bytes of additional memory.
Since the 8080 or Z80 can only address 64K of memory at one time, to get access to any of this additional memory, some part of your computer's "normal" memory must be removed from the 64K address space and the additional memory must be switched in. This is known as bank-switched memory.

Figure 8-7 shows the memory organization that is supported by the example M-disk drivers.

You can see that the system has a total of 256K bytes of RAM, organized with the top 16K, from 64K down to 48K, being "common"—that is, switched into the address space all the time. The lower 48K can be selected from five banks, numbered 0 to 4. Bank 0 is switched in for normal CP/M operations.

The M-disk parameter blocks describe a disk with eight "tracks," numbered 0 to 7. The least significant bit of the track number determines whether the base address of the track will be 0000H or 6000H. Shifting the track number right one bit gives the bank number. Each track consists of 192 sectors. To get the relative address of a sector within its "track," shift the sector number eight bits left, thus multiplying it by 128.

The M-disk is referenced by logical disk M:. A few special-case instructions are required to return the special M-disk parameter header in SELDSK.

One problem, fortunately easily solved, is that the user's DMA address coexists in the address space with the M-disk image itself. There is no direct way to move data between bank 0 and any other bank. The M-disk uses an intermediary buffer in common memory (above 48K), moving data into this, switching banks, and then moving the data down again. Figure 8-8 shows an example of this sequence, as used when reading from the M-disk.

![Figure 8-7. Memory organization for M-disk](image-url)
During cold boot initialization, the M-disk driver checks the very first directory entry (in bank 1) to see if it matches a dummy entry for a file called "M$Disk." If this entry is present, the M-disk is assumed to contain valid information. If the entry is absent, the initialization code makes this special directory entry and fills the remainder of the directory with 0E5H, making it appear empty. The dummy entry makes it appear that the "M$Disk" file is in user 15, marked System status and Read-Only—all of which are designed to prevent its accidental erasure.

Custom Patches to CP/M

Two features shown in the enhanced BIOS, one in the CCP and one in the BDOS, require changes to CP/M itself. These features are implemented by modifying the CCP and BDOS to transfer control to the BIOS at specific points, execute a few instructions in the BIOS, and then return to CP/M. The patches could be made by modifying the MOVCPM program to install the changes permanently. The changed version of MOVCPM, however, must be used with a specific version of the BIOS. Therefore, patching CP/M "on the fly" ensures that there will be no mismatch between the BIOS and the rest of CP/M.

Both of these patches were produced with the assistance of Digital Research.
User 0 Files Made Public

The first change permits files created in user area 0 to be accessible from all other user numbers. This feature comes into its own only with hard disk systems. On a hard disk, user numbers can partition the disk, but the frequently used utilities must then be duplicated in each user area. Allowing files in user area 0 to be public means that these files will be accessible from all the other user numbers. Hence the files need not be copied into each user area.

The public files feature alters the way that the BDOS performs the Search Next function, allowing access to files declared in user area 0 even when the current user number is not 0. However, the feature is a double-edged sword—user 0 files can be accidentally erased or damaged as well as accessed. Therefore, user 0 files should be declared as System status and Read-Only to protect them. As an additional precaution, public files can be turned off by a control flag in the long term configuration block. This flag is set to an initial state that disables public files.

Modified User Prompt

This modification makes the CCP display the current user number as well as the default disk. For example,

3B>

indicates that you are currently in user number 3, with disk B: as the default. In addition, if you have enabled public files, the prompt is preceded by the letter “P” to serve as a reminder:

P3B>

An Enhanced BIOS

The remainder of this chapter consists of the assembly language source code for the enhanced BIOS described here. It is rather a daunting listing, but will be well worth your study. The copious commentary has been written to make this study easier, and emphasis has been placed on explaining why as well as what things are done.

As with the standard BIOS, each line is numbered so that you can use the functional index in Figure 8-9 to find areas of interest in the listing. Note that the line numbers are not contiguous. They jump several hundred at the start of each major section or subroutine. This facilitates minor changes in the listing without revision of the functional index. The full listing is given in Figure 8-10.
<table>
<thead>
<tr>
<th>Start Line</th>
<th>Functional Component or Routine</th>
</tr>
</thead>
<tbody>
<tr>
<td>00001</td>
<td>Introductory Comments and Equates</td>
</tr>
<tr>
<td>00200</td>
<td>BIOS Jump Table with Additional Private Entries</td>
</tr>
<tr>
<td>00400</td>
<td>Long Term Configuration Block</td>
</tr>
<tr>
<td>00800</td>
<td>Interrupt Vector</td>
</tr>
<tr>
<td>00900</td>
<td>Device Port Numbers and Other Equates</td>
</tr>
<tr>
<td>01100</td>
<td>Display$Message Subroutine</td>
</tr>
<tr>
<td>01200</td>
<td>Enter$CPM Setup</td>
</tr>
<tr>
<td>01300</td>
<td>Device Table Equates</td>
</tr>
<tr>
<td>01500</td>
<td>Device Table Declarations</td>
</tr>
<tr>
<td>01700</td>
<td>General Device Initialization</td>
</tr>
<tr>
<td>01800</td>
<td>Specific Device Initialization</td>
</tr>
<tr>
<td>02000</td>
<td>Output Byte Stream</td>
</tr>
<tr>
<td>02100</td>
<td>CONST Routine</td>
</tr>
<tr>
<td>02200</td>
<td>CONIN Routine with Function Key Processing</td>
</tr>
<tr>
<td>02500</td>
<td>Console Output</td>
</tr>
<tr>
<td>02700</td>
<td>CONOUT Routine with Escape Sequence Processing</td>
</tr>
<tr>
<td>02900</td>
<td>AUXIST—Auxiliary Input Status Routine</td>
</tr>
<tr>
<td>03000</td>
<td>AUXOST—Auxiliary Output Status Routine</td>
</tr>
<tr>
<td>03100</td>
<td>AUXIN—Auxiliary Input Routine</td>
</tr>
<tr>
<td>03200</td>
<td>AUXOUT—Auxiliary Output Routine</td>
</tr>
<tr>
<td>03300</td>
<td>LISTST—List Status Routine</td>
</tr>
<tr>
<td>03400</td>
<td>LIST—List Output Routine</td>
</tr>
<tr>
<td>03500</td>
<td>Request User Choice—Request Action After Error</td>
</tr>
<tr>
<td>03600</td>
<td>Output Error Message</td>
</tr>
<tr>
<td>03656</td>
<td>Get Composite Status from Selected Output Devices</td>
</tr>
<tr>
<td>03800</td>
<td>Multiple Output of Byte to All Output Devices</td>
</tr>
<tr>
<td>04000</td>
<td>Check Output Device Logically (Protocol) Ready</td>
</tr>
<tr>
<td>04200</td>
<td>Process ETX/ACK Protocol</td>
</tr>
<tr>
<td>04400</td>
<td>Select Device Table from I/O Redirection Bit Map</td>
</tr>
<tr>
<td>04600</td>
<td>Get Input Character from Input Buffer</td>
</tr>
<tr>
<td>04800</td>
<td>Introductory Comments for Interrupt-Driven Drivers</td>
</tr>
<tr>
<td>04900</td>
<td>Character Interrupt Service Routine</td>
</tr>
<tr>
<td>05000</td>
<td>Service Device—Puts Character into Input Buffer</td>
</tr>
<tr>
<td>05300</td>
<td>Get Address of Character in Input Buffer</td>
</tr>
<tr>
<td>05400</td>
<td>Check if Control Character (not CR, LF, TAB)</td>
</tr>
<tr>
<td>05500</td>
<td>Output Data Byte</td>
</tr>
<tr>
<td>05700</td>
<td>Input Status Routine</td>
</tr>
<tr>
<td>05900</td>
<td>Set Watchdog Timer Routine</td>
</tr>
<tr>
<td>06000</td>
<td>Real Time Clock Interrupt Service Routine</td>
</tr>
<tr>
<td>06200</td>
<td>Shift HL Right One Bit Routine</td>
</tr>
<tr>
<td>06300</td>
<td>Introductory Comments for High-Level Disk Drivers</td>
</tr>
<tr>
<td>06400</td>
<td>Disk Parameter Headers</td>
</tr>
<tr>
<td>06600</td>
<td>Disk Parameter Blocks</td>
</tr>
<tr>
<td>06800</td>
<td>SELDSK—Select Disk Routine</td>
</tr>
<tr>
<td>07000</td>
<td>SETTRK—Set Track Routine</td>
</tr>
<tr>
<td>07100</td>
<td>SETSEC—Set Sector Routine</td>
</tr>
</tbody>
</table>

Figure 8-9. Functional index for listing in Figure 8-10
Chapter 8: Writing an Enhanced BIOS 237

<table>
<thead>
<tr>
<th>Line</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>07200</td>
<td>SETDMA—Set DMA Routine</td>
</tr>
<tr>
<td>07300</td>
<td>Skew Tables for Sector Translation</td>
</tr>
<tr>
<td>07400</td>
<td>SECTRAN—Sector Translation Routine</td>
</tr>
<tr>
<td>07500</td>
<td>HOME—Home Disk to Track and Sector 0</td>
</tr>
<tr>
<td>07600</td>
<td>Equates for Physical Disk and Deblocking Variables</td>
</tr>
<tr>
<td>07800</td>
<td>READ—Sector Read Routine</td>
</tr>
<tr>
<td>07900</td>
<td>WRITE—Sector Write Routine</td>
</tr>
<tr>
<td>08000</td>
<td>Common Read/Write Code with Deblocking Algorithm</td>
</tr>
<tr>
<td>08300</td>
<td>Move$8 Routine—Moves Memory in 8-Byte Blocks</td>
</tr>
<tr>
<td>08500</td>
<td>Introductory Comments for Disk Controllers</td>
</tr>
<tr>
<td>08700</td>
<td>Nondeblocked Read and Write</td>
</tr>
<tr>
<td>08900</td>
<td>M-Disk Driver</td>
</tr>
<tr>
<td>09100</td>
<td>Select Memory Bank Routine</td>
</tr>
<tr>
<td>09200</td>
<td>Physical Read/Write to Deblocked Disks</td>
</tr>
<tr>
<td>09400</td>
<td>Disk Error Handling Routines</td>
</tr>
<tr>
<td>09700</td>
<td>Disk Control Tables for Warm Boot</td>
</tr>
<tr>
<td>09800</td>
<td>WBOOT—Warm Boot Routine</td>
</tr>
<tr>
<td>10000</td>
<td>Ghost Interrupt Service</td>
</tr>
<tr>
<td>10100</td>
<td>Patch CP/M for Public Files and Prompt Changes</td>
</tr>
<tr>
<td>10300</td>
<td>Get Configuration Block Addresses</td>
</tr>
<tr>
<td>10400</td>
<td>Addresses of Objects in Configuration Blocks</td>
</tr>
<tr>
<td>10500</td>
<td>Short Term Configuration Block</td>
</tr>
<tr>
<td>10700</td>
<td>Note on Why Uninitialized Buffers are at End of BIOS</td>
</tr>
<tr>
<td>10800</td>
<td>Cold Boot Initialization Hidden in Disk Buffer Followed by All Uninitialized Buffers</td>
</tr>
</tbody>
</table>

**FIGURE 8-9.** Functional index for listing in Figure 8-10 (continued)

```plaintext
00001: ; This is a skeletal example of an enhanced BIOS.
00010: ; It includes fragments of the standard BIOS
00011: ; shown as Figure 6-4 in outline, so as to
00012: ; avoid cluttering up the enhancements with the
00013: ; supporting substructure. Many of the original
00014: ; comment blocks have been abbreviated or deleted
00015: ; entirely.
00016: ;
00017: ; -- NOTE: The line numbers at the left are included
00018: ; to allow reference to the code from the text.
00019: ; There are deliberate discontinuities in the
00020: ; numbers to allow space for expansion.
00021: ;
00022: 3030 = 00022 VERSION EDU '00' ;Equates used in the sign-on message
00023: 3230 = 00023 MONTH EDU '02'
00024: 3632 = 00024 DAY EDU '26'
00025: 3338 = 00025 YEAR EDU '93'
00026: ;
00027: ;%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
00028: *
00029: ; This BIOS is for a computer system with the following
00030: ; hardware configuration:
00031: *
00032: ; 8080 CPU
00033: ; --- 64K bytes of RAM
00034: ; --- 3 serial I/O ports (using signetics 2651) for:
00035: ; --- console, communications and list
00036: ; --- Two 5 1/4" mini floppy, double-sided, double-
00037: ; density drives. These drives use 512-byte sectors.
00038: ; These are used as logical disks A: and B:.
00039: ; Full track buffering is supported.
```

**Figure 8-10.** Enhanced BIOS listing
The BIOS length must be determined by inspection. Comment out the ORG BIOS$Entry line below by changing the first character to a semicolon (this will make the assembler start the BIOS at location 0). Then assemble the BIOS and round up to the nearest 100H the address displayed on the console at the end of the assembly.

The BIOS length must be determined by inspection. Comment out the ORG BIOS$Entry line below by changing the first character to a semicolon (this will make the assembler start the BIOS at location 0). Then assemble the BIOS and round up to the nearest 100H the address displayed on the console at the end of the assembly.

---

Two standard diskette drives (128-byte sectors) are used as logical disks C: and D:. A memory-based disk (M-disk) is supported. Two intelligent disk controllers are used, one for each diskette type. These controllers access memory directly, both to read the details of the operations they are to perform and also to read and write data from and to the diskettes.

---

Equates for defining memory size and the base address and length of the system components.

---

Two standard diskette drives (128-byte sectors) are used as logical disks C: and D:. A memory-based disk (M-disk) is supported. Two intelligent disk controllers are used, one for each diskette type. These controllers access memory directly, both to read the details of the operations they are to perform and also to read and write data from and to the diskettes.

---

Figure 8-10. (Continued)
Long term configuration block

Additional "private" BIOS entry points

The following equates are used to indicate specific physical devices.

Each logical device has a 16-bit word associated with it. Each bit in the word is assigned to a specific physical device. For input, only one bit can be set -- input will be read from the corresponding physical device. Output can be directed to several devices, so more than one bit can be set.

The following words are tested by the logical device drivers to transfer control to.

Figure 8-10. (Continued)
The stream format is:

| DB | xx | Port number (00H terminates) |
| DB | nn | Number of bytes to output to port |
| DB | vv, vv, vv.. | Values to be output |

Example data for an 8251A chip:

| DB | 0EDH | Port number for 8251A |
| DB | 0, 0, 0 | Dummy bytes to get chip ready |
| DB | 010000010B | Reset and raise DTR |
| DB | 011011110B | 1 stop, no parity, 8 bits/char, divide down of 16 |

Example data for an 8253 chip:

| DB | 001000101B | Port number for 8253 mode |
| DB | 010000010B | Reset and raise DTR |
| DB | 011011110B | 1 stop, no parity, 8 bits/char, divide down of 16 |

Example data for an 8253 chip:

| DB | ODEH | Port number for 8253 |
| DB | 0, 0, 0 | Dummy bytes to get chip ready |
| DB | 010000010B | Reset and raise DTR |
| DB | 011011110B | 1 stop, no parity, 8 bits/char, divide down of 16 |

Figure 8-10. (Continued)
Variables for the real time clock and watchdog

Example data for an 8251A chip

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0083H</td>
<td>1200 baud (based on 16x divider)</td>
</tr>
</tbody>
</table>

Function key table

This table consists of a series of entries, each one having the following structure:

- **DB** Second character of sequence emitted by terminal's function key
- **DB** Third character of sequence -- NOTE: this field will not be present if the source code has been configured to accept only two characters
- **DB** In function key sequences.
- **DB** NOTE: Adjust the equates for.
- **DB** Function Keys Length
- **DB** Three Character Function

**Figure 8-10.** (Continued)
The logic associated with function key recognition is made easier with the following equate:

```
EQU Function$Key$Length - 2
```

Three$Character$Function will be TRUE if the function keys emit a three character sequence, FALSE if they emit a two character sequence.

Maximum length of substitute Lead character is not string in table entry.

For the terminating 00-byte.

The example values shown below are for a VT-100 terminal.

<table>
<thead>
<tr>
<th>Function$Key$Entry$Size</th>
<th>16 + 1 + Function$Key$Length - 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>00C3 4F5046756E006330</td>
<td>123456789.1234 5 d 7 Use to check length</td>
</tr>
<tr>
<td>0065 0000000000000000</td>
<td>0000000000000000</td>
</tr>
<tr>
<td>010F 8041557020006340</td>
<td>0122 5842446F77006341</td>
</tr>
<tr>
<td>0122 8041557020006341</td>
<td>0135 5843526967006342</td>
</tr>
<tr>
<td>0135 8044466F77006343</td>
<td>0148 8044466F77006344</td>
</tr>
<tr>
<td>0148 0000000000000000</td>
<td>0000000000000000</td>
</tr>
<tr>
<td>0181 0000000000000000</td>
<td>0000000000000000</td>
</tr>
<tr>
<td>0194 0000000000000000</td>
<td>0000000000000000</td>
</tr>
<tr>
<td>01A7 0000000000000000</td>
<td>0000000000000000</td>
</tr>
<tr>
<td>01AA 0000000000000000</td>
<td>0000000000000000</td>
</tr>
<tr>
<td>01BD 0000000000000000</td>
<td>0000000000000000</td>
</tr>
<tr>
<td>01D0 0000000000000000</td>
<td>0000000000000000</td>
</tr>
<tr>
<td>01DD 0000000000000000</td>
<td>0000000000000000</td>
</tr>
<tr>
<td>01EE 0000000000000000</td>
<td>0000000000000000</td>
</tr>
<tr>
<td>01F3 0000000000000000</td>
<td>0000000000000000</td>
</tr>
<tr>
<td>0206 0000000000000000</td>
<td>0000000000000000</td>
</tr>
<tr>
<td>0219 FFFF 0000000000000000</td>
<td>0219 FFFF 0000000000000000</td>
</tr>
<tr>
<td>021E 0040000000000000</td>
<td>021E 0040000000000000</td>
</tr>
<tr>
<td>021F 0040000000000000</td>
<td>021F 0040000000000000</td>
</tr>
<tr>
<td>0220 0000000000000000</td>
<td>0220 0000000000000000</td>
</tr>
<tr>
<td>0221 0000000000000000</td>
<td>0221 0000000000000000</td>
</tr>
<tr>
<td>0222 0000000000000000</td>
<td>0222 0000000000000000</td>
</tr>
</tbody>
</table>

**Figure 8-10.** (Continued)
0224 65 00671 DB 'e' ;Set current date
0225 4E04 00672 DB CONOUT$Set$Date
0226 673
0227 00 00674 DB 0 ;Terminator
00675 | 00676 Long$Term$CB$Ends
00677 ;
00800 ;#
00801 ;
00802 ; Interrupt vector
00803 ;
00804 ; Control is transferred here by the programmable interrupt
00805 ; controller -- an Intel 8259A.
00806 ;
00807 ; NOTE: The interrupt controller chip requires that the
00808 ; Interrupt vector table start on a paragraph
00809 ; boundary. This is achieved by the following ORG line
0240 00810 ORG ($ AND OFFEOH) + 20H
00811 Interrupt$Vector:
00812 ;Interrupt number
0240 C37B08 00813 JMP RTCInterrupt 10 -- clock
0243 00 00814 DB 0 ;Skip a byte
0244 C3EB06 00815 JMP Character$Interrupt 11 -- character I/O
0247 00 00816 DB 0
0248 C3DB0E 00817 JMP Ghost$Interrupt 12 -- not used
024B 00 00818 DB 0
024C C3DB0E 00819 JMP Ghost$Interrupt 13 -- not used
024F 00 00820 DB 0
0250 C3DB0E 00821 JMP Ghost$Interrupt 14 -- not used
0253 00 00822 DB 0
0254 C3DB0E 00823 JMP Ghost$Interrupt 15 -- not used
0257 00 00824 DB 0
0258 C3DB0E 00825 JMP Ghost$Interrupt 16 -- not used
00826 ;
00900 ;#
00901 ;
00902 ; Device port numbers and other equates
00903 ;
0080 = 00904 CIO$Base$Port EQU 80H ;Base port number
00905
0080 = 00906 D0$Base$Port EQU CIO$Base$Port ;Device 0
0080 = 00907 D0$Data$Port EQU D0$Base$Port
0081 = 00908 D0$Status$Port EQU D0$Base$Port + 1
0082 = 00909 D0$Mode$Port EQU D0$Base$Port + 2
0083 = 00910 D0$Command$Port EQU D0$Base$Port + 3
00911
0084 = 00912
0084 = 00913 D1$Base$Port EQU CIO$Base$Port + 4 ;Device 1
0084 = 00914 D1$Data$Port EQU D1$Base$Port
0085 = 00915 D1$Status$Port EQU D1$Base$Port + 1
0086 = 00916 D1$Mode$Port EQU D1$Base$Port + 2
0087 = 00917 D1$Command$Port EQU D1$Base$Port + 3
00918
0088 = 00919 D2$Base$Port EQU CIO$Base$Port + 8 ;Device 2
0088 = 00920 D2$Data$Port EQU D2$Base$Port
0089 = 00921 D2$Status$Port EQU D2$Base$Port + 1
008A = 00922 D2$Mode$Port EQU D2$Base$Port + 2
008B = 00923 D2$Command$Port EQU D2$Base$Port + 3
008C = 00924
004E = 00925 D$Mode$Value$1 EQU 01H00H11110B
00926 ;1 stop bit, no parity
00927 ;8 bits, Async. l6x rate
00928 ;
00929 ;
0092A ;16 bit, Async. 32x rate
0092B ;
0092C ;
0092D ;
0092E ;
0092F ;
00930 ;
00931 ;
00932 ;
00933 ;
00934 ;
00935 ;
00936 ;
00937 ;
00938 ;
00939 ;
00940 ;
0038 = 00935 D$Error EQU 00111000B
0037 = 00936 D$Error$Reset EQU 00111011B
0001 = 00937 ;Same as command value plus error reset
0002 = 00938 D$Output$Ready EQU 000080001B
0003 = 00939 D$Input$Ready EQU 000080010B
0004 = 00940 D$DTR$High EQU 100000000B ;Notes: this is actually the

Figure 8-10. (Continued)
Interrupt controller ports (Intel 8259A)

LXI H, Warm$Boot$Entry ; Get BIOS vector address
SHLD 0001H ; Put address at location 0001H

Note: these equates are placed here so that they follow the definition of the interrupt vector and thus avoid "P" (phase) errors in ASM.

IC$OCWI EQU
IC$OCW2 Port EQU
IC$OCW3 Port EQU
IC$ICWI Port EQU
IC$ICW2 Port EQU
IC$EOI EQU
IC$ICWI EQU

data set-ready pin on the chip. It is connected to the OTR pin on the cable.
Raise RTS. Tx/Rx enable
Drop RTS, Tx/Rx enable

Ensure interrupts are enabled
Handover current default disk to console command processor.
Figure 8-10. (Continued)
Figure 8-10.  (Continued)
Chapter 8: Writing an Enhanced BIOS

Figure 8-10. (Continued)
The CP/M Programmer's Handbook

248

Figure 8-10. (Continued)
Chapter 8: Writing an Enhanced BIOS

---

<table>
<thead>
<tr>
<th>0324 D3</th>
<th>0325 00</th>
<th>0326 00</th>
<th>0327 C2203</th>
<th>032A C31903</th>
</tr>
</thead>
<tbody>
<tr>
<td>OB$Port</td>
<td>DB</td>
<td>DB</td>
<td>JNZ</td>
<td>JMP</td>
</tr>
<tr>
<td>OUT</td>
<td>0</td>
<td>C</td>
<td>OB$Next$Byt</td>
<td>OB$Loop</td>
</tr>
</tbody>
</table>

:;

0000h:**CONST - Console status**;
0001h:;<== BIOS entry point (standard)
0002h:

032D 2A5800: LHLDB$Console<Input>
0330 116400: LXIB8,D8$Device$Table$Addresses
0333 C4AF06: CALL Select$Device$Table
0336 C34708: JMP Get$Input$Status

0220: 

0220: CONST - Console status
0221: ;<== BIOS entry point (standard)
0222: ;

0224: ;

0227: LHLDB$Console<Input>
0228: MOV A,M
0229: ORA A
022A C2203:
022B: JZ CONIN$No$FI
022C:

022D 2A5800: LHLDB$Console<Input>
022E 032D: MOV A,M
0230 032E: ORA A
0231: JZ CONIN$No$FI
0232: INX H
0233: XCH A
0234: SHLD CB$Forced$Input
0235: RET

0236: LHLDB$Console<Input>
0237: D8$Device$Table$Addresses
0238: CALL Select$Device$Table
0239: CALL Get$Input$Character
023A: CALL Get$Input$Character

0241: ::Function key processing
0242: CPI Function$Key$Lead
0243: RNZ Return to BIOS caller if not in character
0244: PUSH PSW

---

Figure 8-10. (Continued)
The CP/M Programmer's Handbook

Figure 8-10. (Continued)
CONIN$Not$Function:

Table 8-10. (Continued)
**Figure 8-10.** (Continued)
Chapter 8: Writing an Enhanced BIOS 253

Figure 8-10. (Continued)
This routine returns the next input character from the list redirection word; if count hits 0; return with output vectored to CONOUT$Process$String; otherwise, stack character; update pointer; stack fail-safe terminator; save updated pointer.

CONOUT$Process$String: ;Get current address for stacking chars
MOV A,C
CONOUT$String$Pointer
A,C
Check if current character is OOH
CONOUT$String$Pointer
H,CONOUT$String$Length
M
CONOUT$String$Pointer
H,
LHLD CB$Auxiliary$Input ;Get redirection word
LXI D,CB$Device$Table$Addresses
CALL Select.Device$Table
JMP Get$Input$Status ;Get status from input device and return to caller

This routine checks the character count in the appropriate input buffer.
The A register is set to indicate whether or not data is waiting.

Entry parameters: none.
Exit parameters:
A = 000H if there is no data waiting
A = OFFH if there is data waiting

Auxiliary input status
This routine checks the character count in the appropriate input buffer.
The A register is set to indicate whether or not data is waiting.

Entry parameters: none.
Exit parameters:
A = 000H if there is no data waiting
A = OFFH if there is data waiting

Auxiliary output status
This routine sets the A register to indicate whether the auxiliary device(s) is/are ready to accept output data.
As more than one device can be used for auxiliary output, this routine returns a Boolean AND of all of their statuses.

Entry parameters: none
Exit parameters:
A = 000H if one or more list devices are not ready
A = OFFH if all list devices are ready

Figure 8-10. (Continued)
appropriate logical auxiliary device.

Entry parameters: none.

Exit parameters:

A = data character

Figure 8-10. (Continued)
The CP/M Programmer's Handbook

04DD 00DA0750720413 LISTBusyMessage: DB CR,LF,7,'Printer not Ready?',CR,LF,0
03414  
03415  
03416  
03417  
04FS 2A6200 03418 LHLDCB*List*Output ;Get list redirection word
04FS 11DD04 03419 LXIDLIST*Busy*Message ;Message to be output if time
03420  
04FB C3A205 03421 JMP Multiple*Output*Byte
03422  
03500  ; Request user choice
03501  
03502  
03503  ; This routine displays an error message, requesting
03504  ; a choice of:
03505  
03506  ; R -- Retry the operation that caused the error
03507  ; I -- Ignore the error and attempt to continue
03508  ; A -- Abort the program and return to CP/M
03509  
03510  ; This routine accepts a character from the console,
03511  ; converts it to uppercase and returns to the caller
03512  ; with the response in the A register.
03513  
03514  RUC*Message:
03515  0500 2020202020202816 DB CR,LF  
03516  DB 'Enter R - Retry, I - Ignore, A - Abort : ' .O
03517  ;
03518  ;
03519  Request*User*Choice:
03520  052F C02D03 CALL CONST ;Gobble up any type-ahead
03521  0532 C30B05 CALL CONIN  
03522  0535 C3A003 CALL CONIN  
03523  0538 C32F05 JMP Request*User*Choice
03524  03525  RUC*Buffer*Empty:
03526  053B 21F04 LXI H,RUC*Message ;Display prompt
03527  053E C05305 CALL Output*Error*Message
03528  03529  
03530  CALL CONIN ;Get console character
03531  CALL A$To$Upper ;Make uppercase for comparisons
03532  STA Disk$Action$Confirm ;Save in confirmatory message
03533  PUSH PSW ;Save for later
03534  0548 21B00D LXI H,Disk$Action$Confirm
03535  054E C05305 CALL Output*Error*Message
03536  03537  POP PSW ;Recover action code
03538  0551 F1  
03539  
03540  
03541  
03542  
03543  
03544  
03545  
03546  
03547  
03548  
03549  
03550  
03551  
03552  
03553  
03554  
03555  
03556  
03557  
03558  
03559  
03560  
03561  
03562  
03563  
03564  
03565  
03566  
03567  
03568  
03569  
03570  
03571  
03572  
03573  
03574  
03575  
03576  
03577  
03578  
03579  
03580  
03581  
03582  
03583  
03584  
03585  
03586  
03587  
03588  
03589  
03590  
03591  
03592  
03593  
03594  
03595  
03596  
03597  
03598  
03599  
03600  
03601  
03602  
03603  
03604  
03605  
03606  
03607  
03608  
03609  
03610  
03611  
03612  
03613  
03614  
03615  
03616  
03617  
03618  
03619  Output*Error*Message:
03620  0553 E5 PUSH H ;Save message address
03621  0554 2A5A00 LHLDCB*Console*Output ;Get console redirection bit map
03622  0557 EB XCHG  
03623  0558 2A6200 LHLDCB*List*Output ;Get list redirection bit map
03624  ; HL = list, DE = console
03625  ; Now set to 0 all bits in the console

Figure 8-10. (Continued)
Chapter 8: Writing an Enhanced BIOS 257

Setting the A Register to Indicate Whether the Output Device(s) Is/Are Ready

This routine sets the A register to indicate whether the output device(s) is/are ready to accept output data. As more than one device can be used for output, this routine returns a Boolean AND of all of their statuses.

Let X be 1/O redirection bit map for output device(s)

Figure 8-10. (Continued)
This routine outputs a data byte to the all of the devices specified in the I/O redirection word. It is similar to CONOUT except that it uses the watchdog timer to detect if any of the devices stays busy for more than 30 seconds at a time. It outputs a message to the console if this happens.

**Entry parameters**

- HL = I/O redirection bit map
- DE -> Message to be output if time runs out
- C = data byte

**Multiple output byte**

- MOB$Maximum$Busy: EDU 1800; Number of clock ticks (each at 1.666 milliseconds) for which the device might be busy
- MOB$Character: DB 0; Character to be output
- MOB$Busy$Message: DW 0; Address of message to be output if time runs out
- MOB$Need$Message: DB 0; Flag used to detect that the watchdog timer timed out

**Addresses of dev. tables on stack ready for loop**

- MOB$Next$Device:
- MOB$Maximum$Busy
- MOB$Not$Ready

**Save device table addresses pointer**

- MOB$Device$Table

**Save redirection bit map**

- MOB$Ignor$Exit

**Multiple$Output$Byte:**

- MOB$Start$Watchdog
- MOB$Wait:

**Figure 8-10.** (Continued)
Figure 8-10. (Continued)
The CP/M Programmer's Handbook

Exi parameter's

Message count downdated (and reset if necessary)

DE -} device table

Entry parameters

Exit parameters

Message count downdated (and reset if necessary)

Process ETX/ACK protocol

This routine maintains ETX/ACK protocol.

After a specified number of data characters have been output to the device, an ETX character is output and the device put into output suspended state. Only when an incoming ACK character is received (under interrupt control) will output be resumed to the device.

Select device table

This routine scans a 16-bit word, and depending on which is the first 1-bit set, selects the corresponding device table address.

Figure 8-10. (Continued)
Chapter 8: Writing an Enhanced BIOS 261

Get input character

Select Device Table:

Entry parameters

DE -> Table of device table addresses

The first address in the list is called

if the least significant bit of the bit map is nonzero, and so on.

Exit parameters

BC -> Current entry in device table addresses

DE = Selected device table address

HL = Shifted bit map

Nonzero if a 1-bit was found

Zero if bit map now entirely 0000

Nhce: If HL is 0000H on input, then the first entry in the
device table addresses will be returned in DE.

066F 7C 04427 MOV A, H ;Get most significant byte of bit map
0671 85 04428 ORA L ;Check if HL completely 0
0672 7D 04429 MOV A, L ;Check if the LS bit is nonzero
0673 E0 04430 ANI 1 ;Return indicating no more bits set
0675 C8 04431 JB SDT $Bit $Set ;Yes, return corresponding address
0676 13 04432 INX D ;No, update table pointer
0677 13 04433 INX D
0678 13 04434 INX D
0679 13 04435 CALL SHLR ;Shift HL right one bit
067A CD 04436 JMP $Select Device Table ;Check next bit
067B 0F 04437 SDT $Bit $Set:

0680 E5 04438 PUSH H ;Save shifted bit map
0681 42 04439 MOV B, D ;Take copy of table pointer
0682 4B 04440 MOV C, E
0683 EB 04441 XOR H ;HL -> address in table
0684 5E 04442 MOV E, M
0685 23 04443 INX H
0686 56 04444 MOV D, M ;DE -> selected device table
0687 01 04445 MOV A, C ;Set up registers for another
0688 E1 04446 ;entry
0689 CD 04447 POP H ;Recover shifted bit map
068A 0F 04448 CALL SHLR ;Shift bit map right one bit
068B 03 04449 INX B ;Update DT address pointer to
068C 03 04450 INX C
068D 03 04451 INX D
068E 03 04452 ORA A ;Indicate that a one bit was found
068F B7 04453 RET ;and registers are set up correctly

0690 C9 04454 ;
0690 00 04455 ;
0690 01 04456 ;
0690 02 04457 ;
0690 03 04458 ;
0690 04 04459 ;
0690 05 04460 ;
0690 06 04461 ;
0690 07 04462 ;
0690 08 04463 ;
0691 21 04464 ;Get input character
0694 19 04465 ;
0694 12 04466 ;
0694 11 04467 ;GIC $Wait:
0695 FB 04468 ;
0696 7E 04469 ;
0697 B7 04470 ;
0698 CA 04471 ;
0699 25 04472 ;
069A 0A 04473 ;
069B 0F 04474 ;
069C 21 04475 ;
069F CD 04476 ;
06A2 7E 04477 ;
06A3 F5 04478 ;
06A4 21 04479 ;
06A7 19 04480 ;

0691 21 04464 GET $Input $Character:

0694 19 04465 DAD D ;Check if any characters have
0694 12 04466 been stored in the buffer
0694 11 04467 ;GIC $Wait:
0695 FB 04468 ;Ensure that incoming chars. will
0696 7E 04469 ;be detected
0697 B7 04470 ;Get character count
0698 CA 04471 ;No characters, so wait
0699 25 04472 ;Down date character count for
069A 0A 04473 ;the character about to be
069B 0F 04474 ;removed from the buffer
069C 21 04475 ;and with get offset updated
069F CD 04476 ;Returns HL -> character
06A2 7E 04477 ;Get the actual data character
06A3 F5 04478 ;Save until later
06A4 21 04479 ;Check down dated count of chars. in
06A7 19 04480 ;buffer, checking if input should be

Figure 8.10. (Continued)
ServiceDevice:

0716 210000 0717 LXI H,DT$Status$Port ;Check if device 0719 19 0718 MOV A,M 071A 19 0716 STA SD$Status$Port ;Store in instruction below 071B 321F07 071D 0520 071F 00 0722 DB IN ;Input status 0721 210300 0723 19 0724 DB 0 ;Set up by instruction above 0725 0525 ;
0726 210700 0727 19 0728 LXI H,DT$Detect$Error$Port ;Check if error status 0729 19 072A MOV A,M 072B 325F07 072C 0524 072E DB IN ;Input error status 072F 00 0730 0537 ;Set up by instruction above 0732 210800 0733 19 0734 DB 0 ;Set up by instruction above 0735 CA707 0736 19 0737 DB OUT 0746 00 0748 7E 074C 325007 ;Set up in instruction above 0749 0557 074A 19 074B 7E 074C 325007 ;Store in instruction below

Figure 8-10. (Continued)
### Figure 8-10. (Continued)

<table>
<thead>
<tr>
<th>CR</th>
<th>Address</th>
<th>Machine Code</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>074F</td>
<td>05061</td>
<td>DB</td>
<td>SD$DataPort</td>
<td>Input data character</td>
</tr>
<tr>
<td>0750</td>
<td>05066</td>
<td>MOV</td>
<td>B, A</td>
<td>Take copy of data character above</td>
</tr>
<tr>
<td>0751</td>
<td>05064</td>
<td>DB</td>
<td>0</td>
<td>&lt;--- Set up by instruction above</td>
</tr>
<tr>
<td>0752</td>
<td>05067</td>
<td>LXI</td>
<td>H, DT$Status</td>
<td>Check if either XON or ETX protocols</td>
</tr>
<tr>
<td>0753</td>
<td>05068</td>
<td>DAD</td>
<td>D</td>
<td>is currently active</td>
</tr>
<tr>
<td>0754</td>
<td>05069</td>
<td>MOV</td>
<td>A, M</td>
<td>Get protocol byte</td>
</tr>
<tr>
<td>0755</td>
<td>05070</td>
<td>ANI</td>
<td>ANI D4$Output<em>Xon + DT$Output</em>Ex</td>
<td>Check if XON/XOFF is active</td>
</tr>
<tr>
<td>0756</td>
<td>05071</td>
<td>JZ</td>
<td>JZ SDNo$Protocol</td>
<td>Neither is active</td>
</tr>
<tr>
<td>0757</td>
<td>05072</td>
<td>ANI</td>
<td>ANI DT$Output*Xon</td>
<td>Yes, check if XON char. input</td>
</tr>
<tr>
<td>0758</td>
<td>05073</td>
<td>JNZ</td>
<td>JNZ SD$Check<em>if</em>Xon</td>
<td>No, assume ETT/ACK active</td>
</tr>
<tr>
<td>0759</td>
<td>05074</td>
<td>MVI</td>
<td>A, ACK</td>
<td>Check if input character is ACK</td>
</tr>
<tr>
<td>0760</td>
<td>05075</td>
<td>CMP</td>
<td>B</td>
<td>Check if suspend output</td>
</tr>
<tr>
<td>0761</td>
<td>05076</td>
<td>JNZ</td>
<td>JNZ SDNo$Protocol</td>
<td>No, process character as data</td>
</tr>
<tr>
<td>0762</td>
<td>05077</td>
<td>SD$Output*Desuspend</td>
<td>Yes, device now ready</td>
<td></td>
</tr>
<tr>
<td>0763</td>
<td>05078</td>
<td>MOV</td>
<td>M, A</td>
<td>Get status/protocol byte again</td>
</tr>
<tr>
<td>0764</td>
<td>05079</td>
<td>CMP</td>
<td>B</td>
<td>XON/XOFF protocol active, so</td>
</tr>
<tr>
<td>0765</td>
<td>05080</td>
<td>CMP</td>
<td>B</td>
<td>Yes, enable output to device</td>
</tr>
<tr>
<td>0766</td>
<td>05081</td>
<td>CMP</td>
<td>B</td>
<td>No, process character as data</td>
</tr>
<tr>
<td>0767</td>
<td>05082</td>
<td>CMP</td>
<td>B</td>
<td>Save back with suspend = 0</td>
</tr>
<tr>
<td>0768</td>
<td>05083</td>
<td>MOV</td>
<td>B, M</td>
<td>Load status/protocol byte again</td>
</tr>
<tr>
<td>0769</td>
<td>05084</td>
<td>CMP</td>
<td>B</td>
<td>Get status/protocol byte again</td>
</tr>
<tr>
<td>076A</td>
<td>05085</td>
<td>MOV</td>
<td>B, A</td>
<td>Set suspend bit to 1</td>
</tr>
<tr>
<td>076B</td>
<td>05086</td>
<td>JMP</td>
<td>SD$Exit</td>
<td>Exit to interrupt service without</td>
</tr>
<tr>
<td>076C</td>
<td>05087</td>
<td>SD$Output*Desuspend</td>
<td>Yes, device now ready</td>
<td></td>
</tr>
<tr>
<td>076D</td>
<td>05088</td>
<td>MOV</td>
<td>B, M</td>
<td>Get status/protocol byte again</td>
</tr>
<tr>
<td>076E</td>
<td>05089</td>
<td>CMP</td>
<td>B</td>
<td>Check if XON character input</td>
</tr>
<tr>
<td>076F</td>
<td>05090</td>
<td>CMP</td>
<td>B</td>
<td>No, process character as data</td>
</tr>
<tr>
<td>0770</td>
<td>05091</td>
<td>CMP</td>
<td>B</td>
<td>Save back in device table</td>
</tr>
<tr>
<td>0771</td>
<td>05092</td>
<td>CMP</td>
<td>B</td>
<td>Exit to interrupt service without</td>
</tr>
<tr>
<td>0772</td>
<td>05093</td>
<td>CMP</td>
<td>B</td>
<td>saving the input character</td>
</tr>
<tr>
<td>0773</td>
<td>05094</td>
<td>MVI</td>
<td>A, XON</td>
<td>Save to this status/protocol byte</td>
</tr>
<tr>
<td>0774</td>
<td>05095</td>
<td>JMP</td>
<td>SD$Exit</td>
<td>Exit to interrupt service without</td>
</tr>
<tr>
<td>0775</td>
<td>05096</td>
<td>CMP</td>
<td>B</td>
<td>saving the input character</td>
</tr>
<tr>
<td>0776</td>
<td>05097</td>
<td>CMP</td>
<td>B</td>
<td>No, process character as data</td>
</tr>
<tr>
<td>0777</td>
<td>05098</td>
<td>CMP</td>
<td>B</td>
<td>XON/XOFF protocol active, so</td>
</tr>
<tr>
<td>0778</td>
<td>05099</td>
<td>JNZ</td>
<td>JNZ SDNo$Protocol</td>
<td>No, process character as data</td>
</tr>
<tr>
<td>0779</td>
<td>05100</td>
<td>SD$Output*Desuspend</td>
<td>Yes, device now ready</td>
<td></td>
</tr>
<tr>
<td>077A</td>
<td>05101</td>
<td>MOV</td>
<td>M, A</td>
<td>Save back with suspend = 0</td>
</tr>
<tr>
<td>077B</td>
<td>05102</td>
<td>CMP</td>
<td>B</td>
<td>Exit to interrupt service without</td>
</tr>
<tr>
<td>077C</td>
<td>05103</td>
<td>CMP</td>
<td>B</td>
<td>saving the input character</td>
</tr>
<tr>
<td>077D</td>
<td>05104</td>
<td>MOV</td>
<td>B, M</td>
<td>Get status/protocol byte again</td>
</tr>
<tr>
<td>077E</td>
<td>05105</td>
<td>CMP</td>
<td>B</td>
<td>Check if XON character input</td>
</tr>
<tr>
<td>077F</td>
<td>05106</td>
<td>CMP</td>
<td>B</td>
<td>No, process character as data</td>
</tr>
<tr>
<td>0780</td>
<td>05107</td>
<td>CMP</td>
<td>B</td>
<td>Save back in device table</td>
</tr>
<tr>
<td>0781</td>
<td>05108</td>
<td>CMP</td>
<td>B</td>
<td>Exit to interrupt service without</td>
</tr>
<tr>
<td>0782</td>
<td>05109</td>
<td>LMX</td>
<td>H, DT$Buffer<em>Length</em>Mask</td>
<td>Check if there is still space</td>
</tr>
<tr>
<td>0783</td>
<td>05110</td>
<td>DAD</td>
<td>D</td>
<td>in the input buffer so</td>
</tr>
<tr>
<td>0784</td>
<td>05111</td>
<td>MOV</td>
<td>B, M</td>
<td>get length - 1</td>
</tr>
<tr>
<td>0785</td>
<td>05112</td>
<td>INR</td>
<td>A</td>
<td>Update to actual length</td>
</tr>
<tr>
<td>0786</td>
<td>05113</td>
<td>LXI</td>
<td>H, DT$Character*Count</td>
<td>Get current count of characters</td>
</tr>
<tr>
<td>0787</td>
<td>05114</td>
<td>DAD</td>
<td>D</td>
<td>in buffer</td>
</tr>
<tr>
<td>0788</td>
<td>05115</td>
<td>CMP</td>
<td>M</td>
<td>Check if count = length</td>
</tr>
<tr>
<td>0789</td>
<td>05116</td>
<td>JZ</td>
<td>JZ SD$Buffer*Full</td>
<td>Yes, output bell character</td>
</tr>
<tr>
<td>078A</td>
<td>05117</td>
<td>PUSH</td>
<td>B</td>
<td>Save data character</td>
</tr>
<tr>
<td>078B</td>
<td>05118</td>
<td>LXI</td>
<td>H, DT$Put*Offset</td>
<td>Compute address of character in</td>
</tr>
<tr>
<td>078C</td>
<td>05119</td>
<td>CALL</td>
<td>Get<em>Address</em>In$Buffer</td>
<td>input buffer</td>
</tr>
<tr>
<td>078D</td>
<td>05120</td>
<td>POP</td>
<td>B</td>
<td>Recover input character</td>
</tr>
<tr>
<td>078E</td>
<td>05121</td>
<td>MOV</td>
<td>M, B</td>
<td>Save character in input buffer</td>
</tr>
<tr>
<td>078F</td>
<td>05122</td>
<td>CMP</td>
<td>B</td>
<td>Update number of characters in input</td>
</tr>
<tr>
<td>0790</td>
<td>05123</td>
<td>LXI</td>
<td>H, DT$Character*Count</td>
<td>buffer, checking if input should</td>
</tr>
<tr>
<td>0791</td>
<td>05124</td>
<td>MPB</td>
<td>B</td>
<td>be temporarily halted</td>
</tr>
<tr>
<td>0792</td>
<td>05125</td>
<td>LXI</td>
<td>H, DT$Character*Count</td>
<td>Update character count</td>
</tr>
<tr>
<td>0793</td>
<td>05126</td>
<td>DAD</td>
<td>D</td>
<td>Get updated count</td>
</tr>
<tr>
<td>0794</td>
<td>05127</td>
<td>MOV</td>
<td>B, M</td>
<td>Check if current count matches</td>
</tr>
<tr>
<td>0795</td>
<td>05128</td>
<td>CMP</td>
<td>B</td>
<td>buffer-full threshold</td>
</tr>
<tr>
<td>0796</td>
<td>05129</td>
<td>CMP</td>
<td>B</td>
<td>Not at threshold, check if control</td>
</tr>
<tr>
<td>0797</td>
<td>05130</td>
<td>JMP</td>
<td>JNZ SD$Check*Control</td>
<td>Character input</td>
</tr>
<tr>
<td>0798</td>
<td>05131</td>
<td>MOV</td>
<td>B, M</td>
<td>Save character in input buffer</td>
</tr>
<tr>
<td>0799</td>
<td>05132</td>
<td>CMP</td>
<td>B</td>
<td>Update number of characters in input</td>
</tr>
<tr>
<td>079A</td>
<td>05133</td>
<td>LXI</td>
<td>H, DT$Status</td>
<td>buffer, checking if input should</td>
</tr>
<tr>
<td>079B</td>
<td>05134</td>
<td>DAD</td>
<td>D</td>
<td>be temporarily halted</td>
</tr>
</tbody>
</table>

Note: The code above is a part of the BIOS and handles input/output operations, including checking for XON and XOFF protocols. The instructions are designed to manage buffer states, character counts, and resume processes after buffer-full or suspend conditions.
Get$Address$In$Buffer:
This routine computes the address of the next character to access in a device buffer.

<table>
<thead>
<tr>
<th>Address</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A,M</td>
<td>Get status/protocol byte</td>
</tr>
<tr>
<td>PUSH PSW</td>
<td>Save updated status in table</td>
</tr>
<tr>
<td>ANI DT$Input$RTS</td>
<td>Check if clear to send to be dropped</td>
</tr>
<tr>
<td>JZ SD$Check$Input$Xon</td>
<td>No, get control port number</td>
</tr>
<tr>
<td>LDA D</td>
<td>Reset H,DT Reset$Int$Port o A,M</td>
</tr>
<tr>
<td>MOV A,M</td>
<td>Store in instruction below</td>
</tr>
<tr>
<td>STA SD$ResetSIntSPort</td>
<td>Store in instruction below</td>
</tr>
<tr>
<td>MOV A,M</td>
<td>Get value needed to drop RTS</td>
</tr>
<tr>
<td>DB OUT</td>
<td></td>
</tr>
<tr>
<td>CALL SD$Check$Control</td>
<td>Check if control character (other than CR, LF, or TAB) input, and update control character count in buffer</td>
</tr>
<tr>
<td>MOV A,M</td>
<td>Yes, output XOFF character</td>
</tr>
<tr>
<td>ORL H.DTSInputXon</td>
<td>Check if clear to send</td>
</tr>
</tbody>
</table>
Chapter 8: Writing an Enhanced BIOS

[Code Listing]

Figure 8-10. (Continued)
Failure to do this may cause involuntary re-entrance.

Entry parameters

C = character to be output
DE -> device table

Save registers

Get output ready status mask; store in instruction below

Get status port number

Read status

Restore registers

Check if readY for output, No

Get data port

Check if fake mode enabled
HL - out status byte in table

Status byte in table

Isolate status bit

Fake mode disabled

Fake mode -- only indicates data ready if control chars. in buffer

Check if any control characters ready in the input buffer

This routine returns a value in the A register indicating whether one or more data characters is/are waiting in the input buffer.

Some products, such as Microsoft BASIC, defeat normal type-ahead by constantly "gobbling" characters in order to see if an incoming Control-S, -Q or -C has been received. In order to preserve type-ahead under these circumstances, the input status return can, as an option selected by the user, return "data waiting" only if the input buffer contains a Control-S, -Q or -C. This fools Microsoft BASIC into allowing type-ahead.

Failure to do this may cause involuntary re-entrance.

Figure 8·10. (Continued)
Chapter 8: Writing an Enhanced BIOS 267

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0856 B4</td>
<td>ORA M</td>
<td>; Set flags according to count</td>
</tr>
<tr>
<td>0857 C0</td>
<td>RZ</td>
<td>; Return indicating zero</td>
</tr>
<tr>
<td>0859 AF</td>
<td>XRA A</td>
<td>; Check if forced input waiting</td>
</tr>
<tr>
<td>085A C9</td>
<td>DCR A</td>
<td>; Set a = OFFH and flags NZ</td>
</tr>
<tr>
<td>085C 2A</td>
<td>MOV A.M</td>
<td>; Get next character of forced input</td>
</tr>
<tr>
<td>0860 C3</td>
<td>JNZ GIS$Data$Ready</td>
<td>; Yes, indicate data waiting</td>
</tr>
<tr>
<td>0863 21</td>
<td>LXI H,RTC$Tick$Count</td>
<td>; Check if any characters ready in input buffer</td>
</tr>
<tr>
<td>0866 19</td>
<td>DAD D</td>
<td>; Check if any characters in buffer</td>
</tr>
<tr>
<td>0867 7E</td>
<td>MOV A.M</td>
<td>; Get character count</td>
</tr>
<tr>
<td>0868 B7</td>
<td>ORA A</td>
<td>; Set character count</td>
</tr>
<tr>
<td>0869 C8</td>
<td>JMP GIS$Data$Ready</td>
<td>; Empty buffer, A = 0, Z-set</td>
</tr>
<tr>
<td>086A C3</td>
<td>MOV A.M</td>
<td>; Return to caller</td>
</tr>
<tr>
<td>086C 2B</td>
<td>SHLD L,RTC$Watchdog$Address</td>
<td>; Set address</td>
</tr>
<tr>
<td>086F C2</td>
<td>SHLD L,RTC$Watchdog$Count</td>
<td>; Set count</td>
</tr>
<tr>
<td>0871 60</td>
<td>MOV H,B</td>
<td></td>
</tr>
<tr>
<td>0872 69</td>
<td>MOV L,C</td>
<td></td>
</tr>
<tr>
<td>0873 22</td>
<td>SHLD RTC$Watchdog$Address</td>
<td>; Set address</td>
</tr>
<tr>
<td>0874 F8</td>
<td>E1</td>
<td></td>
</tr>
<tr>
<td>0877 C9</td>
<td>RET</td>
<td></td>
</tr>
<tr>
<td>0879 22</td>
<td>SHLD PI$User$HL</td>
<td>; Switch to local stack</td>
</tr>
<tr>
<td>087C 21</td>
<td>LXI H,0</td>
<td></td>
</tr>
<tr>
<td>087F 3F</td>
<td>MOV D,SP</td>
<td>; Get user's stack</td>
</tr>
<tr>
<td>0880 22</td>
<td>SHLD PI$User$Stack</td>
<td>; Save it</td>
</tr>
<tr>
<td>0883 31</td>
<td>LXI SP,PI$Stack</td>
<td>; Switch to local stack</td>
</tr>
<tr>
<td>0886 C8</td>
<td>PUSH B</td>
<td></td>
</tr>
<tr>
<td>0887 D5</td>
<td>PUSH D</td>
<td></td>
</tr>
<tr>
<td>0888 21</td>
<td>LXI H,RTC$Tick$Count</td>
<td>; Downdate tick count</td>
</tr>
<tr>
<td>0889 22</td>
<td>PUSH PSW</td>
<td></td>
</tr>
<tr>
<td>088A 31</td>
<td>LXI SP,PSW</td>
<td></td>
</tr>
<tr>
<td>088B 22</td>
<td>LXI SP,DPL</td>
<td></td>
</tr>
<tr>
<td>088C 21</td>
<td>LXI SP,PSW</td>
<td></td>
</tr>
<tr>
<td>088D 22</td>
<td>LXI SP,PSW</td>
<td></td>
</tr>
<tr>
<td>088E 22</td>
<td>LXI SP,PSW</td>
<td></td>
</tr>
<tr>
<td>088F 22</td>
<td>LXI SP,PSW</td>
<td></td>
</tr>
<tr>
<td>0890 22</td>
<td>LXI SP,PSW</td>
<td></td>
</tr>
<tr>
<td>0891 22</td>
<td>LXI SP,PSW</td>
<td></td>
</tr>
<tr>
<td>0892 22</td>
<td>LXI SP,PSW</td>
<td></td>
</tr>
<tr>
<td>0893 22</td>
<td>LXI SP,PSW</td>
<td></td>
</tr>
<tr>
<td>0894 22</td>
<td>LXI SP,PSW</td>
<td></td>
</tr>
<tr>
<td>0895 22</td>
<td>LXI SP,PSW</td>
<td></td>
</tr>
<tr>
<td>0896 22</td>
<td>LXI SP,PSW</td>
<td></td>
</tr>
<tr>
<td>0897 22</td>
<td>LXI SP,PSW</td>
<td></td>
</tr>
<tr>
<td>0898 22</td>
<td>LXI SP,PSW</td>
<td></td>
</tr>
<tr>
<td>0899 22</td>
<td>LXI SP,PSW</td>
<td></td>
</tr>
<tr>
<td>089A 22</td>
<td>LXI SP,PSW</td>
<td></td>
</tr>
<tr>
<td>089B 22</td>
<td>LXI SP,PSW</td>
<td></td>
</tr>
<tr>
<td>089C 22</td>
<td>LXI SP,PSW</td>
<td></td>
</tr>
<tr>
<td>089D 22</td>
<td>LXI SP,PSW</td>
<td></td>
</tr>
<tr>
<td>089E 22</td>
<td>LXI SP,PSW</td>
<td></td>
</tr>
<tr>
<td>089F 22</td>
<td>LXI SP,PSW</td>
<td></td>
</tr>
<tr>
<td>08A0 22</td>
<td>LXI SP,PSW</td>
<td></td>
</tr>
<tr>
<td>08A1 22</td>
<td>LXI SP,PSW</td>
<td></td>
</tr>
<tr>
<td>08A2 22</td>
<td>LXI SP,PSW</td>
<td></td>
</tr>
<tr>
<td>08A3 22</td>
<td>LXI SP,PSW</td>
<td></td>
</tr>
<tr>
<td>08A4 22</td>
<td>LXI SP,PSW</td>
<td></td>
</tr>
<tr>
<td>08A5 22</td>
<td>LXI SP,PSW</td>
<td></td>
</tr>
<tr>
<td>08A6 22</td>
<td>LXI SP,PSW</td>
<td></td>
</tr>
<tr>
<td>08A7 22</td>
<td>LXI SP,PSW</td>
<td></td>
</tr>
<tr>
<td>08A8 22</td>
<td>LXI SP,PSW</td>
<td></td>
</tr>
<tr>
<td>08A9 22</td>
<td>LXI SP,PSW</td>
<td></td>
</tr>
<tr>
<td>08AA 22</td>
<td>LXI SP,PSW</td>
<td></td>
</tr>
<tr>
<td>08AB 22</td>
<td>LXI SP,PSW</td>
<td></td>
</tr>
<tr>
<td>08AC 22</td>
<td>LXI SP,PSW</td>
<td></td>
</tr>
<tr>
<td>08AD 22</td>
<td>LXI SP,PSW</td>
<td></td>
</tr>
<tr>
<td>08AE 22</td>
<td>LXI SP,PSW</td>
<td></td>
</tr>
<tr>
<td>08AF 22</td>
<td>LXI SP,PSW</td>
<td></td>
</tr>
<tr>
<td>08B0 22</td>
<td>LXI SP,PSW</td>
<td></td>
</tr>
<tr>
<td>08B1 22</td>
<td>LXI SP,PSW</td>
<td></td>
</tr>
<tr>
<td>08B2 22</td>
<td>LXI SP,PSW</td>
<td></td>
</tr>
<tr>
<td>08B3 22</td>
<td>LXI SP,PSW</td>
<td></td>
</tr>
<tr>
<td>08B4 22</td>
<td>LXI SP,PSW</td>
<td></td>
</tr>
<tr>
<td>08B5 22</td>
<td>LXI SP,PSW</td>
<td></td>
</tr>
<tr>
<td>08B6 22</td>
<td>LXI SP,PSW</td>
<td></td>
</tr>
<tr>
<td>08B7 22</td>
<td>LXI SP,PSW</td>
<td></td>
</tr>
<tr>
<td>08B8 22</td>
<td>LXI SP,PSW</td>
<td></td>
</tr>
<tr>
<td>08B9 22</td>
<td>LXI SP,PSW</td>
<td></td>
</tr>
<tr>
<td>08BA 22</td>
<td>LXI SP,PSW</td>
<td></td>
</tr>
<tr>
<td>08BB 22</td>
<td>LXI SP,PSW</td>
<td></td>
</tr>
<tr>
<td>08BC 22</td>
<td>LXI SP,PSW</td>
<td></td>
</tr>
<tr>
<td>08BD 22</td>
<td>LXI SP,PSW</td>
<td></td>
</tr>
<tr>
<td>08BE 22</td>
<td>LXI SP,PSW</td>
<td></td>
</tr>
<tr>
<td>08BF 22</td>
<td>LXI SP,PSW</td>
<td></td>
</tr>
<tr>
<td>08C0 22</td>
<td>LXI SP,PSW</td>
<td></td>
</tr>
<tr>
<td>08C1 22</td>
<td>LXI SP,PSW</td>
<td></td>
</tr>
<tr>
<td>08C2 22</td>
<td>LXI SP,PSW</td>
<td></td>
</tr>
<tr>
<td>08C3 22</td>
<td>LXI SP,PSW</td>
<td></td>
</tr>
<tr>
<td>08C4 22</td>
<td>LXI SP,PSW</td>
<td></td>
</tr>
<tr>
<td>08C5 22</td>
<td>LXI SP,PSW</td>
<td></td>
</tr>
<tr>
<td>08C6 22</td>
<td>LXI SP,PSW</td>
<td></td>
</tr>
<tr>
<td>08C7 22</td>
<td>LXI SP,PSW</td>
<td></td>
</tr>
<tr>
<td>08C8 22</td>
<td>LXI SP,PSW</td>
<td></td>
</tr>
<tr>
<td>08C9 22</td>
<td>LXI SP,PSW</td>
<td></td>
</tr>
<tr>
<td>08CA 22</td>
<td>LXI SP,PSW</td>
<td></td>
</tr>
<tr>
<td>08CB 22</td>
<td>LXI SP,PSW</td>
<td></td>
</tr>
<tr>
<td>08CC 22</td>
<td>LXI SP,PSW</td>
<td></td>
</tr>
<tr>
<td>08CD 22</td>
<td>LXI SP,PSW</td>
<td></td>
</tr>
<tr>
<td>08CE 22</td>
<td>LXI SP,PSW</td>
<td></td>
</tr>
<tr>
<td>08CF 22</td>
<td>LXI SP,PSW</td>
<td></td>
</tr>
<tr>
<td>08D0 22</td>
<td>LXI SP,PSW</td>
<td></td>
</tr>
<tr>
<td>08D1 22</td>
<td>LXI SP,PSW</td>
<td></td>
</tr>
<tr>
<td>08D2 22</td>
<td>LXI SP,PSW</td>
<td></td>
</tr>
<tr>
<td>08D3 22</td>
<td>LXI SP,PSW</td>
<td></td>
</tr>
<tr>
<td>08D4 22</td>
<td>LXI SP,PSW</td>
<td></td>
</tr>
<tr>
<td>08D5 22</td>
<td>LXI SP,PSW</td>
<td></td>
</tr>
<tr>
<td>08D6 22</td>
<td>LXI SP,PSW</td>
<td></td>
</tr>
<tr>
<td>08D7 22</td>
<td>LXI SP,PSW</td>
<td></td>
</tr>
<tr>
<td>08D8 22</td>
<td>LXI SP,PSW</td>
<td></td>
</tr>
<tr>
<td>08D9 22</td>
<td>LXI SP,PSW</td>
<td></td>
</tr>
<tr>
<td>08DA 22</td>
<td>LXI SP,PSW</td>
<td></td>
</tr>
<tr>
<td>08DB 22</td>
<td>LXI SP,PSW</td>
<td></td>
</tr>
<tr>
<td>08DC 22</td>
<td>LXI SP,PSW</td>
<td></td>
</tr>
<tr>
<td>08DD 22</td>
<td>LXI SP,PSW</td>
<td></td>
</tr>
<tr>
<td>08DE 22</td>
<td>LXI SP,PSW</td>
<td></td>
</tr>
<tr>
<td>08DF 22</td>
<td>LXI SP,PSW</td>
<td></td>
</tr>
<tr>
<td>08E0 22</td>
<td>LXI SP,PSW</td>
<td></td>
</tr>
<tr>
<td>08E1 22</td>
<td>LXI SP,PSW</td>
<td></td>
</tr>
<tr>
<td>08E2 22</td>
<td>LXI SP,PSW</td>
<td></td>
</tr>
<tr>
<td>08E3 22</td>
<td>LXI SP,PSW</td>
<td></td>
</tr>
<tr>
<td>08E4 22</td>
<td>LXI SP,PSW</td>
<td></td>
</tr>
<tr>
<td>08E5 22</td>
<td>LXI SP,PSW</td>
<td></td>
</tr>
<tr>
<td>08E6 22</td>
<td>LXI SP,PSW</td>
<td></td>
</tr>
<tr>
<td>08E7 22</td>
<td>LXI SP,PSW</td>
<td></td>
</tr>
<tr>
<td>08E8 22</td>
<td>LXI SP,PSW</td>
<td></td>
</tr>
<tr>
<td>08E9 22</td>
<td>LXI SP,PSW</td>
<td></td>
</tr>
<tr>
<td>08EA 22</td>
<td>LXI SP,PSW</td>
<td></td>
</tr>
<tr>
<td>08EB 22</td>
<td>LXI SP,PSW</td>
<td></td>
</tr>
<tr>
<td>08EC 22</td>
<td>LXI SP,PSW</td>
<td></td>
</tr>
<tr>
<td>08ED 22</td>
<td>LXI SP,PSW</td>
<td></td>
</tr>
<tr>
<td>08EE 22</td>
<td>LXI SP,PSW</td>
<td></td>
</tr>
<tr>
<td>08EF 22</td>
<td>LXI SP,PSW</td>
<td></td>
</tr>
</tbody>
</table>

Real time clock processing

Control is transferred to the RTC$Interrupt routine each time the real time clock ticks. The tick count is downdated to see if a complete second has elapsed. If so, the ASCII time in the configuration block is updated.

With each tick, the watchdog count is downdated to see if control must be "forced" to the previously specified address on return from the RTC interrupt. The watchdog timer can be used to pull control out of what would otherwise be an infinite loop, such as waiting for the printer to come ready.

Set watchdog

This is a noninterrupt level subroutine that simply sets the watchdog count and address.

Entry parameters

BC = number of clock ticks before watchdog should "time out"
HL = address from which control will be transferred when watchdog times out

SetWatchdog:

Avoid interference from interrupts
Set address

; Control is received here each time the real time clock ticks

Figure 8-10. (Continued)
Figure 8-10.  (Continued)
Chapter 8: Writing an Enhanced BIOS 269

Disk definition tables

These consist of disk parameter headers, with one entry per logical disk driver, and disk parameter blocks with either one parameter block per logical disk, or the same parameter block for several logical disks.

<table>
<thead>
<tr>
<th>Address</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>06301</td>
<td>High level diskette drivers</td>
</tr>
<tr>
<td>06302</td>
<td>These drivers perform the following functions:</td>
</tr>
<tr>
<td>06303</td>
<td>SELDSK Select a specified disk and return the address of the appropriate disk parameter header</td>
</tr>
<tr>
<td>06304</td>
<td>SETTRK Set the track number for the next read or write</td>
</tr>
<tr>
<td>06305</td>
<td>SETSEC Set the sector number for the next read or write</td>
</tr>
<tr>
<td>06306</td>
<td>SETDMA Set the DMA (read/write) address for the next read or write</td>
</tr>
<tr>
<td>06307</td>
<td>SECTRN Translate a logical sector number into a physical</td>
</tr>
<tr>
<td>06308</td>
<td>HOME Set the track to 0 so that the next read or write will be on Track 0</td>
</tr>
<tr>
<td>06309</td>
<td>In addition, the high level drivers are responsible for making the 5 1/4&quot; floppy diskettes that use a 512-byte sector appear to CP/M as though they used a 128-byte sector. They do this by using blocking/deblocking code. This blocking/deblocking code is described in more detail later in this listing, just prior to the code itself.</td>
</tr>
</tbody>
</table>

Disk parameter tables

As discussed in Chapter 3, these describe the physical characteristics of the disk drives. In this example BIOS, there are two types of disk drives: standard single-sided, single-density 8", and double-sided, double-density 5 1/4" mini-diskettes.

- The standard 8" diskettes do not need to use the blocking/deblocking code, but the 5 1/4" drives do. Therefore an additional byte has been prefixed onto the disk parameter block to tell the disk drivers what each logical disk's physical diskette type is, and whether or not it needs deblocking.

Disk definition tables

<table>
<thead>
<tr>
<th>Address</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0630A</td>
<td>Logical disk A (5 1/4&quot; diskette)</td>
</tr>
<tr>
<td>0630B</td>
<td>Floppy$5$Skewtable 15 1/4&quot; skew table</td>
</tr>
<tr>
<td>0630C</td>
<td>0,0,0 Reserved for CP/M</td>
</tr>
<tr>
<td>0630D</td>
<td>Directory$Buffer</td>
</tr>
<tr>
<td>0630E</td>
<td>Floppy$5$Parameter$Block</td>
</tr>
<tr>
<td>0630F</td>
<td>Disk$A$Workarea</td>
</tr>
<tr>
<td>06310</td>
<td>Disk$A$Allocation$Vector</td>
</tr>
<tr>
<td>06311</td>
<td>Logical disk B (5 1/4&quot; diskette)</td>
</tr>
</tbody>
</table>
| 06312    | Floppy$5$Skewtable Shares same skew table as A:
| 06313    | 0,0,0 Reserved for CP/M |
| 06314    | Directory$Buffer Shares same buffer as A: |
| 06315    | Floppy$5$Parameter$Block |
| 06316    | Disk$B$Workarea Private work area |
| 06317    | Disk$B$Allocation$Vector Private allocation vector |
| 06318    | Logical disk C (8" floppy) |
| 06319    | Floppy$5$Skewtable 8" skew table |
| 0631A    | 0,0,0 Reserved for CP/M |
| 0631B    | Directory$Buffer Shares same buffer as A: |
| 0631C    | Floppy$5$Parameter$Block |
| 0631D    | Disk$C$Workarea Private work area |
| 0631E    | Disk$C$Allocation$Vector Private allocation vector |
| 0631F    | Logical disk D (8" floppy) |
| 06320    | Floppy$5$Skewtable Shares same skew table as A: |
| 06321    | 0,0,0 Reserved for CP/M |
| 06322    | Directory$Buffer Shares same buffer as A: |
| 06323    | Floppy$5$Parameter$Block |
| 06324    | Disk$D$Workarea Private work area |
| 06325    | Disk$D$Allocation$Vector Private allocation vector |

Figure 8-10. (Continued)
The parameter block has been amended to reflect the new layout of one track per diskette side, rather than viewing one track as both sides on a given head position. It has also been adjusted to reflect one "new" track more being used for the CP/M image, with the resulting change in the number of allocation blocks and the number of reserved tracks.

<table>
<thead>
<tr>
<th>Address</th>
<th>Opcode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>091D 4409</td>
<td>0E 432</td>
<td>Floppy$8$Parameter$Block</td>
</tr>
<tr>
<td>091F 0024</td>
<td>0E 433</td>
<td>Disk$8$Workarea</td>
</tr>
<tr>
<td>0921 5B24</td>
<td>0E 434</td>
<td>Disk$8$Allocation$Vector</td>
</tr>
<tr>
<td>0925</td>
<td>0E 435</td>
<td>(Logical disk M: (memory disk)</td>
</tr>
<tr>
<td>0923 0000</td>
<td>0E 436</td>
<td>DW</td>
</tr>
<tr>
<td>0925 000000000439</td>
<td>0E 437</td>
<td>DW</td>
</tr>
<tr>
<td>0928 B022</td>
<td>0E 440</td>
<td>DW</td>
</tr>
<tr>
<td>092D 3409</td>
<td>0E 441</td>
<td>DW</td>
</tr>
<tr>
<td>092F 0000</td>
<td>0E 442</td>
<td>DW</td>
</tr>
<tr>
<td>0931 7A24</td>
<td>0E 444</td>
<td>DW</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| | EQU | EQU | EQU |
| 0001 | | | | Floppy$8$ EQU 1 | = 5 1/4" mini floppy |
| 0002 | | | | Floppy$8$ EQU 2 | = 8" floppy (SS SD) |
| 0003 | | | | M$8$Disk EQU 3 | = Memory disk |

| | EQU | EQU | EQU |
| 0645 | | | |
| 0646 | | | |
| 0647 | | | Equates for disk parameter block |
| 0648 | | | Disk Types |
| 0649 | | | Block$8$Unblocking indicator |

| | EQU | EQU | EQU |
| 0650 | | | |
| 0651 | | | |
| 0652 | | | |
| 0653 | | | |

| Need$8$Unblocking EQU | 1000$8$0000B | = Vector size > 128 bytes |

| | EQU | EQU | EQU |
| 0658 | | | |
| 0659 | | | |
| 0660 | | | |
| 0661 | | | |
| 0662 | | | |
| 0663 | | | |

| Extra byte prefixed to indicate disk type and blocking required |
| | disk type and blocking required |

| | Floppy$8$Parameter$Block: |
| | The parameter block has been amended |
| | to reflect the new layout of one track per diskette side, rather than viewing one track as both sides on a given head position. It has also been adjusted to reflect one "new" track more being used for the CP/M image, with the resulting change in the number of allocation blocks and the number of reserved tracks. |

| | Floppy$8$Parameter$Block: |
| | 128-byte sectors per track |

| | Floppy$8$Parameter$Block: |
| | Extra byte prefixed to DPB for this version of the BIOS |

| | Floppy$8$Parameter$Block: |
| | Indicates disk type and the fact that no unblocking is required |

| | Floppy$8$Parameter$Block: |
| | Sectors per track |

| | Floppy$8$Parameter$Block: |
| | Extra byte prefixed to DPB for this version of the BIOS |

| | Floppy$8$Parameter$Block: |
| | Indicates disk type and the fact that no unblocking is required |

Figure 8-10. (Continued)
Chapter 8: Writing an Enhanced BIOS 271

CPI

Number of Logical Disks

RNC, Return if > maximum number of disks

DAD H, 0

Select disk in register C

MOV A,C

Assume an error

Check if requested disk valid

CPU 'H' - 'A'

Check if memory disk

SELECTK$MDisk

Yes

CPU Number of Logical Disks

SELECTK$MDisk

Return if > maximum number of disks

Save selected disk number

Set up to return DPH address

Make disk into word value

Compute offset down disk parameter header table by multiplying by

parameter header length (16 bytes)

H:W

H:W

H:W

H:W

H:W

H:W

H:W

H:W

H:W

H:W

H:W

Get base address

DE -> appropriate DPH

Save DPH address

Access disk parameter block to

extract special prefix byte that

identifies disk type and whether
deblocking is required

Get DPB pointer offset in DPH

Get DPB address in DPH

Get DPB address in DPH

Get prefix byte

Get prefix byte

Figure 8-10. (Continued)
<table>
<thead>
<tr>
<th>Address</th>
<th>comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>098A 32360A</td>
<td>06848 STA Selected$Disk$Type ;Save for use in low level driver</td>
</tr>
<tr>
<td>098D 7E</td>
<td>06849 MOV A,M ;Get another copy of prefix byte</td>
</tr>
<tr>
<td>098E 680</td>
<td>06850 ANI Need$Deblocking ;Isolate deblocking flag</td>
</tr>
<tr>
<td>0990 32350A</td>
<td>06851 STA Selected$Disk$Deblock ;Save for use in low level driver</td>
</tr>
<tr>
<td>0993 E1</td>
<td>06852 POP H ;Recover DPH pointer</td>
</tr>
<tr>
<td>0994 C9</td>
<td>06853 RET</td>
</tr>
<tr>
<td>06854</td>
<td>;</td>
</tr>
<tr>
<td>06855 SELDSK$Disk$</td>
<td></td>
</tr>
<tr>
<td>0995 212309</td>
<td>06856 LXI H, M$Disk$HDPH ;Return correct parameter header</td>
</tr>
<tr>
<td>0998 C38609</td>
<td>06857 JMP SELDSK$Set$Disk$Type ;Resume normal processing</td>
</tr>
<tr>
<td>06858</td>
<td>;</td>
</tr>
<tr>
<td>07000</td>
<td>;</td>
</tr>
<tr>
<td>07001</td>
<td>; Set logical track for next read or write</td>
</tr>
<tr>
<td>07002</td>
<td>;</td>
</tr>
<tr>
<td>07003</td>
<td>;</td>
</tr>
<tr>
<td>07004 SETTRK:</td>
<td></td>
</tr>
<tr>
<td>099B 60</td>
<td>07005 MOV H,B ;Selected track in BC on entry</td>
</tr>
<tr>
<td>099C 69</td>
<td>07006 MOV L,C</td>
</tr>
<tr>
<td>099D 222E0A</td>
<td>07007 SHLD Selected$Track ;Save for low level driver</td>
</tr>
<tr>
<td>09A0 C9</td>
<td>07008 RET</td>
</tr>
<tr>
<td>07009</td>
<td>;</td>
</tr>
<tr>
<td>07100</td>
<td>;</td>
</tr>
<tr>
<td>07101</td>
<td>; Set logical sector for next read or write</td>
</tr>
<tr>
<td>07102</td>
<td>;</td>
</tr>
<tr>
<td>07103</td>
<td>;</td>
</tr>
<tr>
<td>07104</td>
<td>;</td>
</tr>
<tr>
<td>07105 SETSEC:</td>
<td></td>
</tr>
<tr>
<td>09A1 79</td>
<td>07106 MOV A,C ;Logical sector in C on entry</td>
</tr>
<tr>
<td>09A2 32300A</td>
<td>07107 STA Selected$Sector ;Save for low level driver</td>
</tr>
<tr>
<td>09A5 C9</td>
<td>07108 RET</td>
</tr>
<tr>
<td>07109</td>
<td>;</td>
</tr>
<tr>
<td>07200</td>
<td>;</td>
</tr>
<tr>
<td>07201</td>
<td>; Set disk DMA (Input/Output) address for next read or write</td>
</tr>
<tr>
<td>07202</td>
<td>;</td>
</tr>
<tr>
<td>07203</td>
<td>;</td>
</tr>
<tr>
<td>09A6 0000</td>
<td>07204 DMA$Address$ (Input/Output) address for next read or write</td>
</tr>
<tr>
<td>09A8 69</td>
<td>07205 ;DMA address</td>
</tr>
<tr>
<td>09A9 66</td>
<td>07206 SETDMA:</td>
</tr>
<tr>
<td>09AA 22A609</td>
<td>07207 MOV L,C ;Address in BC on entry</td>
</tr>
<tr>
<td>09AD C9</td>
<td>07208 MOV H,B</td>
</tr>
<tr>
<td>09AE 00010203</td>
<td>07209 MOV H,B</td>
</tr>
<tr>
<td>09B0 20212223</td>
<td>07210 SHLD DMA$Address$ ;Save for low level driver</td>
</tr>
<tr>
<td>09B4 0000000F</td>
<td>07211 ;</td>
</tr>
<tr>
<td>09B6 222E0A</td>
<td>07212 ;</td>
</tr>
<tr>
<td>09BE 1C1D1E1F</td>
<td>07213 ;</td>
</tr>
<tr>
<td>09C2 08090A08</td>
<td>07214 ;</td>
</tr>
<tr>
<td>09CC 18191A1B</td>
<td>07215 ;</td>
</tr>
<tr>
<td>09CA 04050607</td>
<td>07216 ;</td>
</tr>
<tr>
<td>09CE 14151617</td>
<td>07217 ;</td>
</tr>
<tr>
<td>09D0 24252627</td>
<td>07218 ;</td>
</tr>
<tr>
<td>09D6 34353637</td>
<td>07219 ;</td>
</tr>
<tr>
<td>09DA 44454647</td>
<td>07220 ;</td>
</tr>
<tr>
<td>09DE 30313233</td>
<td>07221 ;</td>
</tr>
<tr>
<td>09E0 40414243</td>
<td>07222 ;</td>
</tr>
<tr>
<td>09E4 2C2D2E2F</td>
<td>07223 ;</td>
</tr>
<tr>
<td>09E6 3C3D3E3F</td>
<td>07224 ;</td>
</tr>
<tr>
<td>09EE 28292A2B</td>
<td>07225 ;</td>
</tr>
<tr>
<td>09F2 38393A38</td>
<td>07226 ;</td>
</tr>
<tr>
<td>07332 Floppy$Disk$table: &amp; Each physical sector contains four</td>
<td></td>
</tr>
<tr>
<td>07309</td>
<td>128-byte sectors.</td>
</tr>
<tr>
<td>07310</td>
<td>;Physical 128b Logical 128b Physical 512-byte</td>
</tr>
<tr>
<td>09AE 00010203</td>
<td>07311 DB 00,01,02,03</td>
</tr>
<tr>
<td>09B2 10111213</td>
<td>07312 DB 16,17,18,19</td>
</tr>
<tr>
<td>09B6 20212223</td>
<td>07313 DB 32,33,34,35</td>
</tr>
<tr>
<td>09BB 0000000F</td>
<td>07314 DB 12,13,14,15</td>
</tr>
<tr>
<td>09BE 1C1D1E1F</td>
<td>07315 DB 28,29,30,31</td>
</tr>
<tr>
<td>09C2 08090A08</td>
<td>07316 DB 32,33,34,35</td>
</tr>
<tr>
<td>09CC 18191A1B</td>
<td>07317 DB 24,25,26,27</td>
</tr>
<tr>
<td>09CA 04050607</td>
<td>07318 DB 20,21,22,23</td>
</tr>
<tr>
<td>09CE 14151617</td>
<td>07319 DB 04,05,06,07</td>
</tr>
<tr>
<td>09D0 24252627</td>
<td>07320 DB 20,21,22,23</td>
</tr>
</tbody>
</table>

Figure 8-10. (Continued)
Chapter 8: Writing an Enhanced BIOS

Figure 8-10. (Continued)
274 The CP/M Programmer's Handbook

Figure 8-10. (Continued)
Chapter 8: Writing an Enhanced BIOS 275

Figure 8-10. (Continued)
276  The CP/M Programmer's Handbook

Figure 8·10. (Continued)
Chapter 8: Writing an Enhanced BIOS

Figure 8-10. (Continued)
There are two "smart" disk controllers on this system. one for the 8" floppy diskette drives, and one for the 5 1/4" mini-diskette drives.

Once the operation has been completed, the controller resets its disk control byte to OOH, and this indicates completion to the disk driver code.

The controller also sets a return code in a disk status block. If the first byte of this status block is less than 80H, then a disk error has occurred. For this simple BIOS, no further details of the status settings are relevant. Note that the disk controller has built-in retry logic; reads and writes are attempted ten times before the controller returns an error.

The disk control table layout is shown below. Note that the controllers have the capability for control tables to be chained together so that a sequence of disk operations can be initiated. In this BIOS this feature is not used. However, the controller requires that the chain pointers in the disk control tables be pointed back to the main control bytes in order to indicate the end of the chain.

| 0040 | 08338 | DiskControle $ | EQU | 40H | 8" control byte |
| 0041 | 08339 | Command$Block$ | EQU | 41H | Control table pointer |
| 0042 | 08340 |  |  |  |  |
| 0043 | 08341 | DiskStatus$Block | EQU | 43H | 8" AND 5 1/4" status block |
| 0044 | 08342 |  | EQU | 45H | 5 1/4" control byte |
| 0045 | 08343 | Command$Block$ | EQU | 46H | Control table pointer |
| 0046 | 08344 |  |  |  |  |
| 0047 | 08345 | Floppy Disk Control Tables |
| 0048 | 08346 |  |  |  |  |
| 0049 | 08347 | Floppy$Commands | DB | 0 | Command |
| 0050 | 08348 | Floppy$Read$Code | EQU | 01H |  |
| 0051 | 08349 | Floppy$Write$Code | EQU | 02H |  |
| 0052 | 08350 | Floppy$Unit$ | DB | 0 | Unit (drive) number = 0 or 1 |
| 0053 | 08351 | Floppy$Head$ | DB | 0 | Head number = 0 or 1 |
| 0054 | 08352 | Floppy$Track$ | DB | 0 | Track number |
| 0055 | 08353 | Floppy$Sector$ | DB | 0 | Sector number |

Figure 8-10. (Continued)
Figure 8-10. (Continued)
For writing, the 128-byte sector must be processed:

1. Move sector DMA Address → M Disk Buffer
2. Select correct track/bank
3. Move sector M Disk Buffer → M Disk image
4. Select bank 0
5. DMA Address
6. M Disk Buffer
7. M Disk
8. 0
9. M Disk

For reading, the processing is:

1. Select correct track/bank
2. Move sector M Disk image → M Disk Buffer
3. Select Bank 0
4. Move sector M Disk Buffer → DMA Address

If there is any risk of any interrupt causing control to be transferred to an address below 48K, interrupts must be disabled when any bank other than 0 is selected.

Figure 8-10. (Continued)
Chapter 8: Writing an Enhanced BIOS

Figure 8.10. (Continued)
Figure 8-10. (Continued)
Chapter 8: Writing an Enhanced BIOS

Figure 8-10. (Continued)
This routine builds and outputs an error message. The user is then given the opportunity to:

- R -- retry the operation that caused the error
- I -- ignore the error and attempt to continue
- A -- abort the program and return to CP/M.

```assembly
09490 09491 09492 09493 09494 09495 09496 09497 09498
  DiskError:
  ; Preserve error code from controller
  PUSH PSN
  MOV A, H
  CALL CAH
  ; Convert code to hex.
  LDA In$Buffer$Disk
  CALL CAH
  ; Make into letter
  ADI 'A'
  CALL Disk$EM$Drive
  ; Convert head number
  LDA Floppy$Head
  CALL Disk$EM$Head
  ; Convert sector number
  LDA Floppy$Sector
  CALL Disk$EM$Sector
  ; Output first part of message
  LXI H, Disk$EM$Message
  CALL Output$Error$Message
  ; Move to next (or first) entry
  LDA H,$Entry$Size
  DAD D
  CALL Request$User$Choice
  ; Get controller command
  CPI Floppy$Read$Code
  JZ Disk$Error$Read
  CALL Output$Error$Message
  ; Get address
  MOV B, A
  CALL Disk$EM$Read
  ; Choose operation text
  LXI H, Disk$EM$Read
  CALL Output$Error$Message
  ; Display operation type
  CALL Output$Error$Message
  ; Ask the user what to do next
  CALL Request$User$Choice
  ; Get controller command
  CPI Floppy$Read$Code
  JZ Disk$Error$Read
  CALL Output$Error$Message
  ; Get address
  MOV B, A
  CALL Disk$EM$Read
  ; Display operation type
  LDA H, Disk$EM$Read
  CALL Output$Error$Message
  ; Choose operation text
  LXI H, Disk$EM$Read
  CALL Output$Error$Message
  ; Display operation type
  CALL Output$Error$Message
  ; Get controller command
  CPI Floppy$Read$Code
  JZ Disk$Error$Read
  CALL Output$Error$Message
  ; Get address
  MOV B, A
  CALL Disk$EM$Read
  ; Display operation type
  LDA H, Disk$EM$Read
  CALL Output$Error$Message
  ; Choose operation text
  LXI H, Disk$EM$Read
  CALL Output$Error$Message
  ; Display operation type
  CALL Output$Error$Message
  ; Get controller command
  CPI Floppy$Read$Code
  JZ Disk$Error$Read
  CALL Output$Error$Message
  ; Get address
  MOV B, A
  CALL Disk$EM$Read
  ; Display operation type
  LDA H, Disk$EM$Read
  CALL Output$Error$Message
  ; Choose operation text
  LXI H, Disk$EM$Read
  CALL Output$Error$Message
  ; Display operation type
  CALL Output$Error$Message
  ; Get controller command
  CPI Floppy$Read$Code
  JZ Disk$Error$Read
  CALL Output$Error$Message
  ; Get address
  MOV B, A
  CALL Disk$EM$Read
  ; Display operation type
  LDA H, Disk$EM$Read
  CALL Output$Error$Message
  ; Choose operation text
  LXI H, Disk$EM$Read
  CALL Output$Error$Message
  ; Display operation type
  CALL Output$Error$Message
  ; Get controller command
  CPI Floppy$Read$Code
  JZ Disk$Error$Read
  CALL Output$Error$Message
  ; Get address
  MOV B, A
  CALL Disk$EM$Read
  ; Display operation type
  LDA H, Disk$EM$Read
  CALL Output$Error$Message
  ; Choose operation text
  LXI H, Disk$EM$Read
  CALL Output$Error$Message
  ; Display operation type
  CALL Output$Error$Message
  ; Get controller command
  CPI Floppy$Read$Code
  JZ Disk$Error$Read
  CALL Output$Error$Message
  ; Get address
  MOV B, A
  CALL Disk$EM$Read
  ; Display operation type
  LDA H, Disk$EM$Read
```
Chapter 8: Writing an Enhanced BIOS

Figure 8-10. (Continued)
On warm boot, the CCP and BOOS must be reloaded into memory. In this BIOS, only the 5 1/4" diskettes will be used, therefore this code is hardware specific to the controller. Two prefabricated control tables are used.

Figure 8-10. (Continued)
Chapter 8: Writing an Enhanced BIOS

287

OEB3 C3750E 09850 JMP WBOOT ; Restart warm boot
09851
09852 WarmBootErrorMessage:
OEB6 0D0A57617209853 DB CR,LF,'Warm Boot Error - retrying... ',CR,LF,0
09854
09855
10000 :0
10001
10002 GhostInterrupt: ; Control will only arrive here under the most
10003 : unusual circumstances, as the interrupt controller will have been programmed to
10004 ; suppress unused interrupts.
10005
10006
OEB9 F5 10007 PUSH PSM ; Save pre-interrupt registers
OED9 3E20 10008 MVI A,IC#EDI ; Indicate end of interrupt
OEDB D5DB 10009 OUT IC#OCM2#Port
OEDD F1 10010 POP PSM
OEDE C9 10011 RET
10101
10102
10103
Patch CP/M
10104
10105
This routine makes some very special patches to the
10106
CPL and BDOS in order to make some custom enhancements
10107
Public files:
10108
On large hard disk systems it is extremely useful
10109
to partition the disk using the user number features.
10110
However, it becomes wasteful of disk space because
10111
multiple copies of common programs must be stored in
10112
each user area. This patch makes User 0 public --
10113
accessible from any other user area.
10114
*** WARNING ***
10115
Files in User 0 MUST be set to system and read/only
10116
status to avoid their being accidentally damaged.
10117
Because of the side effects associated with public
10118
files, the patch can be turned on or off using
10119
a flag in the long term configuration block.
10120
User prompt:
10121
When using CP/M's USER command and user numbers
10122
in general, it is all too easy to become confused
10123
and forget which user number you are "in." This
10124
patch modifies the CCP to display a prompt which
10125
shows not only the default disk id., but also the
10126
current user number, and an indication of whether
10127
public files are enabled:
10128
P3B> or 3B>
10129
* When public files are enabled.
10130
10131
Equates for public files
10132
D35E = 10133
D37C = 10134
D361 = 10135
0000 = 10136
FF#BDOS$Exit$Point EQU BDOS$Entry + 758H
10137
10138
FF#BDOS$Char$Matches EQU BDOS$Entry + 778H
10139
FF#BDOS$Cursor$Bytes EQU BDOS$Entry + 758H
10140
10141
Equate for user prompt
10142
C788 = 10143
C78B = 10144
UP#CCP$Exit$Point EQU CCP$Entry + 38B8
10145
10146
UP#CCP$Resume$Point EQU CCP$Entry + 38B8
10147
C513 = 10148
UP#CCP$Get$User EQU CCP$Entry + 113H
10149
10150
C5D0 = 10151
UP#CCP$Get$Disk$Id EQU CCP$Entry + 1DCH
10152
10153
C4B8 = 10154
UP#CCP$CONOUT EQU CCP$Entry + 8CH
10155
10156
Set up the intervention points
10157
10158
OEDF 3E93 10159
OEE1 325ED3 10160
Set up opcode
OEOC A, JMP ; Set up opcode

Figure 8-10. (Continued)
Figure 8-10. (Continued)
Chapter 8: Writing an Enhanced BIOS

### Figure 8-10. (Continued)
The CP/M Programmer's Handbook

The short term configuration block.

This contains variables that can be set once CP/M has been initiated, but that are never preserved from one loading of CP/M to the next. This part of the configuration block form the last initialized bytes in the BIOS.

The two values below are used by utility programs that need to read in the long term configuration block from disk.

The BIOS starts on a 256-byte page boundary, and therefore will always be on a 128-byte sector boundary in the reserved area on the disk. A utility program can then, using the utility programs can set, determine how many 128-byte sectors need to be read in by the formula:

\[(\text{LTCB\#Offset + LTCB\#Length}) / 128\]

The LTCB\#Offset is the offset from the start of the BIOS to where the first byte of the long term configuration block starts. Using the offset and the length, the utility can copy the RAM version of the LTCB over the disk image that it has read from the disk, and then write the updated LTCB back onto the disk.

With the exception of the main Disk\#Buffer, which contains a few bytes of code, all of the other uninitialized variables occur here. This has the effect of reducing the number of bytes that need to be stored in the CP/M image on the disk.

Figure 8-10. (Continued)
Chapter 8: Writing an Enhanced BIOS

Figure 8-10. (Continued)
The CP/M Programmer's Handbook

292

Figure 8-10.  (Continued)
Chapter 8: Writing an Enhanced BIOS

End of cold boot initialization code

Disk work areas

Disk allocation vectors

These are used by the DDOS to maintain a bit map of which allocation blocks are used and which are free.

One byte is used for eight allocation blocks, hence the expression of the form "allocation blocks/8)+1.

Figure 8-10. (Continued)
<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
<th>Description</th>
<th>Address</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>2426</td>
<td>11025</td>
<td>Disk#B#Allocation#Vector</td>
<td>DS (174/8)+1</td>
<td>B:\</td>
</tr>
<tr>
<td>243C</td>
<td>11026</td>
<td>;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>245B</td>
<td>11027</td>
<td>Disk#C#Allocation#Vector</td>
<td>DS (242/8)+1</td>
<td>C:\</td>
</tr>
<tr>
<td>245B</td>
<td>11028</td>
<td>;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>247A</td>
<td>11029</td>
<td>Disk#D#Allocation#Vector</td>
<td>DS (242/8)+1</td>
<td>D:\</td>
</tr>
<tr>
<td>2493</td>
<td>11030</td>
<td>;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2493</td>
<td>11031</td>
<td>M#Disk#Allocation#Vector</td>
<td>DS (192/8)+1</td>
<td>M#Disk</td>
</tr>
<tr>
<td>2493</td>
<td>11032</td>
<td>END</td>
<td>END</td>
<td>; of enhanced BIOS listing</td>
</tr>
</tbody>
</table>

**Figure 8-10.** (Continued)
This chapter describes the enhancements you can make to improve CP/M’s somewhat primitive error handling. It covers the general classes of errors that the BIOS may have to handle. It describes some of the underlying philosophical aspects of errors, how to detect them, and how to correct them or otherwise make the best of the situation.

At the end of the chapter are some example error-handling subroutines. Some of these have already been shown in the previous chapter as part of the enhanced BIOS (Figure 8-10); they are repeated here so that you can see them in isolation.

Classes of Errors

Basically, the user perceives only two classes of errors—those that are user-correctable and those that are not. There is a third, almost invisible class of errors—those that are recoverable by the hardware or software without the user’s intervention.
The possible sources for hardware errors vary wildly from one computer system to another, since error detection is heavily dependent on the particular logic in the hardware. The BIOS can detect some hardware-related errors—mainly errors caused when something takes too long to happen, such as when a recalcitrant printer does not react in a specified length of time.

The BDOS has no built-in hardware detection code. It can detect system errors, such as an attempt to write to a disk file that is marked "Read-Only" in the file directory or attempts to access files that are not on the disk. These BDOS-detected errors, however, generally are unrelated to the well-being of the hardware. For example, a disk controller with a hardware problem could easily overwrite a sector of the directory, thereby deleting several files. This error would not show up until the user tried to use one of the now-departed files.

**BIOS Error-Handling Functions**

The error-handling code in the BIOS has to serve the following functions:

- Detection
- Analysis
- Indication
- Correction.

**Error Detection**

Clearly, before any later steps can be taken, an error must be detected. This can be done by the software alone or by the BIOS interacting with error-detecting logic in the hardware. In general, the only errors that the BIOS can detect unassisted are caused when certain operations take longer to complete than expected. Because the writer of the BIOS knows the operating environment of the specific peripherals in the system, the code can predict how long a particular operation should take and can signal an error when this time is exceeded. This would include such problems as printers that fail to react within a specified time period.

The BIOS can work in cooperation with the hardware to determine whether the hardware itself has detected an error. Armed with the hardware’s specifications, the BIOS can input information on controller or device status to trigger error-detecting logic. How this should be done depends heavily on the peripheral devices in your computer system and the degree to which these devices have "smart" controllers capable of processing independently of the computer. Unfortunately, many manufacturers document the significance of individual status bits that indicate errors, but not combinations of errors, or what to do when a particular error occurs.
Error Analysis

Given that your BIOS has detected an error, it must first determine the class of error; that is, whether or not the error can be corrected by simply trying the operation again. Some errors appear at first to be correctable, but retrying the operation several times still fails to complete it. An example would be a check-sum error while reading a disk sector. If several attempts to read the sector all yield an error, then it becomes a “fatal” error. The code in your BIOS must be capable of initial classification and then subsequent reclassification if remedial action fails.

Other types of errors can be classified immediately as fatal errors—nothing can be done to save the situation. For example, if the floppy disk controller indicates that it cannot find a particular sector number on a diskette (due to an error in formatting), there is nothing that the BIOS can do other than inform the user of the problem and supply other helpful information.

Analysis of errors may require some basic research, such as inducing failures in the hardware and observing combinations of error indicators. For example, some printers (interfaced via a parallel port) indicate that they are “Out of Paper” or “Busy” when, in fact, they are switched off. The BIOS should detect this condition and tell the user to switch the printer on, not load more paper.

Error Indication

An incomplete or cryptic error message is infuriating. It is the functional equivalent of saying, “There has been an error. See if you can guess what went wrong!”

An error message, to be complete, should inform the recipient of the following:

- The fact that an error has occurred.
- Whether or not automatic recovery has been attempted and failed.
- The details of the error, if need be in technical terms to assist a hardware engineer.
- What possible choices the user has now.

To put these points into focus, consider the error message that can be output by CP/M after you have attempted to load a program by entering its name into the CCP. What you see on the console is the following dialog:

```
A>myprog <cr>
BAD LOAD
A>
```

All you know is that there has been an error, and you must guess what it is, even though the specific cause of the error was known to CP/M when it output the message. This error message is output by the CCP when it attempts to load a
".COM" file larger than the current transient program area. The message "BAD LOAD" is only understandable after you know what the error is. Even then, it does not tell you what went wrong, whether there is anything you can do about it, and how to go about doing it.

To be complete, this error message could say something like this:

\[ \texttt{A>myprog<cr>} \]

"MYPROG.COM" exceeds the available memory space by 1,024 bytes, and therefore cannot be loaded under the current version of CP/M.

Notice how the message tells you what the problem is, and even quantifies it so that you can determine its severity (you need to get 1 K more memory or reduce the program's size). It also tells you how you stand—you cannot load this program under the current version of CP/M, so retrying the operation is futile.

Not many systems programmers like to output messages like the example above. They argue that such a message is too long and too much work for something that does not happen often. Admittedly, the message is too long. It could be shortened to read

\[ \texttt{(131) Program 1,024 bytes too large to load.} \]

This conveys the same information; the number in parentheses can serve as a reference to a manual where the full impact of the message should be described.

The major problem with the way error messages are designed is that they usually are written by programmers to be read by nontechnical lay users, and programmers are notoriously bad at guessing what nonexperts need to know.

Error indications you design should address the following issues, from the point of view of the user:

- The cause of the error
- The severity of the error
- The corrective action that has and can be taken.

Examine the error messages in the error processor for the example BIOS in Figure 8-10, from line 03600 onward. Although these are an improvement on the BDOS all-purpose

\[ \texttt{BDOS Error on A: Bad Sector} \]

even these messages do not really meet all of the requirements of a good error message system.

Another often overlooked aspect of errors is that most hardware errors form a pattern. This pattern is normally only discernible to the trained eye of a hardware maintenance engineer. When these engineers are called to investigate a problem,
they will quiz the user to determine whether a given failure is an isolated incident or part of an ongoing pattern. This is why an error message should contain additional technical details. For example, a disk error message should include the track and sector used in the operation that resulted in an error. Only with these details can the engineer piece together the context of a failure or group of failures.

Error Correction

Given that a lucid error message has been displayed on the console, the user is still confronted with the question: “Now what do I do?” Not only can this be difficult for the user to answer, but also the particular solution decided upon can be hard for the BIOS to execute.

Normally, there are three possible options in response to errors:

- Try the operation again
- Ignore the error and attempt to continue
- Abort the program causing the error and return to CP/M.

For some errors, retrying can be effective. For example, if you forget to put the printer on-line and get a “Printer Timeout” error message, it is easy to put the printer back on-line and ask the BIOS to try again to send data to the printer.

Seldom can you ignore an error and hope to get sensible results from the machine; many disk controllers do not even transfer data between themselves and the disk drive if an error has been detected. Only ignorant users, or brave ones in desperation, ignore errors.

Aborting the program causing the error is a drastic measure, although it does escape from what could otherwise be a “deadly embrace” situation. For example, if you misassign the printer to an inactive serial port and turn on printer echoing (with the CONTROL-P toggle), you will send the system into an endless series of “Printer Timeout” messages. If you abort the program, the error handler in the BIOS executes a System Reset function (function 0) in the BDOS, CP/M warm boots, and control is returned to the CCP. In the process, the printer toggle is reset and the circle is broken.

Practical Error Handling

This section discusses several errors, describing their causes and the way in which the BIOS and the user can handle them when they occur.

Character I/O Errors

At the BIOS level, most detectable errors related to character input or output will be found by the hardware chips.
Parity Error

Parity, in this context, refers to the number of bits set to 1 in an 8-bit character. The otherwise unused eighth bit in ASCII characters can be set to make this number always odd, or alternatively, always even. Your computer hardware can be programmed to count the number of 1 bits in each character and to generate an error if the number is odd (odd parity) or, alternatively, if it is even (even parity). If the hardware on the other end of the line is programmed to operate in the same mode, parity checking provides a primitive error-detection mechanism — you can tell that a character is bad, but not what it should have been.

CP/M does not provide a standard mechanism for reporting a parity error, so your only option is to reset the hardware and substitute an ASCII DEL (7FH; delete) character in the place of the erroneous character.

If your BIOS is operating in a highly specialized environment, you may need to count the number of such parity errors so that a utility program can report on the overall performance of the system.

Framing Error

When an 8-bit ASCII character is transmitted over a serial line, the eight bits are transmitted serially, one after the other. A start bit is transmitted first, followed by the data character and then a stop bit. If the hardware fails to find the stop and start bits in the correct positions, a framing error will occur. Again, the only option available to the BIOS is to reset the hardware chip and substitute an ASCII DEL.

Overrun Error

This error occurs when incoming data characters arrive faster than the program can handle them, so that the last characters overrun those being processed by the hardware chip. This error can normally be avoided by the use of serial line protocols, such as those in the example BIOS in Figure 8-10.

An overrun error implies that the protocol has broken down. As with the parity and framing errors, almost the only option is to reset the hardware and substitute a DEL character.

Printer Timeout Error

This is one of the few errors where the BIOS can sensibly attempt an error recovery. The error occurs when the BIOS tries to output a character to a serial printer and finds that the printer is not ready for more than, say, 30 seconds. The most common cause of this error is that the user forgets to put the printer on-line. Many printers require that they be off-line during a manual form feed, and users will often forget to push the on-line button afterward.

After a 30-second delay, the BIOS can send a message to the console device(s) informing the user of the error and asking the user to choose the appropriate course of action. Note that console output can be directed to more than one device.
Parallel Printers

Printers connected to your system by means of a parallel port can indicate their status to the computer much more easily than can serial printers. They can communicate such error states as “Out of Paper,” “End of Ribbon,” and “Off-line.” These single-error indicators can also be used in combination to indicate whether the printer cable is connected, or even whether the printer is receiving power. You need to experiment, deliberately putting the printer into these states and reading status in order to identify them. It is misleading to indicate to the inexperienced user that the printer is “Out of Paper” when the problem is that the data cable has inadvertently become disconnected.

However, each of these errors can be dealt with in the same way as the serial printer’s timeout problem: display an error message and request the user’s choice of action.

Example Printer Error Routine

Figure 9-1 shows an example of a program that handles printer errors. It consists of several subroutines, including

- The error detection classification and indication routine
- The error correction routine.

It uses other subroutines that are omitted from the figure to avoid obscuring the logic. These subroutines are listed in full in the example BIOS in Figure 8-10.

| 0000 = | DDS System Reset | EQU 0 | ;BDOS system reset function |
| 0005 = | BDOS | EQU 5 | ;BDOS entry point |
| 0000 00 | Printer Timeout Flag: DB 0 ;This flag is set by the interrupt |
| 0708 = | Printer Delay Count | EQU 1800 ;Given a clock period of 16.666 ms |

Figure 9-1. Serial printer error handling
Figure 9-1. (Continued)
Disk Errors

Disks are much more complicated than character I/O devices. Errors are possible in the electronics and in the disk medium itself. Most of the errors concerned with electronics need only be reported in enough detail to give a maintenance engineer information about the problem. This kind of error is rarely correctable by retrying the operation. In contrast, media errors often can be remedied by retrying the operation or by special error processing software built into the BIOS. This chapter discusses this class of errors.

Media errors occur when the BIOS tries to read a sector from the disk and the hardware detects a check-sum failure in the data. This is known as a cyclic redundancy check (CRC) error. Some disk controllers execute a read-after-write check, so a CRC error can also occur during an attempt to write a sector to the disk.
With floppy diskettes, the disk driver should retry the operation at least ten times before reporting the error to the user. Then, because diskettes are inexpensive and replaceable, the user can choose to discard the diskette and continue with a new one.

With hard disks, the media cannot be exchanged. The only way of dealing with bad sectors is to replace them logically, substituting other sectors in their place.

There are two fundamentally different ways of doing this. Figure 9-2 shows the scheme known as sector sparing—substituting sectors on an outer track for a sector that is bad.

The advantage of this scheme is that it is dynamic. If a sector is found to be bad in a read-after-write check, even after several retries, then the data intended for the failing sector can be written to a spare sector. The failing sector’s number is placed into a spare-sector directory on the disk. Thereafter, the disk drivers will be redirected to the spare sector every time an attempt is made to read or write the bad sector.

The disadvantage of this system is that the read/write heads on the disk must move out to the spare sector and then back to access the next sector. This can be a problem if you attempt to make a high-speed backup on a streaming tape drive (one that writes data to a tape in a single stream rather than in discrete blocks). The delay caused by reading the spare sector interrupts the data flow to the streaming tape drive.

You need a special utility program to manipulate the spare-sector directory, both to substitute for a failing sector manually and to attempt to rewrite a spare sector back onto the bad sector.

![Figure 9-2. Sector sparing](image-url)
Figure 9-3 shows another scheme for dealing with bad sectors. In this method, bad sectors are skipped rather than having sectors substituted for them.

The advantage of sector skipping is that the heads do not have to perform any long seeks. The failing sector is skipped, and the next sector is used in its place. Because of this, sector skipping can give much better performance. Data can be read off the disk fast enough to keep a streaming tape drive "fed" with data.

The disadvantage of sector skipping is that it does not lend itself to dynamic operation. The bad sector table is best built during formatting. Once data has been written to the disk, if a sector goes bad, all subsequent sectors on the disk must be "moved down one" to make space to skip the bad sector. On a large hard disk, this could take several minutes.

**Example Bad Sector Management**

Sector sparing and sector skipping use similar logic. Both require a spare-sector directory on each physical disk, containing the sector numbers of the bad sectors. This directory is read into memory during cold start initialization. Thereafter, all disk read and write operations refer to the memory-resident table to see if they are about to access a bad sector.

For sector sparing, if the sector about to be read or written is found in the spare directory, its position in the directory determines which spare sector should be read.

![Figure 9-3: Sector skipping](image)
In the case of sector skipping, every access to the disk makes the driver check the bad sector directory. The directory is used to tell how many bad sectors exist between the start of the disk and the failing bad sector. This number must be added to the requested track and sector to compensate for all the bad sectors.

The physical low-level drivers need four entry points:

- Read the specified sector without using bad sector management. This is used to read in the spare directory itself.
- Write the specified sector without using bad sector management. This is used to write the spare directory onto the disk, both to initialize it and to update it.
- Read and write the sector using bad sector management. These entry points are used for normal disk input/output.

Figure 9-4 shows the code necessary for both sector sparing and (using conditional code) sector skipping.

```
; This example shows the modifications to be made in order
; to implement bad sector management using sector sparing
; and sector skipping.
;
0000 = False EQU 0
FFFF = True EQU Not False
0000 = Sector$Sparing EQU False
FFFF = Sector$Skipping EQU Not Sector$Sparing

; Additional equates and definitions
Spare$Directories: Table of spare directory addresses
; Note: The directories themselves are declared at the end of the
; BIOS
0000 D500 DW Spare$Directory$0 ;Physical disk 0
0002 9701 DW Spare$Directory$1 ;Physical disk 1

Spare$Dir$In$Memory: Flags used to indicate whether spare

0004 00 DB 0 ;directory for a given physical disk
0005 00 DB 0 ;has been loaded into memory. Set by SELDSK

0000 = Spare$Track EQU 0 ;Track containing spare directory
0004 = Spare$Sector EQU 4 ;Sector containing directory
0005 = First$Spare$Sector EQU Spare$Sector + 1

; Variables set by SELDSK
; Selected$Spare$Directory:
0006 0000 DW 0 ;Pointer to directory
0008 00 Selected$Disk$ DB 0 ;Logical disk number
0009 00 Disk$Type: DB 0 ;Floppy/hard disks
0004 00 Deblocking$Required: DB 0 ;Deblocking flag
0008 00 Selected$Physical$Disk: DB 0 ;Physical disk number
000C 0000 Disk$Track: DW 0 ;These variables are part of the command
000E 00 Disk$Sector: DB 0 ;block handed over to the disk controller
```

Figure 9-4. Bad sector management
Chapter 9: Dealing with Hardware Errors

307

Figure 9-4. (Continued)
Figure 9·4. (Continued)
Chapter 9: Dealing with Hardware Errors

Figure 9-4. (Continued)
LXI H,Spare$Track
SHLD Disk$Track
MVI A,First$Spare$Sector
ADD B
STA Disk$Sector
ENDIF
IF Sector$Skipping
;Use the following code for sector skipping
;The object is to find the entry in the table which is greater or equal to the requested sector/track

0095 CA9E00 JZ Tracks$Match ;Possible match of track and sector  
0098 D2AC00 JNC Compute$Increment ;Requested track < table entry  
009B C3B900 JMP Check$Next$Entry ;Requested track > table entry

Tracks$Match:

009E 23 INX H ;HL -> MS byte of track  
009F 23 INX H ;HL -> sector  
00A0 77 MOV M,A ;Get sector from table  
00A1 B9 CMP C ;Compare with requested sector  
00A2 CAAB00 JZ Sectors$Match ;Track/sector matches  
00A5 D2AC00 JNC Compute$Increment ;Req. trk/sec < spare trk/sec  
00A8 C3B900 JMP Check$Next$Entry2 ;Move to next table entry

Sectors$Match:

00AB 04 INR B ;If track and sectors match with a table entry, then an additional sector must be skipped

Compute$Increment:

00AC 79 MDV A,C ;B contains number of cumulative number of sectors to skip  
00AD 80 ADD B ;Skip required number  
00AE 0612 HVI B,Sectors$Per$Track ;Determine final sector number and track increment  
00B0 CDC300 CALL DIV$A$BY$B ;Returns C = quotient, A = remainder  
00B3 320E00 STA Disk$Sector ;A = new sector number  
00B6 59 MDV E,C ;Make track increment a word  
00B7 1600 MVI D,0  
00B9 2A0C00 LHLD Disk$Track ;Get requested track  
00BC 19 DAD D ;Add on increment  
00BD 220C00 SHLD Disk$Track ;Save updated track  
ENDIF

Not$Bad$Sector:

;Either track/sector were not bad, or requested track and sector have been updated. Go to physical disk read/write

00C0 C3D500 JMP Read$Write$Disk  
;IF Sector$Skipping
;Subroutine required for skipping routine

;DIV$A$BY$B
;Divide A by B

;This routine divides A by B, returning the quotient in C and the remainder in A.

;Entry parameters
;A = dividend
;B = divisor

;Exit parameters

Figure 9-4. (Continued)
Chapter 9: Dealing with Hardware Errors

A remainder
C quotient

DIV$ASBV$BSLoop;

CMPM

This subroutine compares the contents of DE to (HL) and (HL+1)
returning with the flags as though the subtraction (HL) - DE
were performed.

Entry parameters
HL -> word in memory
DE = value to be compared

Exit parameters
Flags set for (HL) - DE

CMPM
MOV A,H ;Get MS byte
CMP D ;Return now if MS bytes unequal
INX H ;HL -> LS byte
MOV A,H ;Get LS byte
CMP E ;Return with HL unchanged
DCX H
RET

Absolute$Read:

| MOV A,H | ;Get MS byte |
| CMP D | ;Return now if MS bytes unequal |
| INX H | ;HL -> LS byte |
| MOV A,H | ;Get LS byte |
| CMP E | ;Return with HL unchanged |

Read$Write$Disk:

<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>The remainder of the low level disk drivers follow,</td>
</tr>
<tr>
<td>reading the required sector and track.</td>
</tr>
<tr>
<td>---------------------------------------------------------------</td>
</tr>
</tbody>
</table>

Spare directory declarations

Note: The disk format utility creates an initial spare
directory with track/sector entries for those track/sectors
that it finds are bad. It fills the remainder of the
directory with OFFH's (these serve to terminate the
searching of the directory).
The CP/M Programmer's Handbook

312

Bad sector management (continued)

Improving Error Messages

The final extension to BIOS error handling discussed here is in disk-driver error-message handling. The subroutine shown in the example BIOS in Figure 8-10, although a significant improvement on the messages normally output by the BDOS, did not advise the user of the most suitable course of action for each error. Figure 9-5 shows an improved version of the error message processor.

Figure 9-5. User-friendly disk-error processor
Chapter 9: Dealing with Hardware Errors 313

<table>
<thead>
<tr>
<th>Address</th>
<th>Assembly Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0005 00</td>
<td>DB 0</td>
<td>DB 0</td>
</tr>
<tr>
<td>0006 00</td>
<td>DB 0</td>
<td>DB 0</td>
</tr>
<tr>
<td>0007 00</td>
<td>DB 0</td>
<td>DB 0</td>
</tr>
<tr>
<td>0008 C31500</td>
<td>JMP</td>
<td>Wait For Disk Complete</td>
</tr>
<tr>
<td>0008 3E02</td>
<td>MI A.FloppyWriteCode</td>
<td>Set write function code</td>
</tr>
<tr>
<td>000C C31200</td>
<td>JMP</td>
<td>CommonPhysical : Go to common code</td>
</tr>
<tr>
<td>0010 3E01</td>
<td>MI A.FloppyReadCode</td>
<td>Set read function code</td>
</tr>
<tr>
<td>0012 320100</td>
<td>STA</td>
<td>FloppyCommand : Set command table</td>
</tr>
<tr>
<td>0015 AF</td>
<td>XRA</td>
<td>A</td>
</tr>
<tr>
<td>0016 320000</td>
<td>STA</td>
<td>DiskHungFlag</td>
</tr>
<tr>
<td>0019 213100</td>
<td>LIX</td>
<td>H.DiskHungOut</td>
</tr>
<tr>
<td>001C 013902</td>
<td>LIX</td>
<td>B.DiskTimer</td>
</tr>
<tr>
<td>001F C32803</td>
<td>CALL</td>
<td>SetWatchdog</td>
</tr>
<tr>
<td>0022 7E</td>
<td>MOV</td>
<td>A,M</td>
</tr>
<tr>
<td>0023 B7</td>
<td>ORA</td>
<td>A</td>
</tr>
<tr>
<td>0024 CA3700</td>
<td>JZ</td>
<td>DiskComplete</td>
</tr>
<tr>
<td>0027 3A0000</td>
<td>LDA</td>
<td>DiskHungFlag</td>
</tr>
<tr>
<td>002A B7</td>
<td>ORA</td>
<td>A</td>
</tr>
<tr>
<td>002B C29F02</td>
<td>JNZ</td>
<td>DiskError</td>
</tr>
<tr>
<td>002E C32200</td>
<td>JMP</td>
<td>DiskWaitLoop</td>
</tr>
</tbody>
</table>

**Figure 9-5.** (Continued)
Disk$Complete:

0037 010000       LXI  B,0        ;Reset watchdog timer  
003A CD3803       CALL  Set$Watchdog; HL is irrelevant here
003D 3A4300       LDA  Disk$Status$Block; Complete -- now check status
0040 FE80         CPI  80H         ;Check if any errors occurred
0042 DA9F02       JC   Disk$Error  ;Yes

; Disk$Error$Ignore:
0045 AF            XRA  A            ;No
0046 302600       STA  Disk$Error$Flag; Clear error flag
0049 C9            RET

; Disk error message handling

; Disk$Error$Messages:   ;This table is scanned, comparing the
; disk error status with those in the
; table. Given a match, or even when
; the end of the table is reached, the
; address following the status value
; points to the correct advisory message text.
; Following this is the address of an
; error description message.

004A 40            DB   40H
0048 B0019000       DW   Disk$Advice1, Disk$Msg$40
004F 41            DB   41H
0050 C0919000       DW   Disk$Advice2, Disk$Msg$41
0054 42            DB   42H
0055 E301A400       DW   Disk$Advice3, Disk$Msg$42
0059 21            DB   21H
005A 0702B400       DW   Disk$Advice4, Disk$Msg$21
005E 22            DB   22H
005F 1B02B900       DW   Disk$Advice5, Disk$Msg$22
0063 23            DB   23H
0064 1B02C000       DW   Disk$Advice5, Disk$Msg$23
0069 24            DB   24H
006A 3D02D200       DW   Disk$Advice6, Disk$Msg$24
006D 25            DB   25H
006E 3D02DE00       DW   Disk$Advice6, Disk$Msg$25
0072 11            DB   11H
0073 5302F100       DW   Disk$Advice7, Disk$Msg$11
0077 12            DB   12H
0078 5302FF00       DW   Disk$Advice7, Disk$Msg$12
007C 13            DB   13H
007D 5302C001       DW   Disk$Advice7, Disk$Msg$13
0081 14            DB   14H
0082 53021A01       DW   Disk$Advice7, Disk$Msg$14
0086 15            DB   15H
0087 53022901       DW   Disk$Advice7, Disk$Msg$15
008B 16            DB   16H
008C 53023501       DW   Disk$Advice7, Disk$Msg$16
0090 00            DB   0            ;== Terminator
0091 53024501       DW   Disk$Advice7, Disk$Msg$Unknown ;Unmatched code

0005 = 5            ;Entry$Size EQU 5 ;Entry size in error message table

; Message texts
0095 48756E700700Disk$Msg$40:  DB   'Hung',0 ;Timeout message
009A 466F74205200Disk$Msg$41:  DB   'Not Ready',0
009F 5772697465Disk$Msg$42:  DB   'Write Protected',0
00A4 4461746100Disk$Msg$43:  DB   'Data',0
00A9 466F7266610Disk$Msg$44:  DB   'Format',0
00AF 4277657265Disk$Msg$45:  DB   'Missing Data Mark',0
00B4 4275736564Disk$Msg$46:  DB   'Bus Timeout',0
00B9 436F6E7472Disk$Msg$47:  DB   'Controller Timeout',0
00C4 4472656D52Disk$Msg$48:  DB   'Drive Address',0
00C9 48656674650Disk$Msg$49:  DB   'Head Address',0
00D4 5747261636Disk$Msg$50:  DB   'Track Address',0
0100 E3F03000       DB   'Unknown',0

Figure 9-5.  (Continued)
Chapter 9: Dealing with Hardware Errors

Figure 9-5. (Continued)
Figure 9-5. (Continued)
Chapter 9: Dealing with Hardware Errors

0336 0E00 MVI C,0
0338 CD0500 CALL BDOS

; This is a radical approach, but
; it does cause CP/M to restart
; System reset

; Omitted subroutines (listed in full in Figure 8-10)

; Set&Watchdog: ; Set watchdog timer (to number of "ticks" in BC, and
; to transfer control to (HL) if timer hits zero).
; Convert A to two ASCII hex characters, storing
; the output in (HL) and (HL+1).
; Output&Error&Message: ; Display the 00-byte terminated error message
; pointed to by HL. Output is directed only to
; those console devices not being used for list
; output as well.
; Request&User&Choice: ; Display prompt "Enter R, A, I..." and return
; ; single keyboard character (uppercase) in A
; ; Dummy

Figure 9-5. User-friendly disk-error processor (continued)
build a machine, take it to the top of a hill, throw it off, and, when it crashes, examine the debris to discover what went wrong.

Each time you do an assembly and test, you are building the aircraft and lobbing it off the edge of a cliff. Each time it crashes, you examine the wreckage and try to determine the possible cause.

This is a highly inferential process. With the wreckage as a starting point, you use inference and intuition to extrapolate the real problem and the correction for it.

**Built-In Debug Code**

The single most important concept that you will need in testing CP/M systems is the same as that used in the modern day "black box" flight recorder. This device is essentially a multi-channel tape recorder that records all of the relevant conditions of the aircraft, its height, altitude, throttle settings, flap settings, and even the voice communications among crew members. If the airplane crashes, investigators can replay the information and understand what happened during the flight.

Applying this concept to debugging CP/M means that you must build into your code some method for recording what it is doing, so that if the system crashes, you can see what it was doing. Make the code tell you what went wrong.

The debug code should be designed at the same time as the rest of the program. Plan the debugging code while the design is still on the drawing board. The source code for debugging should be a permanent part of the BIOS. Use conditional assembly to "IF" out most of the debug code from the final version, or make the code sensitive to a flag in the configuration block so that you can re-enable the debug code at a moment’s notice if the system begins to behave strangely.

The more meaningful the debug output data, the less you will have to guess at what is wrong, and therefore the less painful and time-consuming the debugging process will be. Make the output intelligible to others who may use it or yourself several months hence. Data that tells you what is happening is more useful than internal hexadecimal values, particularly if someone else must interpret it or relay it to you over the telephone.

**Debug Subroutines**

Many programmers do their debugging on a casual “catch as catch can” basis because they are overwhelmed by the task of building the necessary tools. Others are too eager to start on a new program to take a few extra hours or days to build debug subroutines.

To help solve this problem, the following section provides some ready-made debugging tools that can be used “as is.” Each of these routines has been thor-
Oughtly debugged (there's nothing worse than debug code with bugs in it!) and has been used in actual program testing.

**Overall Design Philosophy**

Some common methods run through the examples that follow. These include displaying meaningful “captions” (including the specific address that called the debug routine), grouping all debugging code together, preserving the contents of all registers, and setting up the stack area in a standard way.

**Debug Code Captions**

When the contents of registers or memory are output as part of a debugging process, a caption of explanatory text describing the values should be displayed. For example, rather than displaying the contents of the A register like this,

\[ A = 1F \]

you can use a meaningful caption such as:

Transaction Code A = 1F.

When you write additional debugging code, especially if you need to add it to an existing routine, it is cumbersome to have to write the call to the debug routine and then search through the source code to find a convenient place to put an ASCII caption string. A caption string several pages removed from the point where it is referenced makes for problems when you want to relate the debug output on the screen or listing to the source code itself. Therefore, all of the routines that follow allow you to declare the caption strings “in-line” like this:

```plaintext
IF DEBUG CALL Debug$Routine DB 'Caption string here',CR,LF,O ENDIF

MVI ...... ;Next instruction
```

All of the following routines that output a caption recognize one specific 8-bit value in the caption string. If they encounter a value of 0ADH (mnemonic for ADdress), they will output the address of the byte following the call to the debug routine. For example,

```plaintext
0210 CALL Debug$Routine
0213 DB 0ADH,'Caption string',0
```

will cause the routine to display the following:

```
0213 Caption string
```

This identifies the point in your program from which the debug routine was called, and thus avoids any possible ambiguity between different calls to the same debug routine with similar captions.
Chapter 10: Debugging a New CP/M System

Grouping Debug Code  Grouping all the debug code together lends itself to using conditional assembly with IF/ENDIF statements.

Setting Up the Stack Area  All of the following routines preserve the CPU registers so that there are no side effects from using them. All of them assume that they can use the stack pointer and that there is sufficient room in the stack area. Hence you will need to declare adequate stack space for your main code and for the debug routines. Fill the stack area with a known pattern like this:

```
DW 9999H, 9999H, 9999H, 9999H, 9999H, 9999H
DW 9999H, 9999H, 9999H, 9999H, 9999H, 9999H
DW 9999H, 9999H, 9999H, 9999H, 9999H, 9999H
```

`Stack Area: ;Label the upper end of the area`

Then, during debugging, you can examine the stack area and determine how much of it is unused. For example, if you looked at the stack area you might see something like this:

```
01 29 00 00 1A 28 10 FF FF 39 02 ED 11 01 37 44
DD 00 00 11 1A 23 31 00 41 AE FE 00 01 10 70 C9
```

Stack area overflow can give arcane bugs; the program seems to leap off into space in a nondeterministic way. By setting up the stack area in this way, you can recognize an overflow condition easily.

Debug Initialization  Before you can execute any of the debug subroutines in this chapter, you must make a call to the initialization subroutine, OB$Init. The OB$Init routine sets up some of the internal variables needed by the debug package. You may need to add some of your own initialization code here.

Console Output

Normally, you can use the CONOUT functions either via the BDOS (Function 2), or via the BIOS by calling the jump vector directly. You cannot do this when you need to debug console routines themselves, nor when you need to debug interrupt service routines. In the latter case, if an interrupt pulled control out of the CONOUT routine in the BIOS, you would get unwanted re-entrancy if the debug code again entered the CONOUT driver to display a caption. Therefore, the debug routines have been written to call their own local CONOUT routine, which is called DB$CONOUT. DB$CONOUT can be changed to call the BDOS, the BIOS, or a "private" polled output routine.

A counterpart DB$CONIN routine for console input is provided for essentially the same reasons.
Controlling Debug Output

All output of debug routines in this chapter is controlled by a single master flag, DB$Flag. If this flag is nonzero, debug output will occur; if zero, all output is suppressed.

This flag can be set and cleared from any part of the program you are testing. It is especially useful when you need to debug a subroutine that is called many times from many different places. You can write additional code to enable debug output when certain conditions prevail; for example, when a particular track or sector is about to be written or when a character input buffer is almost full.

Two subroutines, DB$On and DB$Off, are shown that access the debug control flag. These, as their names suggest, turn debug output on and off.

Turning the debug output on and off from within the program can create a confusing display of debug output, lacking any apparent continuity. DB$Off gives you the option of outputting a character string indicating that debug output has been turned off.

Pass Counters

Another method of controlling debug output is to use a pass counter, enabling debug output only after control has passed through a particular point in the code a specific number of times.

Two subroutines are provided for this purpose. DB$SetPass sets the pass counter to a specific value. DB$Pass decrements this pass count each time control is transferred to it. When the pass count hits zero, the debug control flag DB$Flag is nonzero and debug output begins.

Using pass counter techniques can save you time and effort in tracking down a problem that occurs only after the code has been running for several minutes.

Displaying Contents of Registers and Memory

Figure 10-2 shows a series of display subroutines, the primary one of which is DB$Display. It takes several parameters, depending on the information you want displayed. The generic call to DB$Display is as follows:

```
CALL DB$Display
DB Code <-- Indicates the data to be displayed
DW Optional additional parameters
DB 'Caption string',0
```

The codes that can be used in this call are shown in Table 10-1.

The only function that uses additional parameters is DB$Memory. This displays bytes from memory in hexadecimal and ASCII, using the start and finish
addresses following the call. Here is an example:

```
CALL          DB$Display
DB            DB$Memory
DW            Start$Address,End$Address
DB            'Caption string',0
```

**Table 10-1. Codes for DB$Display**

<table>
<thead>
<tr>
<th>Code</th>
<th>Value displayed</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><strong>8-bit registers</strong></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>DB$F</td>
<td>Condition Flags</td>
</tr>
<tr>
<td>DB$A</td>
<td>Register A</td>
</tr>
<tr>
<td>DB$B</td>
<td>Register B</td>
</tr>
<tr>
<td>DB$C</td>
<td>Register C</td>
</tr>
<tr>
<td>DB$D</td>
<td>Register D</td>
</tr>
<tr>
<td>DB$E</td>
<td>Register E</td>
</tr>
<tr>
<td>DB$H</td>
<td>Register H</td>
</tr>
<tr>
<td>DB$L</td>
<td>Register L</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Memory</strong></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>DB$Memory</td>
<td>Bytes starting and ending at the addresses</td>
</tr>
<tr>
<td></td>
<td>specified by the two word values following</td>
</tr>
<tr>
<td></td>
<td>the code value.</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>16-bit registers</strong></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>DB$BC</td>
<td>Register pair BC</td>
</tr>
<tr>
<td>DB$DE</td>
<td>Register pair DE</td>
</tr>
<tr>
<td>DB$HL</td>
<td>Register pair HL</td>
</tr>
<tr>
<td>DB$SP</td>
<td>Stack Pointer</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Byte values</strong></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>DB$B$BC</td>
<td>Byte addressed by BC</td>
</tr>
<tr>
<td>DB$B$DE</td>
<td>Byte addressed by DE</td>
</tr>
<tr>
<td>DB$B$HL</td>
<td>Byte addressed by HL</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Word values</strong></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>DB$W$BC</td>
<td>Word addressed by BC</td>
</tr>
<tr>
<td>DB$W$DE</td>
<td>Word addressed by DE</td>
</tr>
<tr>
<td>DB$W$HL</td>
<td>Word addressed by HL</td>
</tr>
</tbody>
</table>
Debugging Program Logic

In addition to displaying the contents of registers and memory, you need to display the program’s execution path, not in terms of addresses, but in terms of the problem. You can do this by displaying debug messages that indicate what decisions have been made by the program as it executes. For example, if your BIOS checks a particular value to see whether the system should read or write on a particular device, the debug routine should display a message like this:

**Entering Disk Read Routine**

This is more meaningful than just displaying the function code for the drivers — although you may want to display this as well, in case it has been set to some strange value.

Two subroutines are provided to display debug messages. They are DB$MSG and DB$MSGI. Both of these display text strings are terminated with a byte of OOH. You can see the difference between the two subroutines if you examine the way they are called.

DB$MSG is called like this:

```
LXI H,Message$Text
CALL DB$MSG
```

DB$MSGI is called like this:

```
CALL DB$MSG
DB ODH,OAH,'Message Text'
```

DB$MSGI is more convenient to use. If you decide that you need to add a message, you can declare the message immediately following the call. This also helps when you look at the listing, since you can see the complete text at a glance.

Use DB$MSG when the text of the message needs to be selected from a table. Get the address of the text into HL and then call DB$MSG to display it.

Creating Your Own Debug Displays

If you need to build your own special debug display routines, you may find it helpful to incorporate some of the small subroutines in the debug package. The following are the subroutines you may want to use:

**DB$CONOUT**
Displays the character in the C register.

**DB$CONIN**
Returns the next keyboard character in A.

**DB$CONINU**
Returns the next keyboard character in A, converting lowercase letters to uppercase.
DB$DHLH
Displays contents of HL in hexadecimal.

DB$DAH
Displays contents of A in hexadecimal.

DB$CAH
Converts contents of A to hexadecimal and stores in memory pointed at by HL.

DB$Nibble$To$Hex
Converts the least significant four bits of A into an ASCII hexadecimal character in A.

DB$CRLF
Displays a CARRIAGE RETURN/LINE FEED.

DB$Colon
Displays the string “: “.

DB$Blank
Displays a single space character.

DB$Flag$Save$On
Saves the current state of the debug output control flag and then sets the flag “on” to enable debug output.

DB$Flag$Restore
Restores the debug output control flag to the state it was in when the DB$Flag$Save$On routine was last called.

DB$GHV
Gets a hexadecimal value from the keyboard, displaying a prompt message first. From one to four characters can be specified as the maximum number of characters to be input.

DB$A$To$Upper
If the A register contains a lowercase letter, this converts it to an uppercase letter.

Debugging I/O Drivers

Debugging low-level device drivers creates special problems. The major one is that you do not normally want to read and write via actual hardware ports while you are debugging the code—either because doing so would cause strange things to happen to the hardware during the debugging, or because you are developing and debugging the drivers on a system different from the target hardware on which the drivers are to execute.

Before considering the solution, remember that the input and output instructions (IN and OUT) are each two bytes long. The first byte is the operation code
(0DBH for input, 0D3H for output), and the second byte is the port number to "input from" or "output to."

Debug subroutines are provided here to intercept all IN and OUT instructions, displaying the port number and either accepting a hexadecimal value from the console and putting it into the A register (in the case of IN), or displaying the contents of the A register (for the OUT instruction).

IN and OUT instructions can be "trapped" by changing the operation code to one of two RST (restart) instructions. An RST is effectively a single-byte CALL instruction, calling down to a predetermined address in low memory. The debug routines arrange for JMP instructions in low memory to receive control when the correct RST is executed. The code that receives control can pick up the port number, display it, and then accept a hex value for the A register (for IN) or display the current contents of the A register (for OUT). The example subroutines shown later in this chapter use RST 4 in place of IN instructions, RST 5 for OUT.

Wherever you plan to use IN, use the following code:

```
IF RST Debug
  4
ENDIF
IF NOT Debug
  IN
ENDIF
DB Port#Number
```

Note that you can use the IN operation code as the operand of a DB statement. The assembler substitutes the correct operation code.

Use the following code wherever you need to use an OUT instruction:

```
IF RST Debug
  5
ENDIF
IF NOT Debug
  OUT
ENDIF
DB Port#Number
```

When the RST 4 (IN) instruction is executed, the debug subroutine displays

```
1AB3: Input from Port 01 : _
```

The "1AB3" is the address in memory of the byte containing the port number. It serves to pinpoint the IN instruction in memory. You can then enter one or two hexadecimal digits. These will be converted and put into the A register before control returns to the main program at the instruction following the byte containing the port number.

When the RST 5 (OUT) instruction is encountered, the debug subroutine displays

```
1AB5: Output to Port 01 : FF
```

This identifies where the OUT instruction would normally be as well as the port number and the contents of the A register when the RST 5 (OUT) is executed.
Chapter 10: Debugging a New CP/M System

Debugging Interrupt Service Routines

You can use a technique similar to that of the RST instruction just described to "fake" an interrupt. You preset the low-memory address for the RST instruction you have chosen for the jump into the interrupt service routine under test.

When the RST instruction is executed, control will be transferred into the interrupt service routine just as though an interrupt had occurred. You will need to intercept any IN or OUT instructions as described above—otherwise the code probably will go into an endless loop.

Before executing the RST instruction to fake the interrupt, load all the registers with known values. For example:

```
MVI A,0AAH
LXI B,0BBCCH
LXI D,0DDEEH
LXI H,01122H
RST 6 ;Fake interrupt
NOP
```

When control returns from the service routine, you can check to see that it restored all of the registers to their correct values. An interrupt service routine that does not restore all the registers can produce bugs that are very hard to find.

Check, too, that the stack pointer register has been restored and that the service routine did not require too many bytes on the stack.

You also can use the CALL instruction to transfer control to the interrupt service routine in order to fake an interrupt. RST and CALL achieve the same effect, but RST is closer to what happens when a real interrupt occurs. As it is a single-byte instruction, it also is easier to patch in.

Subroutine Listings

Figure 10-1 is a functional index to the source code listing for the debug subroutines shown in Figure 10-2. The listing's commentary defines precisely how each debug subroutine is called.

Figure 10-3 shows the output from the debug testbed.

Software Tools for Debugging

In addition to building in debugging subroutines, you will need one of the following proprietary debug programs:

DDT (Dynamic Debugging Tool)

This program, included with the standard CP/M release, allows you to load programs, set and display memory and registers, trace through your program instruction by instruction, or execute it at full speed, but stopping
at certain addresses (called breakpoints). It also has a built-in mini-assembler and disassembler so you do not have to hand assemble any temporary code "patches" you add.

**SID (Symbolic Interactive Debug)**

Similar to DDT in many ways, SID has enhancements that are helpful if you use Digital Research’s MAC (Macro Assembler) or RMAC (Relocating Macro Assembler). Both of these assemblers can be told to output a file

---

<table>
<thead>
<tr>
<th>Start Line</th>
<th>Functional Component or Routines</th>
</tr>
</thead>
<tbody>
<tr>
<td>00001</td>
<td>Debug subroutine's Testbed</td>
</tr>
<tr>
<td>00100</td>
<td>Test register display</td>
</tr>
<tr>
<td>00200</td>
<td>Test memory dump display</td>
</tr>
<tr>
<td>00300</td>
<td>Test register pair display</td>
</tr>
<tr>
<td>00400</td>
<td>Test byte indirect display</td>
</tr>
<tr>
<td>00500</td>
<td>Test DB$On/Off</td>
</tr>
<tr>
<td>00600</td>
<td>Test DB$Set$Pass and DB$Pass</td>
</tr>
<tr>
<td>00700</td>
<td>Test debug input/output</td>
</tr>
<tr>
<td>00800</td>
<td>Debug subroutines themselves</td>
</tr>
<tr>
<td>01100</td>
<td>DB$Init - initialization</td>
</tr>
<tr>
<td>01200</td>
<td>DB$CONINU - get uppercase keyboard character</td>
</tr>
<tr>
<td>01300</td>
<td>DB$CONIN - get keyboard character</td>
</tr>
<tr>
<td>01400</td>
<td>DB$CONOUT - display character in C</td>
</tr>
<tr>
<td>01500</td>
<td>DB$On - enable debug output</td>
</tr>
<tr>
<td>01600</td>
<td>DB$Off - disable debug output</td>
</tr>
<tr>
<td>01700</td>
<td>DB$Set$Pass - set pass counter</td>
</tr>
<tr>
<td>01800</td>
<td>DB$Pass - execute pass point</td>
</tr>
<tr>
<td>01900</td>
<td>DB$Display - main debug display routine</td>
</tr>
<tr>
<td>02200</td>
<td>Main display processing subroutines</td>
</tr>
<tr>
<td>02500</td>
<td>DB$Display$CALLA - display CALL's address</td>
</tr>
<tr>
<td>02600</td>
<td>DB$DHLH - display HL in hexadecimal</td>
</tr>
<tr>
<td>02700</td>
<td>DB$DAH - display A in hexadecimal</td>
</tr>
<tr>
<td>02800</td>
<td>DB$CAH - convert A to hexademical in memory</td>
</tr>
<tr>
<td>02900</td>
<td>DB$Nibble$To$Hex - convert LS 4 bits of A to hex.</td>
</tr>
<tr>
<td>02930</td>
<td>DB$CRLF - display Carriage Return, Line Feed</td>
</tr>
<tr>
<td>02938</td>
<td>DB$Colon - display &quot; : &quot;</td>
</tr>
<tr>
<td>02946</td>
<td>DB$Blank - display &quot; &quot;</td>
</tr>
<tr>
<td>03100</td>
<td>DB$MSGI - display in-line message</td>
</tr>
<tr>
<td>03147</td>
<td>DB$MSG - display message addressed by HL</td>
</tr>
<tr>
<td>03300</td>
<td>DB$Input - debug INput routine</td>
</tr>
<tr>
<td>03500</td>
<td>DB$Output - debug OUTput routine</td>
</tr>
<tr>
<td>03700</td>
<td>DB$Flag$Save$On - save debug flag and enable</td>
</tr>
<tr>
<td>03800</td>
<td>DB$Flag$Restore - restore debug control flag</td>
</tr>
<tr>
<td>03900</td>
<td>DB$GHV - get hexadecimal value from keyboard</td>
</tr>
<tr>
<td>04100</td>
<td>DB$A$To$Upper - convert A to upper case</td>
</tr>
</tbody>
</table>

Figure 10-1. Functional index for Figure 10-2
Chapter 10: Debugging a New CP/M System

Debug Subroutines

NOTE:
The line numbers at the extreme left are included purely to reference the code from the text.
Because of the need to test these routines thoroughly, and in case you wish to make any changes, the tested 
routine for the debug package itself has been left in this figure.

Debug tested

ORG 100H

START:
LXI SP, TestStack
; Set up local stack
CALL DBInit
; Initialize the debug package
CALL DBOn
; Enable debug output

NOTE:
The line numbers at the extreme left are included purely to reference the code from the text.

There are deliberately induced discontinuities in the numbers in order to allow space for expansion.

Figure 10.2. Debug subroutines
<table>
<thead>
<tr>
<th>Address</th>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>00213</td>
<td>01B7</td>
<td>CALL DBDisplay</td>
</tr>
<tr>
<td>00214</td>
<td>01BA</td>
<td>DB DBMEM</td>
</tr>
<tr>
<td>00215</td>
<td>01BB</td>
<td>DW 101H,100H</td>
</tr>
<tr>
<td>00216</td>
<td>01BF</td>
<td>4D656D6F72 DB 'Memory Dump $3',0 ; start &gt; end address</td>
</tr>
<tr>
<td>00217</td>
<td></td>
<td>; Test register pair display</td>
</tr>
<tr>
<td>00218</td>
<td>01CE</td>
<td>CALL DBDisplay</td>
</tr>
<tr>
<td>00219</td>
<td>01D1</td>
<td>DB DBMEM</td>
</tr>
<tr>
<td>00220</td>
<td>01D2</td>
<td>00010001 DW 100H,100H ; Check end-case of single byte output</td>
</tr>
<tr>
<td>00221</td>
<td>01D6</td>
<td>4D656D6F72 DB 'Memory Dump $4',0</td>
</tr>
<tr>
<td>00300</td>
<td>;#</td>
<td></td>
</tr>
<tr>
<td>00301</td>
<td></td>
<td>; Test byte indirect display</td>
</tr>
<tr>
<td>00302</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00303</td>
<td>01E5</td>
<td>CALL DBDisplay ; Call the debug routine</td>
</tr>
<tr>
<td>00304</td>
<td>01E9</td>
<td>10 DB DBBC</td>
</tr>
<tr>
<td>00305</td>
<td>01E9</td>
<td>4243205265 DB 'BC Register',0</td>
</tr>
<tr>
<td>00306</td>
<td></td>
<td>; Call the debug routine</td>
</tr>
<tr>
<td>00307</td>
<td>01F5</td>
<td>CALL DBDisplay</td>
</tr>
<tr>
<td>00308</td>
<td>01FB</td>
<td>12 DB DBDE</td>
</tr>
<tr>
<td>00309</td>
<td>01F9</td>
<td>4445505265 DB 'DE Register',0</td>
</tr>
<tr>
<td>00310</td>
<td></td>
<td>; Call the debug routine</td>
</tr>
<tr>
<td>00311</td>
<td>0205</td>
<td>CALL DBDisplay ; Call the debug routine</td>
</tr>
<tr>
<td>00312</td>
<td>0206</td>
<td>14 DB DBHL</td>
</tr>
<tr>
<td>00313</td>
<td>0209</td>
<td>48C205265 DB 'HL Register',0</td>
</tr>
<tr>
<td>00314</td>
<td></td>
<td>; Call the debug routine</td>
</tr>
<tr>
<td>00315</td>
<td>0215</td>
<td>CALL DBDisplay</td>
</tr>
<tr>
<td>00316</td>
<td>0218</td>
<td>16 DB DBSP</td>
</tr>
<tr>
<td>00317</td>
<td>0219</td>
<td>3550205265 DB 'SP Register',0</td>
</tr>
<tr>
<td>00318</td>
<td></td>
<td>; Call the debug routine</td>
</tr>
<tr>
<td>00319</td>
<td>0220</td>
<td>013203 LXI B,Byte@BC ; Set up registers for byte tests</td>
</tr>
<tr>
<td>00320</td>
<td>0228</td>
<td>113303 LXI D,Byte@DE</td>
</tr>
<tr>
<td>00321</td>
<td>0228</td>
<td>213403 LXI H,Byte@HL</td>
</tr>
<tr>
<td>00400</td>
<td></td>
<td>; Test byte indirect display</td>
</tr>
<tr>
<td>00401</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00402</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00403</td>
<td>022C</td>
<td>CALL DBDisplay ; Call the debug routine</td>
</tr>
<tr>
<td>00404</td>
<td>0231</td>
<td>1A DB DB@BDC</td>
</tr>
<tr>
<td>00405</td>
<td>0232</td>
<td>4279746520 DB 'Byte at (BC)',0</td>
</tr>
<tr>
<td>00406</td>
<td></td>
<td>; Call the debug routine</td>
</tr>
<tr>
<td>00407</td>
<td>023F</td>
<td>CALL DBDisplay ; Call the debug routine</td>
</tr>
<tr>
<td>00408</td>
<td>0242</td>
<td>1C DB DB@DE</td>
</tr>
<tr>
<td>00409</td>
<td>0243</td>
<td>4279746520 DB 'Byte at (DE)',0</td>
</tr>
<tr>
<td>00410</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00411</td>
<td>0250</td>
<td>CALL DBDisplay ; Call the debug routine</td>
</tr>
<tr>
<td>00412</td>
<td>0253</td>
<td>1E DB DB@HLL</td>
</tr>
<tr>
<td>00413</td>
<td>0254</td>
<td>4279746520 DB 'Byte at (HL)',0</td>
</tr>
<tr>
<td>00414</td>
<td></td>
<td>; Set up the registers for word tests</td>
</tr>
<tr>
<td>00415</td>
<td>0261</td>
<td>013903 LXI B,Word@BC</td>
</tr>
<tr>
<td>00416</td>
<td>0264</td>
<td>113703 LXI D,Word@DE</td>
</tr>
<tr>
<td>00417</td>
<td>0267</td>
<td>213903 LXI H,Word@HL</td>
</tr>
<tr>
<td>00418</td>
<td></td>
<td>; Call the debug routine</td>
</tr>
<tr>
<td>00419</td>
<td>026A</td>
<td>CALL DBDisplay</td>
</tr>
<tr>
<td>00420</td>
<td>026D</td>
<td>20 DB DB@BDC</td>
</tr>
<tr>
<td>00421</td>
<td>026E</td>
<td>576F726420 DB 'Word at (BC)',0</td>
</tr>
<tr>
<td>00422</td>
<td></td>
<td>; Call the debug routine</td>
</tr>
<tr>
<td>00423</td>
<td>027B</td>
<td>CALL DBDisplay</td>
</tr>
<tr>
<td>00424</td>
<td>027E</td>
<td>22 DB DB@DE</td>
</tr>
<tr>
<td>00425</td>
<td>027F</td>
<td>576F726420 DB 'Word at (DE)',0</td>
</tr>
<tr>
<td>00426</td>
<td></td>
<td>; Call the debug routine</td>
</tr>
<tr>
<td>00427</td>
<td>028C</td>
<td>CALL DBDisplay</td>
</tr>
<tr>
<td>00428</td>
<td>028F</td>
<td>24 DB DB@HL</td>
</tr>
<tr>
<td>00429</td>
<td>0290</td>
<td>576F726420 DB 'Word at (HL)',0</td>
</tr>
<tr>
<td>00500</td>
<td>;#</td>
<td>Test DBOn/Off</td>
</tr>
<tr>
<td>00501</td>
<td></td>
<td>Disable debug output</td>
</tr>
<tr>
<td>00502</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00503</td>
<td>029D</td>
<td>CALL DBDisplay ; Disable debug output</td>
</tr>
<tr>
<td>00504</td>
<td>02A0</td>
<td>DB@BDC</td>
</tr>
<tr>
<td>00505</td>
<td>02A3</td>
<td>00A546869 DB 0DH,0AH, 'This message should NOT appear',0</td>
</tr>
<tr>
<td>00506</td>
<td></td>
<td>; Call the debug routine</td>
</tr>
<tr>
<td>00507</td>
<td>02CA</td>
<td>CALL DBOn</td>
</tr>
<tr>
<td>00508</td>
<td>02CD</td>
<td>DB@BDC</td>
</tr>
<tr>
<td>00509</td>
<td>02CA</td>
<td>00A446562 DB 0DH,0AH, 'Debug output has been re-enabled.',0</td>
</tr>
<tr>
<td>00600</td>
<td>;#</td>
<td>Test pass count logic</td>
</tr>
<tr>
<td>00601</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00602</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 10-2. (Continued)
Test*Pass@Loop:

RST 4
Debug input
DB 1111 ; Port number
RST 5
Debug output (value return from input)

JMP 0
Warm boot at end of testbed

; Debug subroutine
; Dummy values for byte and word displays

0322 BC Byte@BC: DB OEH
0333 DE Byte@DE: DB ODEH
0334 F1 Byte@HL: DB OF1H
0319 05C0 Word@BC: DW OBOCH
0320 05CED Word@DE: DW ODDEH
0321 05C0D Word@HL: DW OF01H

0323 9999999999 DW 9999H, 9999H, 9999H, 9999H, 9999H, 9999H, 9999H, 9999H
0324 9999999999 DW 9999H, 9999H, 9999H, 9999H, 9999H, 9999H, 9999H, 9999H
0325 9999999999 DW 9999H, 9999H, 9999H, 9999H, 9999H, 9999H, 9999H, 9999H

0326 Test@Stack:

; When only the testbed changes

Table 10-2. (Continued)
Figure 10-2. (Continued)
Chapter 10: Debugging a New CP/M System 335

Figure 10-2. (Continued)
Figure 10-2. (Continued)
Figure 10-2. (Continued)
Calling sequence

When the display code specifies a block of memory the sequence is:

- Display identifies which register(s) are to be displayed.
- When the display code specifies a block of memory the sequence is:

Calling sequence

- CALL DB(Display)
- DB Display Code
- DW StartAddress, EndAddress
- DB 'Caption String', 0
- DB Display Enabled

DB Display Enabled:

- SHLD DB Save HL
- POP H
- Save actual address of CALL
- Recover return address

DB Display:

- PUSH PSW
- Temporarily save flags to avoid them being changed by DAD SP
- Preserve stack pointer
- Correct for extra PUSH PSW needed to save the flags
- Save other user's registers
- The stack area is specially laid out to access these registers
- DB Call Address
- Recover flags
- Switch to local stack
- DB Start Address
- DB End Address
- HL = start address
- HL -> end address
- HL -> caption string

Figure 10-2. (Continued)
Figure 10-2. (Continued)
Figure 10-2. (Continued)
Chapter 10: Debugging a New CP/M System 341

<table>
<thead>
<tr>
<th>Location</th>
<th>Assembly Code</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>02227</td>
<td>0651 C3EE07</td>
<td>JMP</td>
<td>Display message and return</td>
</tr>
<tr>
<td>02228</td>
<td></td>
<td>DBMMSGO</td>
<td>Get saved value</td>
</tr>
<tr>
<td>02229</td>
<td></td>
<td>DBMSaveA</td>
<td>Display it and return</td>
</tr>
<tr>
<td>02230</td>
<td>0654 3A4B04</td>
<td>LDA</td>
<td>Get saved value</td>
</tr>
<tr>
<td>02231</td>
<td>0657 C39107</td>
<td>JMP</td>
<td>Display it and return</td>
</tr>
<tr>
<td>02232</td>
<td></td>
<td>DBMSaveB</td>
<td>Get saved value</td>
</tr>
<tr>
<td>02233</td>
<td></td>
<td>DBMSaveC</td>
<td>Get saved value</td>
</tr>
<tr>
<td>02234</td>
<td>0655 3A4904</td>
<td>LDA</td>
<td>Display it and return</td>
</tr>
<tr>
<td>02235</td>
<td>065D C39107</td>
<td>JMP</td>
<td>Display it and return</td>
</tr>
<tr>
<td>02236</td>
<td></td>
<td>DBMSaveD</td>
<td>Get saved value</td>
</tr>
<tr>
<td>02237</td>
<td></td>
<td>DBMSaveE</td>
<td>Get saved value</td>
</tr>
<tr>
<td>02238</td>
<td>0660 3A4804</td>
<td>LDA</td>
<td>Display it and return</td>
</tr>
<tr>
<td>02239</td>
<td>0663 C39107</td>
<td>JMP</td>
<td>Display it and return</td>
</tr>
<tr>
<td>02240</td>
<td></td>
<td>DBMSaveF</td>
<td>Display it and return</td>
</tr>
<tr>
<td>02241</td>
<td></td>
<td>DBMSaveG</td>
<td>Display it and return</td>
</tr>
<tr>
<td>02242</td>
<td>0666 3A4704</td>
<td>LDA</td>
<td>Display it and return</td>
</tr>
<tr>
<td>02243</td>
<td>0669 C39107</td>
<td>JMP</td>
<td>Display it and return</td>
</tr>
<tr>
<td>02244</td>
<td></td>
<td>DBMSaveH</td>
<td>Display it and return</td>
</tr>
<tr>
<td>02245</td>
<td></td>
<td>DBMSaveI</td>
<td>Display it and return</td>
</tr>
<tr>
<td>02246</td>
<td>066C 3A4604</td>
<td>LDA</td>
<td>Display it and return</td>
</tr>
<tr>
<td>02247</td>
<td>066F C39107</td>
<td>JMP</td>
<td>Display it and return</td>
</tr>
<tr>
<td>02248</td>
<td></td>
<td>DBMSaveJ</td>
<td>Display it and return</td>
</tr>
<tr>
<td>02249</td>
<td></td>
<td>DBMSaveK</td>
<td>Display it and return</td>
</tr>
<tr>
<td>02250</td>
<td>0672 3A0A04</td>
<td>LDA</td>
<td>Display it and return</td>
</tr>
<tr>
<td>02251</td>
<td>0675 C39107</td>
<td>JMP</td>
<td>Display it and return</td>
</tr>
<tr>
<td>02252</td>
<td></td>
<td>DBMSaveL</td>
<td>Display it and return</td>
</tr>
<tr>
<td>02253</td>
<td></td>
<td>DBMSaveM</td>
<td>Display it and return</td>
</tr>
<tr>
<td>02254</td>
<td>0678 3A0904</td>
<td>LDA</td>
<td>Display it and return</td>
</tr>
<tr>
<td>02255</td>
<td>067B C39107</td>
<td>JMP</td>
<td>Display it and return</td>
</tr>
<tr>
<td>02256</td>
<td></td>
<td>DBMSaveN</td>
<td>Display it and return</td>
</tr>
<tr>
<td>02257</td>
<td></td>
<td>DBMSaveO</td>
<td>Display it and return</td>
</tr>
<tr>
<td>02258</td>
<td>067E 2A4804</td>
<td>LHL</td>
<td>Display it and return</td>
</tr>
<tr>
<td>02259</td>
<td>0681 C30807</td>
<td>JMP</td>
<td>Display it and return</td>
</tr>
<tr>
<td>02260</td>
<td></td>
<td>DBMSaveP</td>
<td>Display it and return</td>
</tr>
<tr>
<td>02261</td>
<td></td>
<td>DBMSaveQ</td>
<td>Display it and return</td>
</tr>
<tr>
<td>02262</td>
<td>0684 2A4604</td>
<td>LHL</td>
<td>Display it and return</td>
</tr>
<tr>
<td>02263</td>
<td>0687 C30807</td>
<td>JMP</td>
<td>Display it and return</td>
</tr>
<tr>
<td>02264</td>
<td></td>
<td>DBMSaveR</td>
<td>Display it and return</td>
</tr>
<tr>
<td>02265</td>
<td></td>
<td>DBMSaveS</td>
<td>Display it and return</td>
</tr>
<tr>
<td>02266</td>
<td>068A 2A0904</td>
<td>LHL</td>
<td>Display it and return</td>
</tr>
<tr>
<td>02267</td>
<td>068D C30807</td>
<td>JMP</td>
<td>Display it and return</td>
</tr>
<tr>
<td>02268</td>
<td></td>
<td>DBMSaveT</td>
<td>Display it and return</td>
</tr>
<tr>
<td>02269</td>
<td></td>
<td>DBMSaveU</td>
<td>Display it and return</td>
</tr>
<tr>
<td>02270</td>
<td>0690 2A0B04</td>
<td>LHL</td>
<td>Display it and return</td>
</tr>
<tr>
<td>02271</td>
<td>0693 C30807</td>
<td>JMP</td>
<td>Display it and return</td>
</tr>
<tr>
<td>02272</td>
<td></td>
<td>DBMSaveV</td>
<td>Display it and return</td>
</tr>
<tr>
<td>02273</td>
<td></td>
<td>DBMSaveW</td>
<td>Display it and return</td>
</tr>
<tr>
<td>02274</td>
<td>0696 2A1304</td>
<td>LHL</td>
<td>Increment end address to make</td>
</tr>
<tr>
<td>02275</td>
<td>0699 23</td>
<td>INX</td>
<td>arithmetic easier</td>
</tr>
<tr>
<td>02276</td>
<td>069A 221304</td>
<td>SHL</td>
<td>DBEndAddress</td>
</tr>
<tr>
<td>02277</td>
<td></td>
<td>DBMSaveX</td>
<td>Display it and return</td>
</tr>
<tr>
<td>02278</td>
<td>069D 2A1104</td>
<td>LHL</td>
<td>DBStartAddress</td>
</tr>
<tr>
<td>02279</td>
<td>06A0 CD3A07</td>
<td>CALL</td>
<td>DBMSCheckEnd</td>
</tr>
<tr>
<td>02280</td>
<td>06A3 D81006</td>
<td>JC</td>
<td>End &gt; start</td>
</tr>
<tr>
<td>02281</td>
<td>06A6 CD6007</td>
<td>CALL</td>
<td>DBMSGIG</td>
</tr>
<tr>
<td>02282</td>
<td>06AF DDA0A2A2A20</td>
<td>DB</td>
<td>ODH, OAH, &quot;ERROR - Start Address &gt; End &quot;</td>
</tr>
<tr>
<td>02283</td>
<td></td>
<td>DBMNext</td>
<td></td>
</tr>
<tr>
<td>02284</td>
<td></td>
<td>DBMNextLine</td>
<td></td>
</tr>
<tr>
<td>02285</td>
<td>06CE CDC107</td>
<td>CALL</td>
<td>DBCRCLF</td>
</tr>
<tr>
<td>02286</td>
<td></td>
<td>DBMMAddressOK</td>
<td></td>
</tr>
<tr>
<td>02287</td>
<td></td>
<td>DBMSGIG</td>
<td></td>
</tr>
<tr>
<td>02288</td>
<td>06D1 CD6007</td>
<td>CALL</td>
<td>DBMSGIG</td>
</tr>
<tr>
<td>02289</td>
<td>06D4 202000</td>
<td>DB</td>
<td>&quot;0&quot;</td>
</tr>
<tr>
<td>02290</td>
<td>06D7 2A1104</td>
<td>LHL</td>
<td>DBStartAddress</td>
</tr>
<tr>
<td>02291</td>
<td>06DA CDB07</td>
<td>CALL</td>
<td>DBMSGIG</td>
</tr>
<tr>
<td>02292</td>
<td>06DD CDC07</td>
<td>CALL</td>
<td>DBColon</td>
</tr>
<tr>
<td>02293</td>
<td></td>
<td>DBMNextByte</td>
<td></td>
</tr>
<tr>
<td>02294</td>
<td>06ED 2A1104</td>
<td>LHL</td>
<td>DBStartAddress</td>
</tr>
<tr>
<td>02295</td>
<td></td>
<td>DBMNextByte</td>
<td></td>
</tr>
<tr>
<td>02296</td>
<td></td>
<td>DBMNextByte</td>
<td></td>
</tr>
<tr>
<td>02297</td>
<td>06ED 2A1104</td>
<td>LHL</td>
<td>DBStartAddress</td>
</tr>
<tr>
<td>02298</td>
<td>06E3 E5</td>
<td>MOV</td>
<td>Save memory address</td>
</tr>
<tr>
<td>02299</td>
<td>06E4 CD0007</td>
<td>CALL</td>
<td>DBMBlank</td>
</tr>
<tr>
<td>02300</td>
<td>06E7 E1</td>
<td>MOV</td>
<td>Recover current byte address</td>
</tr>
<tr>
<td>02301</td>
<td>06E8 7E</td>
<td>INX</td>
<td>Get byte from memory</td>
</tr>
<tr>
<td>02302</td>
<td>06E9 23</td>
<td>PUSH</td>
<td>Update memory pointer</td>
</tr>
<tr>
<td>02303</td>
<td>06EA E5</td>
<td>MOV</td>
<td>Save for later</td>
</tr>
<tr>
<td>02304</td>
<td>06ED CD0107</td>
<td>CALL</td>
<td>DBMSGIG</td>
</tr>
<tr>
<td>02305</td>
<td></td>
<td>DBMNextByte</td>
<td></td>
</tr>
</tbody>
</table>

Figure 10-2. (Continued)
02305 06FF CD3A07 CALL DBSMSCheckEnd ;Compare HL vs. end address
02306 06FF CAFE06 JZ DBSMSCheckEnd;Yes, end of area
02307 06FF 7D MOV A, L ;Check it at start of new line.
02308 06FF 60F ANI 00000111B ;(is address XXXOH?)
02309 06FF CAFE06 JZ DBSMSCheckEnd;Yes
02310 06FF C0E306 JMP DBSMNextASCIIByte ;No, loop back for another
02311 
02312 DBSMSCheckEnd: ;Display bytes in ASCII
02313 06FF CDC007 CALL DBSMColon ;Display .
02314 0701 2A1104 DHL DBStartAddress ;Start ASCII as beginning of line
02315 DBSMNextASCIIBytes: ;Get byte from memory
02316 0704 7E MOV A, M ;Save memory address
02317 0705 E5 PUSH H ;Save memory address
02318 0706 E7F ANI 01111111B ;Remove parity
02319 0707 4F MOV C, A ;Prepare for output
02320 0709 FE20 CPI ' ' ;Check if non-graphic
02321 070B D21007 JNC DBSMSCheckEndChar ;Char > space
02322 070E 0E2E MOV C, ' ' ;Display non-graphic as '.
02323 DBSMSCheckEndChar: ;Check if DEL (may be non-graphic)
02324 0710 FE7F CPI 'FH ;Check if DEl (may be non-graphic)
02325 0712 C21707 JNZ DBSMNotDEL ;No, it is graphic
02326 0715 0E2E MOV C, ' ' ;Force to '.
02327 
02328 DBSMNotDEL: ;Display character
02329 0717 CD0A05 CALL DBSMColon ;Display character
02330 071A E1 POP H ;Recove memory address
02331 071B 23 INX H ;Update memory pointer
02332 071C 221104 SHL DBStartAddress ;Update memory copy
02333 071F CD0A07 CALL DBSMCheckEnd ;Check if end of memory dump
02334 0722 CA3707 JZ DBSMExit ;Yes, done
02335 0725 7D MOV A, L ;Check if end of line
02336 0726 E4F ANI 00000111B ; by checking address = XXXOH
02337 072B CACE06 JZ DBSMNextLine ;Yes, start next line
02338 072B 7D MOV A, L ;Check if extra blank needed
02339 072E E603 ANI 00000011B ; if address is multiple of 4
02340 072F C20407 JNZ DBSMNextASCIIByte ;No -- go back for next character
02341 0731 C00D007 CALL DBSMBlank ;Yes, output blank
02342 0734 C30407 JMP DBSMNextASCIIByte ;Go back for next character
02343 
02344 
02345 DBSMExit: ;Output carriage return, line feed
02346 0737 C3C107 JMP DBSMCR LF ; and return
02347 
02348 DBSMCheckEnd: ;Compare HL vs. EndAddress
02349 073A D5 PUSH D ;Save DE (defensive programming)
02350 073B EB XCHD ;DE = current address
02351 073C 2A1304 LHL DBEndAddress ;Get end address
02352 073F 7A MOV A, D ;Compare MS bytes
02353 0740 BC CMP H ;Compare LS bytes
02354 0741 C2A607 JNZ DBSMCheckEndX ; Exit now as they are unequal
02355 0744 7B MOV A, E
02356 0745 BD CMP L
02357 0746 EB XCHD ;HL = current address
02358 0747 DB POP D ;Recove DE
02359 0748 C9 RET ;Return with condition flags set
02360 
02361 DP*DBD: ;(BC)
02362 0749 2A4804 LHL DBSave*C ;Get saved word value
02363 074C 7E MOV A, M ;Get byte addressed by it
02364 074D C39107 JMP DBSMDAH ;Display it and return
02365 
02366 DP*DBE: ;(DE)
02367 0750 2A4604 LHL DBSave*E ;Get saved word value
02368 0753 7E MOV A, M ;Get byte addressed by it
02369 0754 C39107 JMP DBSMDAH ;Display it and return
02370 
02371 DP*DBH: ;(HL)
02372 0757 2A9004 LHL DBSave*HL ;Get saved word value
02373 075A 7E MOV A, M ;Get byte addressed by it
02374 075B C39107 JMP DBSMDAH ;Display it and return
02375 
02376 DP*DBC: ;(BC+), (BC)
02377 075E 2A4804 LHL DBSave*C ;Get saved word value
02378 0761 5E MOV E, M ;Get word addressed by it
02379 0762 23 INX H

Figure 10-2. (Continued)
Figure 10-2. (Continued)
Figure 10-2. (Continued)
Figure 10-2. (Continued)
This routine helps debug code in which input instructions would normally occur. The opcode of the IN instruction must be replaced by a value of OE7H (RST 4).

This routine picks up the port number contained in the byte following the RST 4, converts it to hexadecimal, and displays the message:

```
This routine helps debug code in which output instructions would normally occur. The opcode of the OUT instruction must be replaced by a value of OEFH (RST 5).
```

```
This routine picks up the port number contained in the byte following the RST 5, converts it to hexadecimal, and displays the message:
```

```
Figure 10-2. (Continued)
```
Output to port XX : AA

where AA is the contents of the A register prior to the RST 5 being executed.

Control is then returned to the byte following the port number.

********

WARNING - This routine uses both DB$CONOUT and BDOS calls.

********

where AA is the contents of the A register prior to the RST 5 being executed.

Control is then returned to the byte following the port number.

Figure 10-2. (Continued)
The CP/M Programmer's Handbook

Figure 10-2. (Continued)
Figure 10-2. Debug subroutines (continued)

```assembly
03963 0905 FE41 CPI 'A' ;Check if < 'A'
03966 0907 DB RC ;Yes, terminate
03967 0908 FE47 CPI 'F' + 1 ;Check if > 'F'
03968 090A 00 RNC ;Yes, terminate
03969 090B DA37 SUI 'A' - 10 ;Combine with current result
03970 090D C31209 JMP DB$GHV$Shift$Left$4

03971 090F 0000 JMP DB$GHV$Shift$Left$4:
03972 0910 6630 DB$AHV$Hex$Digits:
03973 0911 0000 JMP DB$GHV$Shift$Left$4:
03974 0912 29 DAD H ;Shift HL left four bits
03975 0913 29 DAD H
03976 0914 29 DAD H
03977 0915 29 DAD H
03978 0916 65 ADD L ;Add binary value in LS 4 bits of A
03979 0917 64 MOV L.A ;Put back into HL total
03980 0918 C3F608 JMP DB$GHV$Loop ;Loop back for next character

04100 0000 2116
04101 07 A to upper
04102 08 Converts the contents of the A register to an uppercase
04103 09 letter if it is currently a lowercase letter
04104 0A Entry parameters
04105 0B
04106 0C A = character to be converted
04107 0D Exit parameters
04108 0E A = converted character
04109 0F
04110 10 DB$AHV$To$Upper:
04111 11 091B FE61 CPI 'a' ;Compare to lower limit
04112 12 091D DB ;No need to convert
04113 13 091E FE7B CPI 'z' + 1 ;Compare to upper limit
04114 14 0920 DO ;No need to convert
04115 15 0921 00 ANI 5FH ;Convert to uppercase
04116 16 0922 00 JMP DB$GHV$Loop
04117 17 0923 01 MOV L.A ;Put back into HL total
04118 18 0924 00 JMP DB$GHV$Loop
04119 19 0925 01 MOV L.A
04120 1A 0926 00 JMP DB$GHV$Loop

Figure 10-3. Console output from debug testbed run

```
containing all of the symbols in your program, along with their respective addresses. Once the program has been loaded by SID, you can refer to the memory image of your program not by address, but by the actual symbol name from your source code. SID also supports the “pass count” concept when using breakpoints.

**ZSID (Z80 Symbolic Debug)**

This is the Z80 CPU’s version of SID. The mini-assembler/disassembler uses Zilog instruction mnemonics rather than those used by Intel.

---

**Bringing Up CP/M for the First Time**

It is much harder to bring up CP/M on a new computer system than to debug an enhanced version on a system already running CP/M. You will often find yourself staring at a programmatic “brick wall” with no adequate debugging tools to assist you.

For example, you install the CP/M system on a diskette (using another CP/M-based computer system), put the diskette into the new computer, and press the **RESET** button. The disk head loads on the disk, and then—nothing! You cannot use any programs such as DDT or SID because you do not yet have CP/M up and running on the new computer. Or can you?

The answer is, wherever possible, debug the code for the new machine on an existing CP/M system. You may have to “fake” some aspects of the new bootstrap or BIOS so that the act of testing it on the host machine does not interact with the CP/M already running on it.

This scheme permits you to be fairly sure of your program logic before loading the diskette into the new machine. It will help pin down problems caused by hardware problems on the new computer.
The hardest situation of all is if you have only the new computer and the release diskettes from Digital Research. Your only option is to find a way of reading the CP/M image on the release diskette into memory, hand patch in new console and disk drivers (not a trivial task), write the patched image back onto a diskette, and resort to Orville Wright testing.

If you value your time, it is always more cost-effective to use another system with CP/M already installed. This is true even if the two systems do not have the same diskette format. You can still do the bootstrap and build the CP/M image on the host machine. Then download the image directly into the memory of the new machine and write it out to a diskette.

This downloading process does require, however, that the new computer have a read-only memory (ROM) monitor program. Depending on the capability of this ROM monitor program, you may have to hand patch into the new machine's memory a primitive “download” program that reads 8-bit characters from a serial port, stacking them up in memory and returning control to the monitor program when you press a keyboard character on the new machine’s console. In fact, some ROM monitor programs have a downloading program built in.

### Debugging the CP/M Bootstrap Loader

The CP/M bootstrap loader, as you may recall, is written on one of the outermost tracks on a diskette or hard disk. On a standard 8-inch single-sided, single-density diskette, CP/M’s bootstrap loader is stored on the first sector of the first track. The loader is brought into memory by firmware that gets control of the CPU when you turn your machine on or press the reset button.

The bootstrap has to be compact, as the diskette space on which it is stored is limited: no more than 128 bytes for standard 8-inch diskettes. This tends to rule out the use of the debug subroutines already described, so you have to fall back to more primitive techniques.

### Testing the Bootstrap Under CP/M

A bootstrap is best developed on a CP/M-based system. The task is easiest of all if you already have CP/M running on your new machine and are simply preparing an enhanced version of the bootstrap loader. In this case, you can test most of the code as though it were a user program running in the transient program area (TPA).

Most bootstraps get loaded into memory at location 0000H, so at the front of the code to be debugged you must put a temporary origin line that reads

```
ORG 100H
```
If you omit this and ask DDT to load the HEX file output by the assembler, it will load at the true origin, 0000H, and wipe out the contents of the base page for the version of CP/M that you are running. This will cause a system crash; you will have to press the RESET button and reload CP/M. When this happens, DDT does not tell you directly that anything is amiss; it just displays a "?" after your request to load the HEX file. You will discover that the system has "gone away" only when you try to do something else.

You also will need to adjust the addresses into which the bootstrap tries to load the CP/M image. If you do not, you will overwrite the version of CP/M presently running.

With these adjustments made, you can load the bootstrap under DDT and watch it execute, confirming that it does load the correct image into the correct addresses for debugging and transfer control to the BIOS jump vector. When everything appears to be functioning correctly, use the IF instruction to disable the debug code, reassemble the bootstrap, and write it onto a diskette. Then put the diskette into drive A and press RESET.

**Was the Bootstrap Loaded?**

At this point you must establish whether the bootstrap is being loaded into memory when the machine is turned on or RESET is pressed. The best way of doing this, and one that you can leave in place permanently, is to output a sign-on message as soon as the loader gets control. This requires hardware set up to prepare the USART (Universal Synchronous/Asynchronous Receive/Transmit) chip to output data, although some manufacturers write this initialization code into the firmware that loads the bootstrap. A suitable sign-on message would be the following:

```
CP/M Bootstrap Loader : Vn 1.0 11/18/82
```

If you do not see this message, assume that control is not being transferred to the bootstrap loader. This will be useful in the future if someone should call you with a complaint that CP/M cannot be loaded. If this message does not appear, they probably do not have CP/M on the disk.

**Did the Bootstrap Load CP/M?**

This is a harder question to answer than whether the bootstrap itself has been loaded, especially if the bootstrap loader sign-on is displayed and then the system crashes. A sign-on message early in the BIOS cold boot processing can confirm the correct transfer of control into the BIOS.

If the problems with the bootstrap program are severe, you may have to adapt the memory-dump debugging subroutine, dumping the contents of memory to the console in order to see what information the bootstrap loader is placing in memory. Display 100H bytes starting from the front of the BIOS jump vector. This
Chapter 10: Debugging a New CP/M System

Table has an immediately recognizable pattern of 0C3H values every three bytes.

You should also check to see that the bootstrap is loading the correct number of sectors from the disk into memory. If it loads too few, CP/M may sign on only to crash a few moments later because it attempts either to execute code or access a constant at the end of the BIOS. If the bootstrap loads too many sectors from the disk, the excess may "wrap around" the top of memory and overwrite the bootstrap itself, down at location 0000H, before it has completed its task. In this case, you would see only the sign-on for the bootstrap, not for the BIOS.

Debugging the BIOS

Rather than try to debug the BIOS as a single piece of code, debug it as a series of separate functional modules.

Notwithstanding current "top-down" philosophies of dealing with overall structure first, it can be quicker to debug the low-level subroutines in a device driver first. This gives you a solid base on which to build.

The BIOS can be divided up into its constituent modules as follows:

Character input
  Interrupt service
  Non-interrupt service

Character output

Interrupt routines
  Real time clock
  Watchdog timers

Disk drivers
  High-level (debloking)
  Low-level (physical I/O)

Plan to write a testbed program for each of these modules. This testbed code serves two purposes; first, it provides a means of transferring control into the module under test in a controlled way. Second, it includes the necessary modules or dummy modules to "fool" the module under test into responding as if it were running in a complete BIOS under CP/M.

Using the testbed, you can check every part of the module's logic except the part that may be time-critical. Problems caused by timing, such as interrupts disabled for too long or code that is too slow or too fast for a particular peripheral controller chip, tend to show up only when you are testing on the final hardware and when you are running your new BIOS under CP/M.
What You Should Test for in the BIOS

Describing fully how to debug each module in the BIOS could fill several books. Remember that you are trying to establish the absence of errors using a technique that, by its very nature, tends to show only their presence.

There are two basic approaches to debugging. One is the plodding method, checking every aspect of the code to ensure that every feature really does work. The second is to try to do something useful with the code.

Plan to use both. Start with the plodding method, testing each feature under control of the testbed until you are sure that it is working in vitro. When all of the BIOS modules have been tested individually, build a CP/M system and try to do some useful work with it. Trying to use the system for actual work testing in vitro can be a good test.

Feature Checklist

Make a list of the specific features included in the various BIOS modules. Then devise specific test sequences that will show that each of the features is working correctly.

The same testbed code can often test all of the features of a driver module. If it cannot, create a new testbed for the more exotic features.

Keep the testbed routines. Experience shows that they are most often needed shortly after you have erased them. Even after you have tested the BIOS, the testbed routines will come in handy if you decide to enhance a particular driver later on. You can extract the driver code from the BIOS, glue it together with the testbed, and test the new feature code in isolation from the BIOS.

The following sections show example testbeds for the various drivers, along with example checklists. These checklists were used to test the example BIOS routines shown in earlier chapters.

Character Drivers

Figure 10-4 shows the code for an example testbed routine for character I/O drivers in the BIOS. This code would be followed by the actual character I/O drivers, exactly as they would appear in the BIOS except that all IN and OUT instructions would be replaced with RST 4's and 5's respectively (see Figure 10-2) so that you could enter input values and inspect output values on the console.

This example contains the initialization code for the debug package shown in Figure 10-2 and the code setting up an RST 6 used to “fake” incoming character interrupts.

The main testbed loop consists of a faked incoming character interrupt followed by optional calls to CONIN or CONOUT, the return of control to DDT, or a loop back to fake another character interrupt. You can only return control to DDT if you used DDT to load the testbed and driver programs in the first place.
The complete source file consists of three components:

1. The testbed code shown here
2. The character I/O drivers destined for the BIOS
3. The debug package shown in Figure 10-2.

Figure 10-4. Testbed for character I/O drivers in the BIOS
The CP/M Programmer's Handbook

```
Test#Stack:
;
; Dummy routines for those shown in other figures
;
; BIOS routines (Figure 8-10)
;
CONST:
CONIN: ;BIOS console status
CONOUT: ;BIOS console output
Character$Interrupt: ;Interrupt service routine for incoming chars.
;
Debug routines (Figure 10-2)
;
DBSInit: ;Debug initialization
DBSMSOI: ;Display message in-line
DBSCONINU: ;Set uppercase character from keyboard
DBSDisplay: ;Main debug display routine
DBSEQU 02 ;Display code for DBSDisplay

Figure 10-4. Testbed for character I/O drivers in the BIOS (continued)

Executing an RST 7 without using DDT will cause a system crash, as DDT sets up the necessary JMP instruction at location 0038H in the base page.

The faked incoming character interrupt transfers control directly to the interrupt service routine in the BIOS (see the example in Figure 8-10, line 04902, label Character$Interrupt). This reads the status ports of each of the character devices; you can enter the specific status byte values that you want. If you enter a value that indicates that a data character is "incoming," you will be prompted for the actual 8-bit data value to be "input." You can make the interrupt service routine appear to be inputting characters and stacking characters up in the input buffer. For debugging purposes, reduce the size of the input buffer to eight bytes. Making it larger means you will have to input more characters to test the buffer threshold logic. To check the interrupt service routine, you will pass through the main testbed loop doing nothing but faking incoming character interrupts and entering status and data values. The data characters will then be stacked up in the input buffer.

To check the correct functioning of the interrupt service routines, you can stay in control with DDT from the outset. Alternatively, you can just use DDT to load the testbed/driver HEX file, loop around inputting several characters, and then request that the testbed return control to DDT. Then you can use DDT to inspect the contents of the device table(s) and input buffers.

Another possibility is to create debugging routines that display the contents of the device table in a meaningful way, with each field captioned like this:

```
DEVICE TABLE 0
Status Port 81 Data Port 80
Output Ready 01 Input Ready 02
DTR high 40
Reset Int. Prt D8 Reset Int. Val. 20
;
;
Status Byte 1
Output Suspended
Output Xon Enabled
```
This display device table routine will require a fair amount of effort to code and debug — but it will pay dividends. You can obtain a complete “snapshot” of the device table without having to decode hexadecimal memory dumps and individual bits. Constant values in the device tables are also displayed, so that if a bug in your code corrupts the table, you will know about it immediately.

The next section shows examples of the specific tests you need to make, along with a description of the strategy you can use.

**Interrupt Service Routine Checklist**

In a functioning BIOS, control is transferred to the interrupt service module whenever an incoming character causes an interrupt. In the example BIOS in Figure 8-10 (line 4900), the code scans each character device in turn to determine which one is causing the interrupt.

When you are debugging the interrupt service routines using the “fake” input/output instructions, you will have to enter specific status byte values. Refer to the device table declarations in Figure 8-10, line 1500, to determine what values you must enter to make the service routine think that an incoming character is arriving or that data terminal ready (DTR) is high or low.

Start the debugging process using the first device table. Then repeat the tests on the other device tables.

The following is a checklist of features that should be checked in debugging the interrupt service routine:

*Are all registers restored correctly on exit from the interrupt servicing?*

Using DDT, start execution from the beginning of the testbed. Set a breakpoint (with the G100,nnnn command) to get control back immediately before the CALL Character$Interrupt. Use the X command to display all of the registers, and then, by using the G,nnnn command, you set a breakpoint at the instruction that immediately follows the CALL Character$Interrupt. The character drivers will prompt you for the status values. Enter 00 (which indicates that no character is incoming). Display the registers again — their values should be the same. Remember to check the value of the stack pointer and the amount of the stack area that has been used.

**NOTE:** Do not be too surprised if you lose control of the machine when you first try this test. You may have some fundamental logic errors initially. If the system crashes, reset it, reload CP/M, and then start the test again. This time, rather than setting the second breakpoint at the instruction following the CALL Character$Interrupt, venture down into the Character$Interrupt code and go through the code a few instructions...
at a time, setting breakpoints before any instructions that could cause a
transfer of control. Find out how far you are getting into the driver before
it either jumps off into space or settles into a loop.

Does the service routine push a significant number of bytes onto the stack
after an interrupt has occurred?

When you get control back after the CALL Character$Interrupt, use
the D (dump) command to dump the stack area’s memory on the console.
Check how far down the stack came by looking for the point where the
constants that used to fill the stack area are overwritten by other data.

The example BIOS in Figure 8-10 saves only the contents of the HL
register pair on the pre-interrupt stack. It then switches over to a private
BIOS stack to save the contents of the rest of the registers and service the
interrupt.

Are data characters added to the input buffer correctly?

“Input” a noncontrol character via the Character$Interrupt routine.
Then check the contents of the appropriate device table. The character
count and the put offset should both be set to one. Then check the contents
of the input buffer itself; does it contain the character that you
“input?”

Are control characters added to the input buffer correctly?

“Input” a control character such as 01H. Do not use ETX, ACK, XON, or
XOFF (03H, 06H, 11H, and 13H, respectively); these may cause side effects
if you have errors in the protocol handling logic. Check that the character
is stored in the next byte of the input buffer and that the character and
control counts are set to two and one, respectively. The put offset should
also be set to two.

When the input buffer full threshold is reached, does the driver output the
correct protocol character?

Set the first status byte in the first device table to enable input XON
or RTS protocol, or both. Then go round the main testbed loop putting
characters into the input buffer. Check the console display to see if the
drivers output the correct values when the buffer is almost full (the default
threshold is when five bytes remain). The driver should then drop the RTS
line or output an XOFF character or both, according to the input protocol
that you enabled.

When the input buffer is completely full, does the driver respond correctly?

This is an extension of the test above. Input one more character than
can fit into the buffer. Check to see that the drivers do not stack the
character into the input buffer and that a BELL character (07H) is output to
the data port.
Are protocol characters XON/XOFF recognized and the necessary control flags set or reset?

Reload the testbed and drivers. Set the status byte to enable the output XON/XOFF protocol. Then use the Character$Interrupt routine to input an XOFF character (13H). Check to see that the XOFF character has not been put into the input buffer. Instead, the status byte should be set to indicate that output has indeed been suspended.

Input an XON and check to see that the output suspended flag has been reset.

Does the driver detect and reset hardware errors correctly?

Proceed as though you were going to input a character into the input buffer, but instead enter a status byte value that indicates that a hardware error has occurred (enter the value given in the device table for DTS$Detect$Error$Value).

Check that the driver detects the error status and outputs the correct error-reset value to the appropriate control port.

Non-Interrupt Service Routine Checklist

In a “live” BIOS, non-interrupt service routines are accessed via the CONIN and CONST entry points in the BIOS jump vector. During debugging, the testbed can call the CONIN and CONST code directly.

Is input redirection functioning? Does control arrive in the driver with the correct device table selected?

This is best tested directly with DDT. Use the Gnnnn,bbbb command to transfer control into the CONIN code with a breakpoint at the RET instruction at the end of the Select$Device$Table routine (see Figure 8-10, line 04400). Check that the DE register pair is pointing at device table 0. If it is not, you will have to restart the test. Use the Tn command to make DDT trace through the Select$Device$Table subroutine to find the bug.

Are characters returned correctly from the buffer?

Use the testbed to “input” a character or two. Then use the testbed to make several entries into CONIN. Check the characters returned from the buffer.

Are the data character and control character counts correctly decremented?

After each character has been removed from the buffer by CONIN, use DDT to examine the device table and check that the data character and control character counts have been decremented correctly. Also check that the get pointer has moved up the input buffer.

When the buffer “almost empty” threshold is reached, does the driver emit the correct protocol character or manipulate the request to send (RTS) line correctly?

Use DDT to enable the input RTS or XON protocol or both. Then input characters into the input buffer until it reaches the buffer full threshold (the
default is when only five spare bytes remain in the buffer). Confirm that “buffer almost full” processing occurs. Then make repetitive calls to CONIN to flush data out of the buffer. Check that the “buffer emptying” processing occurs when the correct threshold is reached. For RTS protocol, the driver should output a raise RTS value to the specified RTS control port. For XON, the driver should output an XON character to the data port (after first having read the status port to ensure that the hardware can output the character).

**Does the driver handle buffer “wraparound” correctly?**

Input characters to the input buffer until it becomes completely full. Then make a single CONIN call to remove the first character from the buffer. Follow this by inputting one more character to the buffer. Check that the get pointer is set to one and the put pointer set to zero.

Next, make successive CONIN calls to empty the buffer. Then input one more character to the buffer. Check that this last character is put into the first byte of the input buffer.

**Can the driver handle “forced input” correctly?**

Using DDT, set the forced input pointer to point to a 00-byte-terminated string; for example, use one of the function key decode default strings. (In Figure 8-10, the forced input pointer is initialized to point to a “startup string”—this is declared at the beginning of the configuration block at line 00400.)

Using DDT, call the CONST routine and check that it returns with A = OFFH (indicating that there appears to be input data waiting).

Make successive calls to CONIN and confirm that the data bytes in the forced input string are returned. Check that the forcing of input ends when the 00H-byte is detected.

**Does the console status routine operate correctly when it checks for data characters in the buffer, control characters in the buffer, and forced input?**

Input a single noncontrol character, such as 41H, into the input buffer. Using DDT, check that the second status byte in the device table has the fake type-ahead flag set to zero. Call the CONST routine — it should return with A = 0FFH (meaning that there is data in the buffer). Then set the fake type-ahead bit in the second status byte and call CONST again. It should return with A = 00H (meaning that there is now “no data” in the buffer). Input a single control character into the buffer. Now CONST should return with A = 0FFH because there is a control character in the buffer.

**Does the driver recognize escape sequences incoming from keyboard function keys?**

This is a difficult feature to test when the real time clock routine is not running. The driver uses the watchdog timer to wait until all characters in
the escape sequence have arrived. You will therefore have to modify the code in CONIN so that the watchdog timer appears to time out immediately, rather than waiting for the real time clock to tick. To make this change, refer to Figure 8-10, line 2200; this is the start of the CONIN routine. Look for the label CONIN$Wait$For$Delay$. A few instructions later there is a JNZ CONIN$Wait$For$Delay$. Using DDT, set all three bytes of this JNZ to 00H.

Then, using the testbed, input the complete escape sequence into the input buffer. For example, input hexadecimal values 1B, 4F, 51 (ESCAPE, O, P), which correspond to the characters emitted on a VT-100 terminal when FUNCTION KEY 1 (PFI) is pressed.

Next, use the testbed to make successive calls to CONIN. You should see the text associated with the function key (FUNCTION KEY 1, LINE FEED) being returned by CONIN.

Repeat this test using different function key sequences, including a sequence that does not correspond to any of the preset function keys. Check that the escape sequence itself is returned by CONIN without being changed into another string.

*Can the driver differentiate between a function key and the same escape sequence generated by discrete key strokes?*

This is almost the same test as above. Make the same patch to the CONIN code, only this time do not enter the complete escape sequence into the buffer. Enter only the hex characters 1B and 4F. Make sure that the CONIN routine does not substitute another string in place of this quasi-escape sequence.

This test only mimics the results of manually entering an escape sequence. You could not press the keys on a terminal fast enough to get all three characters into the input buffer within the time allowed by the watchdog timer.

**Character Output Checklist**  *Can the driver output a character?*

The CONOUT option in the testbed calls CONIN first to get a character. To start with, you may want to use DDT to set the C register to some graphic ASCII character such as 41H (A), and transfer control into CONOUT directly. Check that CONOUT reads the USART's status, waits for the output ready value, and then outputs the data to the data port. Note that the testbed will output all characters waiting in the input buffer (or forced input) when you select its CONOUT option. This is a convenience for advanced testing of the drivers—for initial testing you may want to modify the testbed to make only one call to CONIN and CONOUT and then return to the top of the testbed loop.
Does the driver suspend output when a protocol control flag indicates that output is to be suspended?

Using DDT, set the status byte in the device table to enable output XON/XOFF protocol. Then input an XOFF character and confirm that the output suspended bit in the status byte is set. Output a single character, and using DDT, confirm that the driver will remain in a status loop waiting for the output suspended bit to be cleared. Clear the bit using DDT and check that the character is output correctly.

When using ETX/ACK protocol, does the driver output an ETX after the specified number of characters have been output, then indicate that output is suspended?

For debugging purposes, alter the ETX message count value in the device table to three bytes. Then output three bytes of data via CONOUT. Check that the driver sends an ETX character (03H) after the three bytes have been output and that the output suspended flag in the status byte has been set.

Then input an ACK character (06H). Check that this character is not stored in the input buffer and that the output suspended flag is cleared.

Does the driver recognize and output escape sequences?

Input an ESCAPE, “t” (1BH, 74H) into the input buffer. Then output them via CONOUT. Using DDT, check that the CONOUT routine recognizes that an escape sequence is being output and selects the correct processing routine. In this case, the forced input pointer should be set to point at the ASCII time of day in the configuration block.

Does each of the escape sequence processors function correctly? Can the time and date be set to specified values using escape sequences?

Repeat the test above using all of the other escape sequences to make sure that they can be recognized and that they function correctly.

Real Time Clock Routines

A separate testbed program, shown in Figure 10-5, is used to check these routines. It calls the interrupt service routine directly to simulate a real time clock “tick,” and then displays the time of day in ASCII on the console.

As you can see, the testbed makes a call into the debug package’s initialization routine, DB$Init, and then uses an RST 6 to generate fake clock “ticks.”

There is a JMP instruction in the testbed that bypasses a call to Set$Watchdog. Remove this JMP, either by editing it out or by using DDT to change it to NO OPERATIONs (NOP, 00H) when you are ready to test the watchdog routines.

Real Time Clock Test Checklist

Is the clock running at all?

Using DDT, trace through the interrupt service routine logic. Check that the seconds are being updated.
Chapter 10: Debugging a New CP/M System

363

The complete source file consists of three components:
1. The testbed code shown here
2. The real-time clock driver destined for the BIOS.
3. The debug package shown in Figure 10-2.

FFFE = TRUE EQU 0FFFFH
0000 = FALSE EQU NOT TRUE

For conditional assembly of RST instructions in place of IN and OUT instructions in the drivers. USE RST 6 for fake clock tick.

LXI SP, Test#Stack ;Use local stack
CALL DB#Init ;Initialize the debug package
MV 0, A, JMP ;Set up RST 6 with JMP opcode
STA RST6
LXI H, RTC#Interrupt ;Set up RST 6 JMP address
SHLD RST6 + 1

JMP TestbedLoop ;(== REMOVE THIS JMP WHEN READY TO TEST WATCHDOG ROUTINES

LXI B, 50 ;50 ticks before timeout
LXI H, WD#Timeout ;Address to transfer to
CALL Set#Watchdog ;Set the watchdog timer

;Make repeated entry to RTC interrupt routine to ensure that clock is correctly updated

;TestbedLoop:
LXI SP, Test#Stack ;Set registers to known pattern
MV 0, A, OAH
LXI B, 0BBCH
LXI D, 0BBEEH
LXI H, 0FFH.H
LXI RST 6 ;Fake interrupt clock

;Setup 436C6F6368
CALL DB#MSI ;Display in-line message
DB 'Clock =', 0

;RTC Interrupt
LXI H, TimeIn%ASCII ;Get address of clock in driver
CALL DB#MSG ;Display current clock value
; (Note: TimeIn%ASCII already has a line feed character in it)

DB#MSI ;Display in-line message
DB ODH, 0 ;Carriage return

JMP TestbedLoop ;Control arrives here when the watchdog timer times out

;WD#Timeout:
CALL DB#MSI
DB ODH, 0AH, 'Watchdog timed out', 0
RET ;Return to watchdog routine

DB 9999H, 9999H, 9999H, 9999H, 9999H, 9999H, 9999H, 9999H, 9999H
DB 9999H, 9999H, 9999H, 9999H, 9999H, 9999H, 9999H, 9999H, 9999H

Test#Stack:
; Dummy routines for those shown in other figures
; BIOS routines (Figure 0-10)
; RTC#Interrupt: ;interrupt service routine for clock tick
; Set#Watchdog: ;Set watchdog timer
; TimeIn%ASCII: ;ASCII string of HH:MM:SS, LF, 0
; Debug routines (Figure 10-2)
; DB#Init: ;Debug initialization
; DB#MSI: ;Display message in-line
; DB#MSG: ;Display message

Figure 10-5. Testbed for real-time-clock driver in the BIOS
Are the hours, minutes, and seconds carrying over correctly?

Let the testbed code run at full speed. You should see the time being updated on the console display—although it will be updated much more rapidly than real time.

Use DDT to set the minutes to 58 and then let the clock run again. Does it correctly show the hour and reset the minutes to 00? Then set the hours to 11 and the minutes to 58 and let the clock run. Do minutes carry over into hours and are hours reset to 0?

Repeat these tests with the clock update constants set for 24-hour format.

Is the clock interrupt service routine restoring the registers correctly?

Using DDT, check that the registers are still set correctly on return from the clock interrupt service routine.

How much of a load on the pre-interrupt stack is the service routine imposing?

Check the "low water mark" of the preset values remaining in the testbed stack area to see how much of a load the interrupt service routine is imposing on the stack.

Can the watchdog timer be set to a nonzero value? Can it be set back to zero?

Using the second part of the testbed, call the Set$Watchdog routine, and then monitor the testbed's execution as the watchdog timer times out. Check that the registers and stack pointer are set correctly when control is transferred to the timeout routine. Also check that control is returned properly from this routine, and thence from the interrupt service routine.

Disk Drivers

It is only feasible to check the low-level disk drivers in isolation from a real BIOS, as the BDOS interface to the deblocking code is very difficult to simulate. The testbed shown in Figure 10-6 serves only as a time-saver. It does not test the interface to the subroutines. Use DDT to set up the disk, track, and sector numbers, and then monitor the calls into SELDSK, SETTRK, SETSEC, SETDMA, and the read/write routines.

Unless you have the same disk controller on the host system as you do on the target machine, you will have to use the fake input/output system described earlier in this chapter, rather than attempt to read and write on real disks.

You can see that the testbed, after initializing the debugging package, makes calls to SELDSK, SETTRK, SETSEC, and SETDMA. It then calls a low-level read or write routine. The low-level routine called depends on which driver you wish to debug. For the standard floppy diskette driver shown in Figure 8-10, use Read$No$Deblock and Write$No$Deblock. For the 5 1/4-inch diskettes, use Read$Physical and Write$Physical. You will have to use DDT to set up some of the variables required by the low-level drivers that would normally be set up by the deblocking code.
Chapter 10: Debugging a New CP/M System

The complete source file consists of three components:
1. The testbed code shown here
2. The Disk I/O drivers destined for the BIOS
3. The debug package shown in Figure 10-2.

Testbed for disk I/O drivers in the BIOS

The complete source file consists of three components:

1. The testbed code shown here
2. The Disk I/O drivers destined for the BIOS
3. The debug package shown in Figure 10-2.

Figure 10-6. Testbed for disk I/O drivers in the BIOS
Figure 10-6. (Continued)
Before issuing the write call, the testbed fills the disk buffer with a known pattern. This pattern is checked on return from a read operation.

For both reading and writing, the testbed shows the contents of the A register. If you have added the enhanced disk error handling described in the previous chapter, the return value in A must always be zero.

**Disk Driver Checklist**

**Does SELDSK return the correct address and set up the required system variables?**

- Check that the correct disk parameter header address is returned for legitimate logical disks. Check, too, that it returns an address of 0000H for illegal disks.
- Check that any custom processing, such as setting the disk type and deblocking requirements from extra bytes on the disk parameter blocks, is performed correctly.

**Does the SETTRK and SETSEC processing function correctly?**

- Using DDT, check that the correct variables are set to the specified values.

**Does the driver read in the spare-sector directory correctly?**

- Set up to execute a physical read and, using DDT, trace the logic of the READ entry point. Check that the spare-sector directory would be loaded into the correct buffer. If you are using fake input/output, use DDT to patch in a typical spare-sector directory with two or three "spared-out" sectors.

**Does the driver produce the correct spare sector in place of a bad one?**

- Continuing with the physical read operation, check that, for "good" track/sectors, the sector-sparing logic returns the original track and sector number, and for "bad" track/sectors, it substitutes the correct spare track and sector. If you are using sector skipping, check that the correct number of sectors is skipped.

**Can a sector be read in from the disk?**

- Continuing further with the physical read, check that the correct sector is read from the specified disk and track. If you are using real I/O (as
opposed to faking it), the “ripple pattern” set by the testbed can be used, or you can fill the disk buffer area with some known pattern (using DDT’s F command) so you can tell if any data gets read in.

Make sure you do not have any disks or diskettes in the computer system that are not write-protected — you may inadvertently write on a disk rather than read it during the early stages of testing.

**Can a sector be written to the disk?**

Using DDT, set up to write to a particular disk, track, and sector. Remove any write protection that you put on the target disk during earlier testing. You can either use the testbed’s ripple pattern or fill the disk buffer area with a distinctive pattern. Write this data onto the disk, fill the buffer area with a different pattern, and read in the sector that you wrote. Check that the disk buffer gets changed back to the pattern written to the disk.

**Does the driver display error messages correctly?**

Rather than deliberately damaging a diskette to create errors, use DDT to temporarily sabotage the disk driver’s logic. Make it return each of the possible error codes in turn, checking each time that the correct error message is displayed.

For each error condition in turn, check that the disk driver performs the correct recovery action, including interacting with the user and offering the choice of retrying, ignoring the error, or aborting the program.

---

**Live Testing a New BIOS**

Given that the drivers have passed all of the testing outlined above, you are ready to pull all of the BIOS pieces together and build a CP/M image.

For your initial testing, disable the real time clock, and use simple, polled I/O for the console driver if you can. It is important to get something up and running as soon as possible, and it is easier to do this without possible side effects from interrupts.

Prepare a complete listing of the BIOS and plan to spend at least an hour checking through it. Take a dry run through the console and disk driver — if there are any serious bugs left in these two drivers, CP/M may not start up. Remember that once the BIOS cold boot code has been executed and control is handed over to the CCP, the BDOS will be requested to log in the system disk, and this involves reading in the disk’s directory.

Pay special attention to checking some of the major data structures. Make certain that everything is at a reasonable place in memory; for example, if the last address used by the BIOS is greater than 0FFFFH, you will need to move the entire CP/M image down in memory.
Chapter 10: Debugging a New CP/M System

Then build a system disk, load it into the machine, and press the RESET button. You should see the bootstrap sign on, then the BIOS, and after a pause of about one second, the A> prompt (or 0A> if you have included the special feature that patches the CCP).

If you see both sign-on messages but do not get an A> prompt, a likely cause of the problem is in the disk drivers. Alternatively, the directory area on the disk may be full of random data rather than 0E5H's.

If you cannot see what is wrong with the system, you might try faking the disk drivers to return a 128-byte block of 0E5H's for each read operation. The CCP should then sign on.

Once you do have the A> prompt, you can proceed with the system checkout. Start by checking that the warm boot logic works. Type a CONTROL-C. There should be a slight pause, and the A> prompt should be output again.

Next, check that you can read the disk directory by using the DIR command. If you have an empty directory, you should get a NO FILE response. If you get strange characters instead, you either forgot to initialize the directory area or the disk parameter block is directing CP/M to the wrong part of the disk for the file directory. If the system crashes, there is a problem with the disk driver.

Check that you can write on the disk by entering the command SAVE 1 TEST. Then use the DIR command to confirm that file TEST shows up in the file directory. If it does, use the ERA command ERA TEST and do another DIR command to confirm that TEST has indeed been erased.

If TEST either does not show up on the disk or cannot be erased, then you have a problem with the disk driver WRITE routine.

Put a standard CP/M release diskette into drive B and use the DIR command to check that you can access the drive and display a disk directory. If you do, then load the DDT utility and exit from it by using a G0 (G, zero) command. This further tests if the disk drivers are functioning correctly.

To test the deblocking logic (if you are using disks that require deblocking), use the command:

\[ \text{PIP A:=B:*.V[V]} \]

This copies all files from drive B to drive A using the verify option. It is a particularly good test of the system, and if you have any problems with the high-level disk drivers and deblocking code, you will get a Verify Error message from PIP. You can also get this message if you have hardware problems with the computer's memory, so run a memory test if you cannot find anything obviously wrong with the deblocking algorithm.

To completely test the deblocking code, you need to use PIP to copy a file of text larger than the amount of memory available. Thus, you may have to create a large text file using a text editor just to provide PIP with test data.

With the disk driver functioning correctly, rebuild the system with the real time clock enabled. Bring up the new system and check that the ASCII time of day is
being updated in the configuration block; use DDT to inspect this in memory. Set
the clock to the current time, let it run for five minutes, and see if it is still accurate.
You may have to adjust one of the initialization time constants for the device that is
providing the periodic interrupts for the clock.

Rebuild the system yet again, this time with the real interrupt-driven console
input and the real console output routines. Check that the system comes up
properly and that the initial forced-input startup string appears on the console.

Check that when you type characters on the keyboard they are displayed as
you type them. If not, there could be a problem with either the CONIN or
CONOUT routines. Experimentally type in enough characters to fill the input
buffer. If the terminal's bell starts to sound, the interrupt service routine is
probably not the culprit. Check the CONOUT routine again.

Check that the function key decode logic is working correctly. With the A>
prompt displayed, press a function key. The CONIN driver should inject the
correct function key string and it should appear on the terminal. For example,
with the BIOS in Figure 8-10, pressing PF1 on the VT-100 terminal should produce
this on the display:

A>Function Key1
Function?
A>

The CCP does not recognize "Function" as a legitimate command name, nor is
there such a COM file — hence the question mark.

Using DDT, write a small program that outputs ESCAPE, "t" to the console, and
check that the ASCII time of day string appears on the console. This checks that
the escape sequence has been recognized.
This chapter contains the narrated source code for several useful utility programs. Two groups of such programs are included—those that supplement Digital Research's standard utility programs, and those that work in conjunction with features shown in the enhanced BIOS (Figure 8-10).

To avoid unnecessary detail, the programs shown in this chapter are all written in the C language. C is a good language to use for such purposes since it can show the overall logic of a program without the clutter of details common in assembly language.

In order to reuse as much source code as possible, this chapter includes a "library" of all the general-purpose C functions that can be called from within any of the utility programs. This file, called "LIBRARY.C", is shown in Figure 11-1. Once a utility program has been compiled, the necessary functions from the library can be linked with the utility's binary output to form the "COM" file.
Figure 11-1. LIBRARY.C, commonly used functions, in C language
Chapter 11: Additional Utility Programs

Figure 11-1. (Continued)
-ve integer if char. in string is > char. in substring
+ve integer if char. in string is < char. in substring

```c
int count;  /* Used to access chars in string and substring */
count = 0;  /* Start with the first character of each */
while (toupper(string[count]) == toupper(substring[count]))
    if (substring[++count] == '\0') /* Last char. in substring */
        return 0;  /* Indicate equality */
return substring[count] - string[count];  /* Compare" chars. */
```

/* End of usstrcmp */

```c
/* Compare file names */
comp fname(scb.name) {
    struct _scb *scb;  /* Pointer to search control block */
    char *name;  /* Pointer to file name */

    NAME_EQ if the names match the mask
    NAME_LT if the name is less than the mask
    NAME_GT if the name is greater than the mask
    NAME_NE if the name is not equal to the mask (but the outcome
    is ambiguous because of the wildcards in the mask)
}
```

```c
/* Entry parameters */
int count;  /* Count of the number of chars. processed */
short ambiguous;  /* NZ when the mask is ambiguous */
char *mask;  /* Pointer to bytes at front of SCB */

/* Set pointer to characters at beginning of search control block */
mask = scb;

/* Ambiguous match on user number, matches only users 0 - 15, and not inactive entries */
if (mask[0] == '?')
    if (name[0] == 0x05)
        return NAME_NE;  /* Indicate inequality */
else  /* First char. of mask is not "?" */
    if (mask[0] != name[0])  /* User numbers do not match */
        return NAME_NE;  /* Indicate inequality */

/* No, check the name (and, if the length is such, the extent) */
for (count = 1;  /* Start with first name character */
    count <= scb->scb_length;  /* For all required characters */
    count++)  /* Move to next character */
    if (mask[count] == '?')  /* Wildcard character in mask */
```

Figure 11-1. (Continued)
Chapter 11: Additional Utility Programs

Figure 11.1 (Continued)

[Diagram of a flowchart or algorithm related to the text content]
1*=====================================================
# ./get_nfn
1*=====================================================

This function sets the FCB at "next_fname" to contain the directory entry found that matches the ambiguous file name in "amb_fname." On the first entry for a given file name, the most significant bit in the FCB's disk field must be set to one (this causes a search first BDOS call to be made). #/

/* Entry parameters */
struct _fcb *amb_fname;  /* Ambiguous file name */
struct _fcb *next_fname; /* First byte must have ms bit set for first time entry */

/* Exit parameters */
0 = No further name found
1 = Further name found (and set up in next_fname)


char bdos_func;  /* Set to either search first or next */
char *pfname;  /* Pointer to file name in directory entry */

/* Initialize tail-end of next file FCB to zero */
setmem(&next_fname->fcb_extent.FCBSIZE-12,0);

bdos_func = SEARCHF;  /* Assume a search first must be given */
if ((next_fname->fcb_disk & Ox80))  /* If not first time */
{
    /* search first on previous name */
    srch_file(next_fname,SEARCHF);
    bdos_func = SEARCHN;  /* Then do a search next */
}
else  /* First time */
{
    next_fname->fcb_disk &= Ox7F;  /* Reset first-time flag */
    /* Refresh next_file FCB from ambiguous file name */
    movmem(amb_fname.next_fname,12);
    /* If first time, issue search first, otherwise issue a search next call. "srch_file" returns a pointer to the directory entry that matches the ambiguous file name, or 0 if no match */
    if (!srch_file(next_fname,bdos_func))
    {
        return 0;  /* Indicate no match */
    }
    /* Move file name and type */
    movmem(pfname,next_fname->fcb_fname,11);
    return 1;  /* Indicate match found */
}

/*===================================================================*/
char *srch_file(fcb,bdos_code)  /* Search for file */
/*===================================================================*/

This function issues either a search first or search next BDOS call. #/

/* Entry Parameters */
struct _fcb *fcb;  /* pointer to file control block */
short bdos_code;  /* either SEARCHF or SEARCHN */

/* Exit parameters */
0 = no match found
NZ = pointer to entry matched (currently in buffer)

Figure 11-1. (Continued)
Chapter II: Additional Utility Programs

Figure 11-1. (Continued)

{ unsigned r_code;  /* Return code from search function */
  char *dir_entry; /* Pointer to directory entry */

  /* The BOS C compiler always sets the BDOS DMA to location 0x80 */
  if (r_code == 255)   /* No match found */
    return 0;

  /* Set a pointer to the matching entry by multiplying return code by 128 
   and adding onto the buffer address (0x80), also add 1 to point to first 
   character of name */
  return ((r_code << 5) + 0x80);

  */ End of srch_file */

/*====================================================================*/

rd_disk(drb);  /* Read disk (via BIOS) */
/*====================================================================*/

/* This function uses the parameters previously set up in the 
incoming request block, and, using the BIOS directly, 
exectues the disk read. */

/* Entry parameters */
struct _drb *drb;  /* Disk request block (disk, track, sector, buffer) */
/* Exit parameters */
0 = No data available
1 = Data available

{ if (!set_disk(drb))   /* Call SELDSK, SETTRK, SETSEC */
    return 0;        /* If SELDSK fails, indicate no data available */
  if (bios(DREAD))   /* Execute BIOS read */
    return 0;        /* Indicate no data available if error returned */

  return 1;        /* Indicate data available */
} /* End of rd_disk */

/*====================================================================*/

wrt_disk(drb);  /* Write disk (via BIOS) */
/*====================================================================*/

/* This function uses the parameters previously set up in the 
incoming request block, and, using the BIOS directly, 
exectues the disk write. */

/* Entry parameters */
struct _drb *drb;  /* Disk request block (disk, track, sector, buffer) */
/* Exit parameters */
0 = Error during write
1 = Data written OK

{ if (!set_disk(drb))   /* Call SELDSK, SETTRK, SETSEC, SETDMA */
    return 0;        /* If SELDSK fails, indicate no data written */
  if (bios(DWRITE))   /* Execute BIOS write */
    return 0;        /* Indicate error returned */

  return 1;        /* Indicate data written */
} /* End of wrt_disk */

/*=====..=====..=====..=====..=====..====..====..====..====..====..====..=====..=====..=====..=====..=====..=====..=====..=====..=====..=====..=====..=====..=====..=====..=====..=====..=====..=====..=====..=====..=====..=====..=====..=====..=====..=====..=====..=====..=====..=====..=====..=====..=====..=====..=====..=====..=====..=====..=====..=====..=====..=====..=====..=====..=====..=====..=====..=====..=====..=====..=====..=====..=====..=====..=====..=====..=====..=====..=====..=====..=====..=====..=====..=====..=====..=====..=====..=====..=====..=====..=====..=====..=====..=====..=====..=====..=====..=====..=====..=====..=====..=====..=====..=====..=====..=====..=====..=====..=====..=====..=====..=====..=====..=====..=====..=====..=====..=====..=====..=====..=====..=====..=====..=====..=====..=====..=====..=====..=====..=====..=====..=====..=====..=====..=====..=====..=====..=====..=====..=====..=====..=====..=====..=====..=====..=====..=====..=====..=====..=====..=====..=====..=====..=====..=====..=====..=====..=====..=====..=====..=====..=====..=====..=====..=====..=====..=====..=====..=====..=====..=====..=====..=====..=====..=====..=====..=====..=====..====.."
short set_disk(drb) /* Set disk parameters */
/* This function sets up the BIOS variables in anticipation of a subsequent disk read or write. */
/* Entry parameters */
struct _drb *drb;
/* Exit parameters */
0 = Invalid disk (do not perform read/write)
1 = BIOS now set up for read/write

{
    /* The sector in the disk request block contains a logical sector. If necessary (as determined by the value in the disk parameter header), this must be converted into the physical sector.
    NOTE: skewtab is declared as a pointer to a pointer to a short integer (single byte). */
    short **skewtab; /* Skewtab -> disk parameter header -> skew table */
    short phy_sec; /* Physical sector */

    /* Call the SELDISK BIOS entry point. If this returns a 0, then the disk is invalid. Otherwise, it returns a pointer to the pointer to the skew table */
    if ( ! (skewtab = biosh(SELDISK,drb -> dr_disk)))
        return 0; /* Invalid disk */
    bios(SETTRK,drb -> dr_track); /* Set track */

    /* Note that the biosh function puts the sector into registers BC, and a pointer to the skew table in registers HL. It returns the value in HL on exit from the BIOS */
    phy_sec = biosh(SECTRN,drb -> dr_sector,*skewtab); /* Get physical sector */
    bios(SETSEC,phy_sec); /* Set sector */
    bios(SETDMA,drb -> dr_buffer); /* Set buffer address */
    return 1; /* Indicate no problems */
} /* End of setp_disk */

/* Directory Management Functions */

get_nde(dir_pb) /* Get next directory entry */
/* This function returns a pointer to the next directory entry. If the directory has not been opened, it opens it. When necessary, the next directory sector is read in. */
/* If the current sector has been modified and needs to be written back onto the disk, this will be done before reading in the next sector. */
/* Entry parameters */
struct _dirpb *dir_pb;
/* Exit Parameters */
returns a pointer to the next directory entry in the buffer.
The directory open and write sector flags in the parameter block are reset as necessary.

{
    if(!dir_pb -> dp_open) /* Directory not yet opened */
        {
            if (open_dir(dir_pb)) /* Initialize and open directory */
            {
                err_dir(O_DIR,dir_pb); /* Report error on open */
                exit();
            }
        }
        /* Deliberately set the directory entry pointer to the end of the buffer to force a read of a directory sector */

Figure 11-1. (Continued)
Chapter 11: Additional Utility Programs

```c
/* Update the directory entry pointer to the next entry in
    the buffer. Check if the pointer is now "off the end"
    of the buffer and another sector needs to be read. */
if (++dir_pb -> dp_entry < dir_pb -> dp_buffer + DIR_BSZ)
    return dir_pb -> dp_entry; /* Return pointer to next entry */

/* Need to move to next sector and read it in */
/* Do not check if at end of directory or move to
   the next sector if the directory has just been
   opened (but the opened flag has not yet been set) */
if (!dir_pb -> dp_open)
    dir_pb -> dp_open = 1; /* Indicate that the directory is now open */
else
    if (dir_pb -> dp_write)
        { dir_pb -> dp_write = 0; /* Reset the flag */
          if (!rw_dir(W_DIR,dir_pb)) /* Write the directory sector */
              { err_dir(W_DIR,dir_pb); /* Report error on writing */
                exit();
              }
        }

    /* Count down on number of directory entries left to process,
       always four 32-byte entries per 128-byte sector */
    dir_pb -> dp_entries -= 4;
    /* Set directory-end flag true if number of entries now < 0 */
    if (dir_pb -> dp_entries == 0) /* now at end of directory */
        { dir_pb -> dp_end = 1; /* Indicate end */
          dir_pb -> dp_open = 0; /* Indicate directory now closed */
          return 0; /* Indicate no more entries */
        }

    /* Update sector (and if need be track and sector) */
    if (++dir_pb -> dp_sector == dir_pb -> dp_track)
        { ++dir_pb -> dp_track;
          dir_pb -> dp_sector = 0; /* Reset sector */
        }

    if (!rw_dir(R_DIR,dir_pb)) /* Read next directory sector */
        { err_dir(R_DIR,dir_pb); /* Report error on reading */
          exit();
        }

    /* Reset directory-entry pointer to first entry in buffer */
    return dir_pb -> dp_entry = dir_pb -> dp_buffer;
}

/*======================================.=======================*/
/* This function "opens" up the file directory
   on a specified disk for subsequent processing
   by rw_dir, next_dir functions. */
/* Entry parameters #/ struct _dirpb #dir_pb; /* Pointer to directory parameter block #/ */
```

Figure 11-1. (Continued)
/* Exit parameters
0 = Error, directory not opened
1 = Directory open for processing
*/

{ struct _dpb dpb;  // CP/M disk parameter block */

/* Get disk parameter block address for the disk specified in
the directory parameter block */
if ((dpb = get_dpb(dir_pb -> dp_disk)) == 0)
    return 0;  // Return indicating no DPB for this disk */

/* Set the remaining fields in the parameter block */
dir_pb -> dp_sptrk = dpb -> dpb_sptrk;  // Sectors per track */
dir_pb -> dp_track = dpb -> dpb_track;  // Track offset of the directory */
dir_pb -> dp_sector = 0;  // Beginning of directory */
dir_pb -> dp_nument = dpb -> dpb_maxden;  // No. of directory entries */
dir_pb -> dp_entrem = dir_pb -> dp_nument;  // Entries remaining to process */
dir_pb -> dp_end = 0;  // Indicate not at end */

/* Set number of allocation blocks per directory entry to
0 or 16 depending on the number of allocation blocks */
dir_pb -> dp_nabpde = (dpb -> dpb_maxabn > 255 ? 16 : 8);  // Set number of allocation blocks (one more than number of
highest block) */
dir_pb -> dp_nab = dpb -> dpb_maxabn;  // Set the allocation block size based on the block shift.
The possible values are: 3 = 1K, 4 = 2K, 5 = 4K, 6 = 8K, 7 = 16K.
So a value of 16 is shifted right by (7 - bshift) bits. */
dir_pb -> dp_absize = 16 >> (7 - dpb -> dpb_bshift);
return 1;  // Indicate that directory now opened */
}  // End of open_dir */

rw_dir(read_op,dir_pb)  // Read/write directory */

/* This function reads/writes the next 128-byte
sector from/to the currently open directory. */
/* Entry parameters */
short read_op;  // True to read, false (0) to write */
struct _dirpb *dir_pb;  // Directory parameter block */
/* Exit parameters */
0 = error -- operation not performed
1 = operation completed */

{ struct _drb drb;  // Disk request (for BIOS read/write) */
drb.dr_disk = dir_pb -> dp_disk;  // Set up disk request */
drb.dr_track = dir_pb -> dp_track;
drb.dr_sector = dir_pb -> dp_sector;
drb.dr_buffer = dir_pb -> dp_buffer;
if (read_op)
    { if (!rd_disk(&drb))  // Issue read command */
        return 0;  // Indicate error -- no data available */
    }
else
    { if (!wrt_disk(&drb))  // Issue write command */
        return 0;  // Indicate error -- no data written */
    }
return 1;  // Indicate operation complete */
}  // End of rd_dir */

Figure 11-1. (Continued)
Chapter 11: Additional Utility Programs

Chapter 11: Additional Utility Programs

### Chapter 11: Additional Utility Programs

#### Figure 11-1. (Continued)

```c
void err_dir(apeode, dir_pb)
/* Display directory error */
/* This function displays an error message to report an error */
/* detected in the directory management functions open_dir and rw_dir. */
/* Entry parameters */
short opcode; /* Operation being attempted */
struct _dirpb *dir_pb; /* Pointer to directory parameter block */
{
    printf("Error during ");
    switch(opcode)
    {
        case R_DIR:
            printf("Reading");
            break;
        case W_DIR:
            printf("Writing");
            break;
        case O_DIR:
            printf("Opening");
            break;
        default:
            printf("Unknown Operation (X) on", opcode);
    }
    printf(" Directory on disk Xc. ", dir_pb -> dp_disk + 'A');
} /* End of err_dir */

void setscb(scb, fname, user, extent, length)
/* Set search control block */
/* This function sets up a search control block according */
/* to the file name specified. The file name can take the */
/* following forms: */
/* filename */
/* filename.type */
/* dirfilename.type (meaning "all disks") */
/* ABCD(NO)filename.type (meaning "just the specified disks") */

The function sets the bit map according to which disks should be
searched. For each selected disk, it checks to see if an error is
generated when selecting the disk (i.e. if there are disk tables
in the BIOS for the disk). */
/* Entry parameters */
struct _scb *scb; /* Pointer to search control block */
char *fname; /* Pointer to the file name */
short user; /* User number to search for */
short extent; /* Extent number to search for */
int length; /* Number of bytes to compare */
/* Exit parameters */
None.
/* */
{
    int disk; /* Disk number currently being checked */
    unsigned adisks; /* Bit map for active disks */
    adisks = 0; /* Assume no disks to search */
    if (strcmp(fname,"")) /* Check if "" in file name */
        {
            if (fname == '/') /* Check if "all disks" */
                {
                adisks = 0xFFFF; /* Set all bits */
            }
            else /* Set specific disks */
                {
                while((fname != '/') /* Until "" reached */
```
[/* Build the bit map by getting the next disk id. (A - P), converting it to a number in the range 0 - 15, shifting a 1-bit left that many places, and OR-ing it into the current active disks. */
  adisks = 1 << (toupper(Mfname) - 'A');
  ++fname; /* Move to next character */
  ++fname; /* Bypass colon */
}
else /* Use only current default disk */
  /* Set just the bit corresponding to the current disk */
  adisks = 1 << bdos(GETDISK);
}
setfcb(scb,fname); /* Set search control block as though it were a file control block. */
/* Make calls to the BIOS SELDSK routine to make sure that all of the active disk drives have disk tables for them in the BIOS. If they don't, turn off the corresponding bits in the bit map. */
for (disk = 0; /* Start with disk A */
  disk < 16; /* Until disk P */
  disk++) /* Use next disk */
  if (!((1 << disk) & adisks)) /* Avoid selecting unspecified disks */
    if (biosh(SELDISK.disk) == 0) /* Make BIOS SELDSK call */
      /* Turn OFF corresponding bit in mask by AND-ing it with bit mask having all the other bits set = 1 */
      adisks &= ((1 << disk) * OxFFFF);
}
scb -&gt; scb_adisks = adisks; /* Set bit map in SCB */
scb -&gt; scb_userno = user; /* Set user number */
scb -&gt; scb_extent = extent; /* Set extent number */
scb -&gt; scb_length = length; /* Set number of bytes to compare */
} /* End setscb */
/*=============================================*/
dm_clr(disk_map) /* Disk map clear (to zeros) */
/*=============================================*/
/* This function clears all elements of the disk map to zero. */
/* Entry Parameters */
unsigned disk_map[16][18]; /* Address of array of unsigned integers */
/* Exit parameters */
None.
/*
  /* WARNING -- The 576 in the setmem call below is based on the disk map array being [16][18] -- i.e. 288 unsigned integers, hence 576 bytes. */
  setmem(disk_map,576,'\0'); /* Fill array with zeros */
} /* End of dm_clr */
/*=============================================*/
dm_disp(disk_map,adisks) /* Disk map display */
/*=============================================*/
/* This function displays the elements of the disk map, showing the count in each element. A zero value-element is shown as blanks. For example:
Figure 11-1. (Continued)*/
Chapter II: Additional Utility Programs

Figure 11-1. (Continued)
Use BOOS SETDISK function

1* Most programs that interact with a user must accept parameters from the user by name and translate the name into some internal code value. They also must be able to work in reverse, examining the setting of a variable, and determining what (ASCII name) it has been set to.

An example is setting baud rates. The user may want to enter "19200," and have this translated into a number parameter output to a chip. Alternatively, a previously set baud rate variable may have to be examined and the string "19200" generated to display its current setting to the user.

A code table is used to make this task easier. Each element in the table logically consists of:

- A code value (unsigned integer)
- An ASCII character string (actually a pointer to it)

```c
ct_init(entry, code, string)
```

This function initializes a specific entry in a code table with a code value and string pointer.

```c
t_parcl(table, string)
```

This function searches the specified table for a matching string, and returns the code value that corresponds to it. If only one match is found in the table, then this function returns that code value. If no match or more than one match is found, it returns the error value, CT_SNF (string not found).

This function is specifically designed for processing parameters on a command line. Note that the comparison is done after conversion to uppercase (i.e., "STRING" matches "string"). A substring compare is used so that only the minimum number of characters for an unambiguous response need be entered. For example, if the table contained:

<table>
<thead>
<tr>
<th>Code Value</th>
<th>string</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 &quot;APPLES&quot;</td>
<td></td>
</tr>
<tr>
<td>2 &quot;ORANGES&quot;</td>
<td></td>
</tr>
<tr>
<td>3 &quot;APRICOTS&quot;</td>
<td></td>
</tr>
</tbody>
</table>

A response of "0" would return code = 2, but "A" or "AP" would be ambiguous. "APR" or "APP" would be required.

```c
struct _ct *table;
char *string;
```
int mcode; /* Matched code to return */
int mcount; /* Count of number of matches found */
mcode = CT_SNFI; /* Assume error */
mcount = 0; /* Reset match count */

while(table -> _ct_code != CT_SNFI) /* Not at end of table */
{ /* Compare keyboard response to table entry using uppercase substring compare */
  if (ustrencmp(table -> _ct_sp, string) == 0) /* Update match count */
  { mcount++; /* Save code */
    mcode = table -> _ct_code; /* Move to next entry */
    table++; /* Only one match found */
  }
  else /* Illegal or ambiguous */
    return CT_SNFI;
} /* End ct_parc */

unsigned ct_code(table, string) /* Return code for string */
{ /* This function searches the specified table for the specified string. If a match occurs, it returns the corresponding code value. Otherwise it returns CT_SNFI (string not found). Unlike ct_parc, this function compares every character in the key string, and will return the code on the first match found. */
  struct _ct *table; /* Pointer to table */
  char *string; /* Pointer to string */

  /* Entry parameters */
  Code value -- if string found
  CT_SNFI -- if string not found */

  while(table -> _ct_code != CT_SNFI) /* For all entries in table */
  { /* Compare strings */
    if (ustrencmp(table -> _ct_sp, string) == 0) /* Return code */
      return table -> _ct_code; /* Move to next entry */
    table++; /* String not found */
  } /* End ct_code */

ct_disp(table) /* Displays all strings in specified table */
{ /* This function displays all of the strings in a given table. It is used to indicate valid responses for operator input. */
  struct _ct *table; /* Pointer to table */

  /* Entry Parameters */
  /* Exit Parameters */
  None.

  while(table -> _ct_code != CT_SNFI) /* Not end of table */
  { /* Print string */
    printf("
		Xs",table -> _ct_sp); /* Move to next entry */
    table++;
  }

Figure 11-4. (Continued)
```c
putchar("\n");  /* Add final return */

}  /* End of ct_disps */

/*===========================================================================*/
ct_index(table,string)  /* Returns index for a given string */
/*===========================================================================*/
/* This function searches the specified table, and returns 
the INDEX of the entry containing a matching string.
All characters of the string are used for the comparison, 
after they have been made uppercase. */

/* Entry parameters */
struct _ct *table;  /* Pointer to table */
char *string;  /* Pointer to string */

/* Exit parameters */
Index of entry matching string, or 
CT_SNF if string not found.
*/

{ int index;  /* Current value of index */
  index = 0;  /* Initialize index */
  while(table -> _ct_code != CT_SNF)  /* Not at end of table */
    { if (strcmp(table -> _ct_sp,string) == 0) /* Return index */
      { table++;  /* Move to next table entry */
        index++;  /* Update index */
      } return CT_SNF;  /* String not found */
    }
}

/*===========================================================================*/
char *ct_str(table,index)  /* Get string according to index */
/*===========================================================================*/
/* This function returns a pointer to the string in the 
table entry specified by the index. */

/* Entry parameters */
struct _ct *table;  /* Pointer to table */
int index;  /* Index into table */

{ struct _ct *entry;  /* Entry pointer */
  entry = table[index];  /* Point to entry */
  return entry -> _ct_sp;  /* Return pointer to string */
}  /* End of ct_str */

/*===========================================================================*/
char *ct_strc(table,code)  /* Get string according to code value */
/*===========================================================================*/
/* This function searches the specified table and returns a 
pointer to the character string in the entry with the 
matching code value or a pointer to a string of "unknown" 
if the code value is not found. */

/* Entry parameters */
struct _ct *table;  /* Pointer to table */
unsigned code;  /* Code value */

{ while(table -> _ct_code != CT_SNF)  /* Until end of table */
  { if (table -> _ct_code == code)  /* Check code matches */
    { return table -> _ct_sp;  /* Yes, return ptr. to str. */
      table++;  /* No, move to next entry */
    }
  }
}
```

Figure 11-1. (Continued)
Bit vector functions

These functions manipulate bit vectors. A bit vector is a group of adjacent bits, packed eight per byte. Each bit vector has the structure defined in the LIBRARY.H file.

Bit vectors are used primarily to manipulate the operating system's allocation vectors and other values that can best be represented as a series of bits.

---

Figure 11-1. (Continued)
Figure 11-1. (Continued)
Chapter 11: Additional Utility Programs

Figure 11-1. (Continued)
Associated with the library of functions is another section of source code called "LIBRARY.H", shown in Figure 11-2. This "header" file must be included at the beginning of each program that calls any of the library functions.

For reasons of clarity, this chapter describes the simplest functions first, followed by the more complex, and finally by the utility programs that use the functions.

Several functions in the library and some definitions in the library header are not used by the utilities shown in this chapter. They have been included to illustrate techniques and because they might be useful in other utilities you could write.

```
#define LIBVN "1.0"     /* Library version number */

/* This file contains groups of useful definitions. It should be included at the beginning of any program that uses the functions in LIBRARY.C */

/* Definition to make minor language modification to C. */
#define short char     /* Short is not supported directly */
```

Figure 11-2. LIBRARY.H, code to be included at the beginning of any program that calls LIBRARY functions in Figure 11-1
Chapter II: Additional Utility Programs

One of the functions (by_make) in the library uses the BDS C function alloc, to allocate memory. The following definitions are provided for alloc: */

```c
struct _header /* Header for block of memory allocated */
  { struct _header *next; /* Pointer to the next header in the chain */
    unsigned size; /* Number of bytes in the allocated block */
  };
struct _header_base; /* Declare the first header of the chain */
struct _header = alloc; /* Used by alloc() and free() functions */
```

/* BDS function call numbers */
#define SETDISK 14 /* Set (select) disk */
#define SEARCH 17 /* Search first */
#define SEARCHN 18 /* Search next */
#define DELETEF 19 /* Delete file */
#define GETDISK 14 /* Get default disk (currently logged in) */
#define SETDMA 26 /* Set DMA (Read/Write) Address */
#define SETDPARM 31 /* Set disk parameter block address */
#define GETUSER 32 /* Get current user number */
#define SETUSER 32 /* Set current user number */

/* Direct BIOS calls */
These definitions are for direct calls to the BIOS. Using these makes program less transportable.
Each symbol is related to its corresponding jump in the BIOS jump vector.
The more useful entries are defined. */

#define CONST 2 /* Console status */
#define CONIN 3 /* Console input */
#define CONOUT 4 /* Console output */
#define LIST 5 /* List output */
#define AUXOUT 6 /* Auxiliary output */
#define AUXIN 7 /* Auxiliary input */
#define HOME 8 /* Home disk */
#define SELDISK 9 /* Select logical disk */
#define SETTRK 10 /* Set track */
#define SETSEC 11 /* Set sector */
#define SETDMA 12 /* Set DMA address */
#define DREAD 13 /* Disk read */
#define DWRITE 14 /* Disk write */
#define LISTST 15 /* List status */
#define SECTRN 16 /* Sector translate */
#define AUXIST 17 /* Auxiliary input status */
#define AUXOST 18 /* Auxiliary output status */
#define CIOINIT 19 /* Specific character I/O initialization */
#define SETDOG 20 /* Set watchdog timer */
#define DEV_INIT 19 /* Configuration block, get address */

/* Definitions for accessing the configuration block */
#define CB_GET 21 /* BIOS jump number to access routine */
#define DEV_INIT 19 /* BIOS jump to initialize device */
#define CB_DATE 2 /* Date in ASCII */
#define CB_TIMEA 1 /* Time in ASCII */
#define CB_DPMIS 2 /* Date, time flags */
#define TIME_SET 0x01 /* This bit NZ means date has been set */
#define DATE_SET 0x02 /* This bit NZ means time has been set */
#define CB_FIP 3 /* Forced input pointer */
#define CB_CI 5 /* Console input */
#define CB_CO 6 /* Console output */
#define CB_AI 7 /* Auxiliary input */
#define CB_AO 8 /* Auxiliary output */

Figure 11-2. (Continued)
Figure 11-2. (Continued)
Chapter II: Additional Utility Programs

```c
#define ALLPROTO 0xDC
/* All protocols combined */

struct _dt
{char dt_f[14];  // Filler #
 char dt Atl;  // Status byte 1 -- has protocol flags #
 char dt_at2;  // Status byte 2 #
 unsigned dt_f2;  // Filler #
 unsigned dt_etxml;  // ETX/ACK message length #
 char dt_F3C12J;  // Filler #
}

/* Values returned by the comp_fname (compare file name) */
#define NAME_EQ 0  // Names equal #
#define NAME_LT 1  // Name less than mask #
#define NAME_GT 2  // Name greater than mask #
#define NAME_NE 3  // Name not equal (and comparison ambiguous) #

/* Define for standard CP/M file control block */
#define FCBSIZE 36  // Define the overall length of an FCB #

struct _fcb
{short fcb_disk;  // Logical disk (0 = default) #
 char fcb_fnam[11];  // File name, type (with attributes) #
 short fcb_extent;  // Current extent #
 unsigned fcb_size;  // Reserved for CP/M #
 short fcb_recent;  // Record count used in current extent #
 union  // Allocation blocks can be either 
   {short fcbab_short[16];  // Single or double bytes #
    unsigned fcbab_long[8];
   } fcbab;
 short fcb_current;  // Current record within extent #
 char fcb_cursrec[3];  // Record for random read/write #
}

/* Parameter block used for calls to the directory management routines */
#define DIR_BSI 128  // Directory buffer size #

struct _dirpb
{short dp_open;  // 0 to request directory to be opened #
 short dp_end;  // NZ when at end of directory #
 short dp_write;  // NZ to write current sector to disk #
 struct _dir dp_entry;  // Pointer to directory entry in buffer #
 char dp_buffer [DIR_BSI] ;  // Directory sector buffer #
 char dp_disk;  // Current logical disk #
 int dp_track;  // Start track #
 int dp_sector;  // Start sector #
 int dp_numents;  // Number of directory entries #
 int dp_entsize;  // Entries remaining to process #
 int dp_sptrk;  // Number of sectors per track #
 int dp_nab;  // Number of allocation blocks per dir. entry #
 unsigned dp_nabd;  // Number of allocation blocks #
 int dp_absize;  // Allocation block size (in Kbytes) #
}

/* The err_dir function is used to report errors found by the 
directory management routines, open_dir and rw_dir. 
Err_dir needs a parameter to define the operation being 
performed when the error occurred. The following definitions 
represent the operations possible. */
#define W_DIR 0  // Writing directory #
#define R_DIR 1  // Reading directory #
#define O_DIR 2  // Opening directory #
```

Figure 11-2. (Continued)
The CP/M Programmer's Handbook

Figure 11-2. (Continued)
Figure 11-2. (Continued)

Library Functions

This section describes the library functions and the sections from the header file that must be included at the beginning of each utility program.

A Minor Change to C Language

One minor problem with the BDS C Compiler is that it does not support "short" integers, or integers that are only a single byte long. It is convenient to declare certain values as short to serve as a reminder of the standard type definition. Therefore, the BDS C compiler must be "fooled" by declaring these values to be single characters. To do this, the library header file contains the declaration

```c
#define short char.
```

shown in Figure 11-2, section a.

The "#define" tells the first part of the C compiler, the preprocessor, to substitute the string "char" (which declares a character variable) whenever it encounters the string "short" (which would ordinarily declare a short integer in standard C).

Note that character strings enclosed in "/*" and "*/" are regarded as comments and are ignored by the compiler.

BDOS Calls

The standard library of functions that comes with the BDS C compiler includes a function to make BDOS calls, called "bdos." It takes two parameters, and a typical call is of the following form:

```c
bdos(c,de);
```

The "c" parameter represents the value that will be placed into the C register. This is the BDOS function code number. The "de" is the value that will be placed in the DE register pair.
The library header contains definitions (#define declarations) for BDOS functions 14 through 32, making these functions easier to use (Figure 11-2, c). Function 32 (Get/Set Current User Number) has two definitions; the “de” parameter is used to differentiate whether a get or a set function is to be performed.

BIOS Calls

The BDS C standard library also contains two functions that make direct BIOS calls. These are “bios” and “biosh.” They differ only in that the bios function returns the value in the A register on return from the BIOS routine, whereas biosh, as its name implies, returns the value in the HL register pair. Examples of their use are

```c
bios(jump_number,bc);
```

and

```c
biosh(jump_number,bc,de);
```

Both functions take as their first parameter the number of the jump instruction in the BIOS jump vector to which control is to be transferred. For example, the console-status entry point is the third JMP in the vector. Numbering from 0, this would be jump number 2.

The library header file contains #defines for BIOS jumps 2 through 21 (Figure 11-2, d). The last group of these #defines (19 through 21) is for the “private” additions to the standard BIOS jump vectors described in Chapter 8.

Remember, though, that using direct BIOS calls makes programs more difficult to move from one system to another.

BIOS Configuration Block Access

As you may recall, the configuration block is a collection of data structures in the BIOS. These structures are used either to store the current settings of certain user-selectable options, or to point to other important data structures in the BIOS.

One of the “private” jumps appended to the standard BIOS jump vector transfers control to a routine that returns the address in memory of a specified data structure. For example, if a utility program needs to locate the word in the BIOS that determines from which physical device the console input is to read, it can transfer control to jump 21 in the BIOS jump vector (actually the 22nd jump) with a code value of 5 in the C register. This jump transfers control to the CBS$Get$Address code, which on its return will set HL to the address of the console input redirection vector. The utility program can then read from or write into this variable. The library header file contains #define declarations relating the code values to mnemonic names (Figure 11-2, e).

You will need to refer to the source code in Figure 8-10 to determine whether the address returned by the BIOS function is the address of the data element or the
address of a higher-level table that in turn points to the data element.

In order to access the current system date, for example, you would include the following code:

```c
char *ptr_to_date;  /* declare date pointer */
ptr_to_date = biosh(CB_DATE);  /* get address */
```

The ptr_to_date can then be used to access the date directly.

During initial debugging of a utility, it is useful to be able to intercept all such accesses to the configuration block, partly to reassure yourself that the utility program is working as it should, and partly to ensure that the BIOS routine is returning the correct addresses to the data structures. Therefore, the utility library contains a function, "get_cba," that gets a configuration block address (Figure 11-1, a).

At first, it appears that get_cba is declared as a function that returns a pointer to characters. This is not strictly true. Sometimes the address it returns will point to characters, sometimes to integers, and sometimes to structures (such as the function key table).

The "printf" instruction has been left in the function in anticipation of debugging a utility. If you need to see some debug output whenever the get_cba function is used, delete the "/*" and "*/" surrounding the "printf" and recompile the library.

**BIOS Function Key Table Access**

The BIOS shown in Figure 8-10 contains code to recognize when an incoming escape sequence indicates that one of the terminal's function keys has been pressed. Instead of returning just the escape sequence, the console driver injects a previously programmed string of characters into the console input stream. For example, on a DEC VT-100 terminal, when the F1 function key is pressed, the terminal emits the following character sequence: ESCAPE, "O", "P". The function key table contains the "OP" and a 00H-byte-terminated string of characters to be injected into the console input stream. In Figure 8-10, the example string is "FUNCTION KEY 1", LINE FEED. The library header file contains a declaration for the structure of the function key table (Figure 11-2, h).

Note the use of "#define" to declare the length of the incoming characters emitted by the terminal as well as the length of the output string.

In order to access a function key table entry, you must declare a pointer to a "_fkt" structure like this:

```c
struct _fkt *ptr_to_fkt;  /* Declare pointer */
ptr_to_fkt = get_cba(CB_FKT);  /* Set pointer */
printf("Display the first string %s",
       ptr_to_fkt -> fk_output);
++ptr_to_fkt;  /* Move to next entry */
```

The get_cba function is used to return the address of the first entry in the function key table and set a pointer to it. Then the printf function (part of the
The CP/M Programmer's Handbook

standard BDS C library) is used to print out the first string, which gets substituted for the "%s" in the quoted string. Note that the statement

```
++ptr_to_fkt
```

does not just add one to the pointer to the function key table—it adds whatever it takes to move the pointer to the next *entry* in the table.

**BIOS Device Table Access**

The device tables are important structures for the serial devices served by the console, auxiliary, and list device drivers in the BIOS. They are declared at line 1500 in Figure 8-10.

The get_cba function does not return a pointer to a specific device table, but a pointer to a table of device table addresses. Each entry in the address table corresponds to a specific device number. If there is no device table for a specific device number, then the corresponding entry in the table will be set to zero. the library header file contains definitions for the device table (Figure 11-2, i).

The device tables contain, among other things, the current serial line protocols used to synchronize the transmission and reception of data by the device drivers and the physical devices. An example utility, PROTOCOL, is shown later in the chapter. The example #define declarations and structure definition shown here are modeled on the requirements of this utility. The only relevant bytes are the two status bytes dLstl and dLst2 and the message length used with the ETX/ACK protocol, dL_etxml. The #defines shown are for the specific bits in the device table's status bytes. The PROTOCOL utility uses the most significant bit to indicate whether a given protocol setting can coexist with others.

To access these fields, use the following code:

```
struct _ppdt
{
    char pdt[16]; /* Array of 16 pointers to device tables */
} *ppdt;
/* Pointer to array of 16 pointers */
struct _dt *dt; /* Pointer to device table */

ppdt = get_cba(CB_DTA); /* Set pointer to array of pointers */
dt = ppdt -> pdt[device_no]; /* Set pointer to specified device table */

if (!dt)
    printf("\nError - no device table for this device.");

dt -> dt_etxml = 0; /* Clear ETX message length */
```

**BIOS Disk Parameter Block Access**

Several of the utility programs shown in this chapter must access the file directory on a given logical disk. The disk parameter block (DPB) indicates the size and location of the file directory. The library header contains a structure definition that describes the DPB (Figure 11-2, n).
To locate the DPB, you can make a direct BIOS call to the SELDSK routine, which returns the address of the disk parameter header (DPH). You then can access the DPB pointer in the DPH. Alternatively, using the BDOS, you can make the required disk the default disk and then request the address of its DPB. The code for the latter method is shown in the `get_dpb` function included in the utility library (Figure 11-1, u).

The `get_dpb` function uses a BIOS SELDSK function first to see if the specified disk is legitimate. Only then does it use the BDOS.

**Reading or Writing a Disk Using the BIOS**

When you write a program that uses direct BIOS calls, you increase the possibility of problems in moving the program from one system to another. However, in certain circumstances it is necessary to use the BIOS. Reading and writing the file directory is one of these; the BDOS cannot be used to access the directory directly. The library header contains a structure declaration for a parameter block that contains the details of an "absolute" disk read or write (Figure 11-2, p).

Note the pointer to the 128-byte data buffer used to hold one of CP/M’s "records."

The disk read and write functions are `rd_disk` (Figure 11-1, k) and `wrt_disk` (Figure 11-1, l). Both of them take a `_drb` as an input parameter, and both call the `set_disk` function to make the individual BIOS calls to SELDSK, SETTRK, and SETSEC.

Of special note is the code in `set_disk` (Figure 11-1, m) that converts a logical sector into a physical sector using the sector translation table and the SECTRAN entry point in the BIOS.

**File Directory Entry Access**

All of the utility programs that access a disk directory share the same basic logic regardless of their specific task. This logic can be described best in pseudocode:

```plaintext
while (not at the end of the directory)
{
    access the next directory entry
    if (this entry matches the current search criteria)
    {
        process the entry
    }
}
```

There are two ways of implementing this logic. The first uses the BIOS to read the directory. Entries are presented to the utility exactly as they occur in the file
directory. The second uses the BDOS functions Search First and Search Next and accesses the directory file-by-file rather than by entry. This latter method is more suited to utilities that process files rather than entries. The ERASE utility, described later in this chapter, illustrates this second method.

Three groups of functions are provided in the library: to access the next entry in the directory, to match the name in the current entry against a search key, and to assist with processing the directory.

**Directory Accessing Functions**

A number of functions involve access to the file directory. The first group of such functions performs the following:

- **get...nde** (get next directory entry; Figure 11-1, n)
  
  This function returns a pointer to the next directory entry, or returns zero if the end of the directory has been reached.

- **open...dir** (open directory; Figure 11-1, o)
  
  This function is called by get...nde to open up a directory for processing.

- **rw...dir** (read/write directory; Figure 11-1, p)
  
  This function reads or writes the current directory sector.

- **err...dir** (error on directory; Figure 11-1, q)
  
  This general-purpose routine displays an error message if the BIOS indicates that it had problems either reading or writing the directory.

All of these functions use a directory parameter block to coordinate their activity. The library header contains the definitions for this structure (Figure 11-2, 1), as well as #define declarations for operation codes used by the directory-accessing functions (Figure 11-2, m).

Before calling get...nde, the calling program needs to set dp...open to zero (forcing a call to open...dir) and the dp...disk field to the correct logical disk. The open...dir function sets up all of the remaining fields, using get...dpb to access the disk parameter block for the disk specified in dp...disk.

Of the remaining flags, dp...end will be set to true, when the end of the directory is reached, and dp...write must be nonzero for rw...dir to write the current sector back onto the disk.

The get...nde function includes all of the necessary logic to move from one directory entry to the next, reading in the next sector when necessary, and writing out the previous sector if the dp...write flag has been set to a nonzero value by the calling program. It also counts down on the number of directory entries processed, detecting and indicating the end of the directory.

The code at the beginning of the function calls open...dir if the dp...open flag is false. Note the code at the end of open...dir that sets the number of allocation blocks per directory entry (dp...nabpde). This number is computed from the maximum
allocation block number in the disk parameter block. If it is larger than 255, each allocation block must occupy a word, and there will be eight blocks per directory entry. If there are 255 or fewer allocation blocks, each will be one byte long and there will be 16 per entry. The allocation block size, in Kbytes, is computed from a simple formula.

In the early stages of debugging utilities, comment out the line that makes the call to wrLdisk. This will prevent the directory from being overwritten. You then can test even those utilities that attempt to erase entries from the directory without any risk of damaging any data on the disk.

The last function in this group, err_dir, is a common error handling function for taking care of errors while reading or writing the directory.

**Directory Matching Functions**

The second group of functions that access the file directory matches each directory entry against specific search criteria. These include the following functions:

- **setscb** (set search control block; Figure 11-1, r)
  A search control block (SCB) is a structure that defines the entries in the directory that are to be selected for processing.

- **comp fname** (compare file name; Figure 11-1, f)
  This function compares the file name in the current directory entry with the one specified in the search control block.

The library header contains the structure definition for the search control block (Figure 11-2, q). This SCB is a hybrid structure. The first part of it is a cross between a file control block (FCB) and a directory entry. The last three fields, scb_length, scb_disk, and scb_adisks, are peculiar to the search control block. Note that its overall length is the same as an FCB's so that the standard BDS C function set_fcb can be used. This function sets the file name and type into an FCB, replacing "*" with as many "?" characters as are required, and clears all unused bytes to zero.

The scb_length field indicates to the comp fname function how many bytes of the structure are to be compared. This field will be set to 12 to compare the user number, file name, and type, or to 13 to include the extent number.

Note that scb_disk is the current disk to be searched, whereas scb_adisks is a bit map with a 1 bit corresponding to each of the 16 possible logical disks that must be searched.

The search control block is initialized by the setscb function.

Note the form of the file name that setscb expects to receive. This is described in the comments at the beginning of the function.

Several of the utility programs use their own special versions of setscb,
renaming it ssetscb (special setscb) to avoid the library version being linked into the programs.

The complementary function comp fname is used to compare the first few bytes of the current directory entry to the corresponding bytes of the SCB.

The comp fname function performs a specialized string match of the user number, the file name, the file type, and, optionally, the extent number. A “?” character in the search control block file name, type, and extent will match with any character in the file directory entry. However, in the SCB user number, a “?” will only match a number in the range 0 to 15; it will not match a directory entry that has the user number byte set to E5H (or 0xE5, as hexadecimal notation in C).

This function also returns one of several values to indicate the result of the comparison. These values are defined in the library header file (Figure 11-2, j).

Directory Processing Functions

The final group of functions that access the directory are those that help process the directory entries themselves. These functions use a structure definition to access each directory entry (Figure 11-2, o).

A union statement is used for the allocation block numbers. These can be single- or two-byte entries, depending on the maximum number of allocation blocks that must be represented. The union statement tells the BDS C compiler whether there will be a 16-byte array of short integers (characters) or an array of eight unsigned two-byte integers.

The functions contained in this group can be divided into three subgroups:

- Those that deal with converting directory entries for display on the console.
- Those that deal with a “disk map”—a convenient array for representing logical disks and the user numbers they contain.
- Those that deal with “bit vectors”—a convenient representation of which allocation blocks on a logical disk are in use or available.

The library contains only one function to convert a directory-entry file name into a suitable form for display on the console. This is the conv_dfname function (Figure 11-1, h). It takes the information from the specified directory entry (or, as a convenience, a search control block) and formats it into a string of the form

```
uu/d:filename.typ
```

The “uu” specifies the user number and the “d” specifies the disk identification.

The repetitive code at the end of the function is necessary to make sure that the characters in the file type do not have their high-order bits set. These bits are the file attributes. If they are set, they can render the characters nondisplayable on some terminals.
Chapter 11: Additional Utility Programs

The second subgroup of functions, those that manipulate a “disk map,” produce an array that looks like this:

```
Disks
↓
User Numbers --> -Totals-
A 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 Used Free
B :
C :
P
```

This disk map is used by several utility programs. For example, the SPACE utility displays a disk map that shows, for each logical disk in the system, and for each user on each logical disk, how many Kbytes of disk space are in use. The totals at the right show the total of used and free space. In another example, the FIND utility shows how many files on each disk and in each user number match the search name.

Each utility program that uses a disk map is coded:

```
unsigned disk_map[16][18];
```

Two functions are provided in the library to deal with the disk map:

- `dm_clr (disk map clear; Figure 11-1, s)`
  - This function fills the entire disk map with zeros.

- `dm_disp (disk map display; Figure 11-1, t)`
  - This function displays the horizontal and vertical caption lines for the disk map and then converts each element of the disk map to a decimal number.

The first function, `dm_clr`, uses one of the standard BDS C functions to set a block of memory to a specific value. It presumes that the disk map is 16 X 18 elements, each two bytes long.

The second function, `dm_disp`, prints horizontal lines only for those disks specified in the bit map parameter. Here is an example of its output:

```
  0 1 2 3 4 ... 10 11 12 13 14 15 Used Free
A:  1 1 15 241
B:  66 20 74 50 3
C: -- None -- 245 779
D: -- None -- 0 1024

(NOTE: All user groups would be shown on the terminal.)
```

The final subgroup deals with processing “bit vectors.” A bit vector is a string of bits packed eight bits per byte. Each bit is addressed by its relative number along the vector; the first bit is number 0.

An example of why bit vectors are used is a utility program that needs to scan the directory of a disk and build a structure showing which allocation blocks are in use. It can do this by accessing each active directory element and, for each nonzero allocation block number, setting the corresponding bit number in a bit vector.

The library header has a structure definition for a bit vector (Figure 11-2, s).
This vector contains the overall length of the bit vector in bytes, and two pointers. The first points to the start of the vector, the second to the end. The bytes that contain the vector bits themselves are allocated by the alloc function—one of the standard BDS C functions.

The following bit vector functions are provided in the library:

- bv_make (bit vector make; Figure 11-1, cc)
  This function allocates memory for the bit vector (using the standard mechanism provided by BDS C) and sets all of the bits to zero.

- bv_fill (bit vector fill; Figure 11-1, dd)
  This fills a specified vector, setting each byte to a specified value.

- bv_set (bit vector set; Figure 11-1, ee)
  This sets the specified bit of a vector to one.

- bv_test (bit vector test; Figure 11-1, ff)
  This function returns a value of zero or one, reflecting the setting of the specified bit in a bit vector.

- bv_nz (bit vector nonzero; Figure 11-1, gg)
  This returns zero or a nonzero value to reflect whether any bits are set in the specified bit vector.

- bv_and (bit vector AND; Figure 11-1, hh)
  This function performs a Boolean AND between two bit vectors and places the result into a third vector.

- bv_or (bit vector OR; Figure 11-1, ii)
  This is similar to bv_and, except that it performs an inclusive OR on the two input vectors.

- bv_disp (bit vector display; Figure 11-1, jj)
  This function displays a caption line and then prints out the contents of the specified bit vector as a series of zeros and ones. Each byte is formatted to make the output easier to read.

The bv_make function uses the alloc function to allocate a block from the unused part of memory between the end of a program and the base of the BDOS. It requires that two data structures be declared at the beginning of the program. These structures are declared in the library header file (Figure 11-2, b).

The bv_fill function uses the standard BDS C setmem function.

The bv_set function converts the bit number into a byte offset by shifting the bit number right three places. The least significant three bits of the original bit number specify which bit in the appropriate byte needs to be ORed in.

The bv_test function is effectively the reverse of bv_set. It accesses the specified bit and returns its value to the calling program.

The bv_nz function scans the entire bit vector looking for the first nonzero
byte. If the entire vector is zero, it returns a value of zero. Otherwise, it returns a pointer to the first nonzero byte.

Both bv_and and bv_or functions take three bit vectors as parameters. The first vector is used to hold the result of either ANDing or ORing the second and third vectors together. Both of these functions assume that the output vector has already been created using bv_make. The shortest of the three vectors will terminate the bv_and or bv_or function; that is, these functions will terminate when they reach the end of the first (shortest) vector.

The final function, bv_disp, displays the title line specified by the calling program, and then displays all of the bits in the vector, with the bit number of the first bit on each line shown on the left.

None of the utility programs uses bv_disp—it has been left in the library purely as an aid to debugging.

Here is an example of bv Disp's output:

Bit Vector: Allocation Blocks in Use
0  : 0000 0000 0001 1000 1000 0001 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
40 : 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
80 : 1100 0000 1111 1100 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
120: 1110 1100 0001 1111 0000 0000 1001 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
160: 1111 0010 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
200: 1111 0010

Checking User-Specified Parameters

The C language provides a mechanism for accessing the parameters specified in the "command tail." It provides a count of the number of parameters entered, "argc" (argument count), and an array of pointers to each of the character strings, "argv" (argument vector). At the beginning of the main function of each program you must define these two variables like this:

```c
main(argc, argv)
{
  int argc;  /* Argument count */
  char *argv[]; /* Array of pointers to char. strings */
  ; /* Remainder of main function */
}
```

Consider the minimum case—a command line with just the program name on it:

A>command

The convention is that the first argument on the line is the name of the program itself. Hence argc would be set to one, and argv[0] would be a pointer to the program name, "command."
Next consider a more complex case—a command line with parameters like the following:

`A>command param1 123`  

In this case, argc will be three; argv[1] will be a pointer to param1; and argv[1][0] will access the 0 (the first) character of argv[1]—in this case the character "p."

To detect whether the second parameter is present and numeric, the code will be:

```c
if (isdigit(argv[1][0]))
{
    /* Process digit */
}
else
{
    /* Parameter either not present or has alpha character at the front */
}
```

In most of the utilities, you will get a much "friendlier" program if the user need only specify enough characters of a parameter to distinguish the value entered from the other possible values. For example, consider a program that can have as a parameter one of the following values: 300, 600, 1200, 2400, 4800, 9600, or 19200. It would be convenient if the user needed to type only the first digit, rather than having to enter redundant keystrokes. However, the values 1200 and 19200 would then be ambiguous. The user would have to enter 12 or 19. Novice users often prefer to specify the entire parameter for clarity and security.

The standard C library provides a character string comparison function, strcmp. Unfortunately, this function does not provide for the partial matching just described. Therefore, the library includes two special functions that do make this possible: ststrcmp (substring compare, Figure 11-1, d) and uststrcmp (uppercase substring compare, Figure 11-1, e). The latter function is necessary when you need to compare a substring that could contain lowercase characters; it converts characters to uppercase before the comparison.

To assist with character string manipulation, two additional functions have been included in the library. These are strscn (string scan, Figure 11-1, b) and ustcmp (uppercase string compare, Figure 11-1, c).

### Using Code Tables

A code table is a simple structure used by all of the utility programs that accept parameters that can have any of several values. The library header contains a structure definition for a code table (Figure 11-2, r).

A code table entry contains an unsigned code value and a pointer to a character string. It is used in the utility programs wherever there is a need to relate some arbitrary code number or bit pattern to an ASCII character string. For example,
to program a serial port baud-rate-generator chip to various baud rates requires
different time constants for each rate. Users do not need to know what these
numbers are; they only need to be able to specify the baud rate as an ASCII string.

Thus, a code table is set up as follows:

<table>
<thead>
<tr>
<th>Baud Rate Constant</th>
<th>User's Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ox35</td>
<td>“300”</td>
</tr>
<tr>
<td>Ox36</td>
<td>“600”</td>
</tr>
<tr>
<td>Ox37</td>
<td>“1200”</td>
</tr>
<tr>
<td>Ox3A</td>
<td>“2400”</td>
</tr>
<tr>
<td>Ox3C</td>
<td>“4800”</td>
</tr>
<tr>
<td>Ox3E</td>
<td>“9600”</td>
</tr>
<tr>
<td>Ox3F</td>
<td>“19200”</td>
</tr>
</tbody>
</table>

A utility program now needs to be able to perform various operations using the
code table:

- Given the input parameter on the command tail, the utility must check
  whether the ASCII string is in the code table, display all of the legal options
  on the console if it is not, and return the code value for subsequent processing
  if it is.

- Given the current baud rate constant (held in the BIOS), the utility must scan
  the code table and display the corresponding ASCII string to tell the user the
  current baud rate setting.

The library includes specialized functions to do this, plus some additional
functions to make code tables more generally usable. These functions are

ct_init (code table initialize; Figure 11-1, v)
This function initializes a specific entry in a code table, setting the code
value and the pointer to the character string.

c_t_parce (code table parameter return code; Figure 11-1, w)
This performs an uppercase substring match on the specified key string,
returning either an error (the value CT_SNF — string not found) or a code
value.

c_t_code (code table return code; Figure 11-1, x)
This function is similar to ct_parce in that it scans a code table and returns
the corresponding code. It differs in the way that the comparison is done.
The entire search string is compared with the string in the code table entry.
A match only occurs when all characters are the same.

c_t_disps (code table display strings; Figure 11-1, y)
This function displays all strings in a given code table. It is used either when
the user has entered an invalid string, or when the utility program is
requested to show what options are available for a parameter.

c_t_index (code table return index; Figure 11-1, z)
This function, given a string, searches the code table and returns the index...
of the entry that has a string matching the search string. The index is not the code value; it is the number of the entry in the table.

cLstri (code table string index; Figure 11-1, aa)
This function, given an entry index number, returns a pointer to the string in that entry.

cLstrc (code table string code; Figure 11-1, bb)
This function, given a code number, returns a pointer to the string in the entry that has a matching code number.

Accessing a Directory via the BDOS

One problem associated with accessing the file directory directly, as illustrated by earlier functions, is that the program is presented with directory entries in exactly the order that they occur in the directory. For some programs, such as those that process groups of files, it is better to use the BDOS Search First and Search Next functions to access the directory.

Using the BDOS, the program can process the first file name to match an ambiguous search key, then go back to the BDOS to get the name of the next file, and so on. The library header contains a structure definition for a standard CP/M file control block (Figure 11-2, k).

Notice that the first byte of the FCB is a disk number rather than the user number of the directory entry. Note also the use of a union statement to describe the allocation block numbers.

The standard BDS C library contains a function, setfcb, that is given the address of an FCB and a pointer to a string containing a file name. It converts any "*" in the name to the appropriate number of "?", and fills the remainder of the FCB with zeros.

The example library contains the following functions designed for BDOS file directory access:

get_nfn (get next file name; Figure 11-1, i)
This function is given a pointer to an ambiguous file name and a pointer to an FCB. It returns with the FCB set up to access the next file that matches the ambiguous file name.

srch_file (search for file; Figure 11-1, j)
This function, used by get_nfn, issues either a Search First or a Search Next BDOS call.

conv_fname (convert file name; Figure 11-1, g)
This function converts a file name from an FCB into a form suitable for display on the console. It is similar to the conv_dfname function described earlier except that it outputs only the disk, file name, and type (not the user number) in the form

d:filename.typ
To signal the get_nfn function that you want the first file name, you must set the most significant bit of the first byte, the disk number.

Here is an example showing how to use the get_nfn function:

```
struct _fcb fcb;  /* Declare a file control block */
setmem(fcb, FCB_SIZE, 0);  /* Clear FCB to zeros */
fcb.fcb_disk = 0x80;  /* Mark FCB for "first time" */
while (get_nfn(fcb, "B:XYZ.*,.*"))
  {  /* Until get_nfn returns a zero */
    while (/* Not at end of file */)  /* Open the file using FCB */
      {  /* Process next record or Character in file*/
        /* Close the file */
      }
  }
```

The quoted string “B:XYZ.*,.*” could also be just a pointer to a string, or a parameter on the command line, argv[n].

The last function for BDOS processing of the file directory, conv_fname, is used to convert a file name for output to a terminal. Again, the repetitive code at the end clears the file attribute bits to avoid any side effects from the terminal.

---

**Utility Programs Enhancing Standard CP/M**

This group of utilities is designed to enhance those supplied by Digital Research. They do not take advantage of any special features of the enhanced BIOS in Figure 8-10 and can be used on any CP/M Version 2.2 installation.

With the exception of the ERASE utility, all of the utilities scan down the file directory using BIOS calls, as described earlier in this chapter.

**ERASE — A Safer Way to Erase Files**

There are two disadvantages to the Console Command Processor's built-in ERA command. First, it will unquestioningly erase groups of files. Second, if you have a file name with nongraphic or lowercase characters, you cannot use the ERA command, as the CCP converts the command tail characters to uppercase and terminates a file name on encountering any strange character in the string.

The ERASE utility shown in Figure 11-3 erases groups of files, but it asks the user for confirmation before it erases each file.

Rather than use the BIOS to access each directory entry, it uses the get_nfn function, which then calls the BDOS. Thus ERASE functions equally well for files
that have multiple entries in the directory. It can use the BDOS Delete File function to erase all extents of a given file.

Here is an example console dialog showing ERASE in operation:

```
P3A> erase<CR>
ERASE Version 1.0 02/23/83 (Library 1.0)
Usage :
    ERASE (d:)file_name.typ

P3A> erase *.com<CR>
CAUSE  Version 1.0 02/23/83 (Library 1.0)
Usage :
    ERASE (d:)file_name.typ
Searching for file(s) matching A:?????????.COM.
Erase A:UNERASE .COM y/n? y
Erase A:TEMP1 .COM y/n? y <= Will be Erased!
Erase A:TEMP2 .COM y/n? n
Erase A:TEMP3 .COM y/n? n
Erase A:TEMP4 .COM y/n? y <= Will be Erased!
Erase A:ERASE .COM y/n? n
Erasing files now...
File A:TEMP1 .COM erased.
```

```
#define VN "1.002/24/83"
/* ERASE
   This utility erases the specified file(s) logically
   by using a BDOS delete function. */
#include <LIBRARY.H>
struct_fcb amb_fcb;
struct_fcb fcb;
char filename[20];
short cur_disk;
#define MAXERA 1024
struct_fcb era_fcb[MAXERA];
int ecount;
int count;
main(argc, argv)
    short argc;
    char **argv;
    { printf("ERASE Version %s (Library %s)\n", VN, LIBVN);
    chk_use(argc);
    cur_disk = bdos(GETDISK);
    ecount = 0;
    setfcb(amb_fcb.fcb[1]);
    if (amb_fcb.fcb_disk)
        { bdos(SETDISK, amb_fcb.fcb_disk + 1); /* Set to specified disk */
    }

Figure 11-3. ERASE.C, a utility that requests confirmation before erasing
```
Chapter II: Additional Utility Programs

411

Figure 11-3. (Continued)
UNERASE — Restore Erased Files

UNERASE, as its name implies, can be used to "revive" an accidentally erased file. Only files whose allocation blocks have not been reallocated to other files can be revived. The UNERASE utility shown in Figure 11-4 builds a bit vector of all the allocation blocks used by active directory entries. Then it builds a bit vector for all the allocation blocks required by the file to be UNERASEd. If a Boolean AND between the two vectors yields a nonzero vector, then one or more blocks that originally belonged to the erased file are now allocated to other files on the disk.

```c
#define VN "1.0 02/12/83"

/* UNERASE --
   This utility does the inverse of ERASE: it restores
   specified files to the directory by changing the first byte of
   their directory entries from 0XE5 back to the specified user
   number. */
#include <LIBRARY.H>

struct _dirpb dir_pb;
struct _dir *dir_entry;
struct _scb scb;
struct _scb scba;
struct _dpb dpb;
struct _by inuse_bv;
struct _by extents;
struct _by file_bv;

char file_name[203];
short cur_disk;
int count;
int user;

main(argc,argv)
{
    /* Argument count */
    char.argv[];
    /* Argument vector (pointer to an array of chars.) */

    printf("UNERASE Version %s (Library %s), VN.LIBVN),
    chk_use(argv);
    /* Check usage */
    cur_disk = bdos(GETDISK);
    /* Get current default disk */

    /* Using a special version of the set search-control-block utility,
    set the disk, name, type (no ambiguous names), the user number
    to match only erased entries, and the length to compare
    the user, name, and type.
    This special version also returns the disk_id taken from
    the file name on the command line. */
    if ((dir_pb.dp_disk = ssetscb(scb,argv[1],0xE5,12)) == 0)
    {
        /* Use default disk */
        dir_pb.dp_disk = cur_disk;
    }
    else
    {
        /* make disk A = 0, B = 1 (for SELDISK) */
        dir_pb.dp_disk--;
    }

    printf("Searching disk %d",dir_pb.dp_disk);
    if(strscn(scb,"?")) /* Check if ambiguous name */
    {
        printf("Error -- UNERASE can only revive a single file at a time.");
        exit();
    }

    printf("%darchiving disk %d.",

    printf("%darchiving disk %d.

Figure 11-4. UNERASE.C, a utility program that "revives" erased files
/* Set up a special search control block that will match with all existing files. */

ssetscb(scba,".\.\?,12); /* Set file name and initialize SCB */

if (argc == 2) /* No user number specified */
    user = bdos(GETUSER,OnFF); /* Get current user number */
else
    user = atoi(argv[2]); /* Get specified number */
    if (user > 15)
        printf("\nUser number can only be 0 - 15.\n");
        exit(1);

/* Build a bit vector that shows the allocation blocks currently in use. SCBA has been set up to match all active directory entries on the disk. */

build_bv(inuse_bv,scba);

/* Build a bit vector for the file to be restored showing which allocation blocks will be needed for the file. */

if (!build_bv(file_bv,scb))
    { 
        printf("\nNo directory entries found for file \%s\.",
                argv[1]);
        exit(1);
    }

/* Perform a boolean AND of the two bit vectors. */

bv_and(file_bv,inuse_bv,file_bv);

/* Check if the result is nonzero -- if so, then one or more of the allocation blocks required by the erased file is already in use for an existing file and the file cannot be restored. */

if (bv_nz(file_bv))
    { 
        printf("\n--- This file cannot be restored as some parts of it have been re-used for other files! ---\n");
        exit(1);
    }

/* Continue on to restore the file by changing all the entries in the directory to have the specified user number. Note: There may be several entries in the directory for the same file name and type, and even with the same extent number. For this reason, a bit map is kept of the extent numbers unerased -- duplicate extent numbers will not be unerased. */

/* Set up the bit vector for up to 127 unerased extents */

bv_make(extents,16); /* 16 = 8 bits */

/* Set the directory to "closed", and force the get_node function to open it. */

dir_pb.dp_open = 0;

/* While not at the end of the directory, return a pointer to the next entry in the directory. */

while(dir_entry = get_node(dir_pb))
    {
        /* Check if user = 0XE5 and name, type match */
        if (strcmp(name(&scb.dir_entry) == NAME_EQ))
            {
                /* Test if this extent has already been unerased */
                if (bv_test(extents,dir_entry - de_extent))
                    { /* Yes it has */
                        printf("\n\tExtent %d of %s ignored.\n", dir_entry - de_extent.argv[1]);
                        continue; /* Do not unerase this one */
                    }
            }

Figure 11-4. (Continued)
else /* Indicate this extent unerased */
{
    bv_set(extents,dir_entry -> de_extent);
    dir_entry -> de_userno = user; /* Unerase entry */
    dir_pb.dp_write = 1; /* Need to write sector block */
    printf("\n\n\nExtent %d of %d unerased.",
        dir_entry -> de_extent.argv[1]);
}
}

printf("\n\nFile %s unerased in User Number %d.",
    argv[1],user);
bdos(SETDISK.cur_disk); /* Reset to current disk */
}

build_bv(bv,scb) /* Build bit vector (from directory) */
/* This function scans the directory of the disk specified in
the directory parameter block (declared as a global variable),
and builds the specified bit vector, showing all the allocation
blocks used by files matching the name in the search control
block. */
/* Entry parameters */
struct _bv *bv; /* Pointer to the bit vector */
struct _scb *scb; /* Pointer to search control block */
/* Also uses : directory parameter block (dir_pb) */
/* Exit parameters */
The specified bit vector will be created, and will have 1-bits
set wherever an allocation block is found in a directory
entry that matches the search control block.
It also returns the number of directory entries matched. */
{
    unsigned abno; /* Allocation block number */
    struct _dpb *dpb; /* Pointer to the disk parameter block in the BIOS */
    int mcount; /* Match count of dir. entries matched */
    mcount = 0; /* Initialize match count */
    d pb = get_dp b(dir_pb.dp_disk); /* Get disk parameter block address */
    /* make the bit vector with one byte for each eight allocation
        blocks + 1 */
    if (! (bv_make(bv, (dpb -> d pb_maxabn >> 3) +1)))
    {
        printf("Error -- Insufficient memory to make a bit vector.");
        exit (1);
    }
    /* Set directory to "closed" to force the set_nde
        function to open it. */
    dir_pb.dp_open = 0;
    /* Now scan the directory building the bit vector */
    while(dir_entry = get_nde(dir_pb))
    { /* Compare user number (which can legitimately be
            0xE5), the file name and the type). */
        if (comp_fname(scb,dir_entry) == NAME_EQ)
        {
            ++mcount; /* Update match count */
            for (count = 0; count < dir_pb.dp_nabpde; /* For number of alloc. blks. per dir. entry */
                count++)
            { /* Set the appropriate bit number for
                each nonzero allocation block number */
                if (dir_pb.dp_nabpde == 8) /* assume 8 2-byte numbers */
                {
                    abno = dir_entry -> _dirab.de_long[count];
                }
                else /* Assume 16 1-byte numbers */
                {
                    abno = dir_entry -> _dirab.de_long[count];
                }
            }
        }
    }
}

Figure 11-4. (Continued)
abno = dir_entry -> _dirab.de_short[count];
}
if (abno) bv_set(bv, abno); /* Set the bit */
}

return count; /* Return number of dir. entries matched */
}

chk_use(argc) /* Check usage */
/* This function checks that the correct number of parameters has been specified, outputting instructions if not. */
int argc; /* Count of the number of arguments on the command line */
{
  /* The minimum value of argc is 1 (for the program name itself), so argc is always one greater than the number of parameters on the command line */
  if (argc == 1 || argc > 3)
    {
      printf("\nUsage:");
      printf("\n\nUNERASE [disk]filename.type [user]");
      printf("\n\nOnly a single unambiguous file name can be used.");
      exit();
    }
} /* end chk_use */

ssetscb(scb, fname, user, length) /* Special version of set search control block */
/* This function sets up a search control block according to the file name, type, user number, and number of bytes to compare.
   The file name can take the following forms:
   filename
   filename.type
   *filename
   It sets the bit map according to which disks should be searched.
   For each selected disk, it checks to see if an error is generated when selecting the disk (i.e. if there are disk tables in the BIOS for the disk). */
struct _scb *scb; /* Pointer to search control block */
char *fname; /* Pointer to the file name */
short user; /* User number to be matched */
int length; /* Number of bytes to compare */
/* Exit parameters */
/* Disk number to be searched. (A = 1, B = 2...) */
{
  short disk_id; /* Disk number to search */
  setfcb(scb, fname); /* Set search control block as though it were a file control block. */
disk_id = scb -> scb_userno; /* Set disk_id before it gets overwritten by the user number */
scb -> scb_userno = user; /* Set user number */
scb -> scb_length = length; /* Set number of bytes to compare */
return disk_id;
} /* end ssetscb */

Figure 11-4. (Continued)
A further complication occurs if two or more directory entries of the erased file have the same extent number. This can happen if the file has been created and erased several times. Under these circumstances, UNERASE revives the first entry with a given extent number that it encounters, and displays a message on the console both when an extent is revived and when one is ignored.

Because of the complicated nature of the UNERASE process, the utility can process only a single, unambiguous file name.

The following console dialog shows UNERASE in operation:

```
P3A>dir *.com<CR>
A: UNERASE COM : TEMP2 COM : TEMP3 COM : ERASE COM

P3A>unerase<CR>
UNERASE Version 1.0 02/12/83 (Library 1.0)
Usage:
  UNERASE {d:}filename.type {user}
  Only a single unambiguous file name can be used.

P3A>unerase temp1.com<CR>
UNERASE Version 1.0 02/12/83 (Library 1.0)
Searching disk A.
  Extent #0 of TEMP1.COM unerased.
  Extent #0 of TEMP1.COM ignored.
File TEMP1.COM unerased in User Number 3.

P3A>dir *.com<CR>
A: UNERASE COM : TEMP1 COM : TEMP2 COM : TEMP3 COM
A: ERASE COM

P3A>unerase temp5.com<CR>
UNERASE Version 1.0 02/12/83 (Library 1.0)
Searching disk A.
No directory entries found for file TEMP5.COM.
```

**FIND — Find “Lost” Files**

The FIND utility shown in Figure 11-5 searches all user numbers on specified logical disks, matching each entry against an ambiguous file name. It can then display either a disk map showing how many matching files were found in each user number for each disk, or the user number, file name, and type for each matched directory entry.

You can use FIND to locate a specific file or group of files, as shown in the following console dialog:

```
P3B>find<CR>
FIND Version 1.0 02/11/83 (Library 1.0)
Usage:
  FIND d:filename.type {NAMES}
  *:filename.type (All disks)
  ABCD.OP:filename.type (Selected Disks)
  NAMES option shows actual names rather than map.

P3B>find ab:*.*<CR>
FIND Version 1.0 02/11/83 (Library 1.0)
```
Chapter II: Additional Utility Programs

Searching disk : A
Searching disk : B

Numbers show files in each User Number.

<table>
<thead>
<tr>
<th>User Numbers</th>
<th>Dir. Entries</th>
</tr>
</thead>
<tbody>
<tr>
<td>A:</td>
<td>1 1 8</td>
</tr>
<tr>
<td>B:</td>
<td>66 20 74 55 3</td>
</tr>
</tbody>
</table>

Press Space Bar to continue....

#include <LIBRARY.H>

- define VN "1.0 02/11/83"

/* FIND - This utility can display either a map showing on which disks and in which user numbers files matching the specified ambiguous file name are found, or the actual names matched. */

/* Directory management parameter block */
struct _dirpb dir_pb;
/* Pointer to directory entry (somewhere in dir_pb) */
struct _dir *dir_entry;
/* Search control block */
char scb scb;
/* Formatted for display : un/d.FILENAME.TYP */

Figure 11-5. FIND.C, a utility program that locates specific files or groups of files
short cur_disk;  /* Current logical disk at start of program */
int mcount;    /* Match count (no. of file names matched) */
int dmcount;   /* Per disk match count */
int lcount;    /* Line count (for lines displayed) */
int map_flag;  /* 0 = show file names of matched files, 
                 NI = show map of number of files */

/* The array below is used to tabulate the results for each 
   disk drive, and for each user number on the drive. 
   In addition, two extra "users" have been added for "free" 
   and "used" values. */

unsigned disk_map[16][18];  /* Disk A -> F, users 0 -> 15, free, used */
#define USED_COUNT 16        /* "User" number for used entities */
#define FREE_COUNT 17        /* "User" number for free entities */

main(argc, argv)
    /* Argument count */
    short argc;       /* Argument vector (pointer to an array of chars.) */
    char *argv[1];
    /* Argument count */
{
    printf("\nFIND Version %s (Library %s), VN,LIBVN);  
    chk_use(argc);    /* Check usage */
    cur_disk = bdos(0,GETDISK);  /* Get current default disk */
    dm clr(disk_map); /* Reset disk map */

    /* Set search control block 
       disks, name, type, user number, extent number, 
       and number of bytes to compare -- in this case, match all users, 
       but only extent 0 */
    setscb(scb, argv[1], '?', 0, 13);  /* Set disks, name, type */
    map_flag = usstrcmp("NAMES", argv[2]);  /* Set flag for map option */

    lcount = dmcount = mcount = 0;  /* Initialize counts */
    for (scb.scb_disk = 0; scb.scb_disk < 16; 
        scb.scb_disk++)  /* Starting with logical disk A */
    {
        /* Check if current disk has been selected for search */
        if !(scb.scb_adisks & (1 << scb.scb_disk))
            continue;  /* No, so byPass this disk */

        printf("\nSearching disk %c", (scb.scb_disk + 'A'));
        lcount++  /* Update line count */
    dir_pb.dp_disk = scb.scb_disk;  /* Set to disk to be searched */
    dmcount = 0;  /* Reset disk matched count */
        if (!map_flag)  /* If file names are to be displayed */
            putchar('\n');  /* Move to column 1 */
        else  /* If map option is on */
            while(dir_entry = get_nde(dir_pb))
                /* While not at the end of the directory, set a pointer to the 
                   next directory entry */
                if (dir_entry -> de_userno == OxE5)  /* Unused */
                    disk_map[scb.scb_disk][FREE_COUNT]++;
                else  /* In use */
                    disk_map[scb.scb_disk][USED_COUNT]++;
        /* Select only those active entries that are the 
           first extent (numbered 0) of a file that matches 
           the name supplied by the user */
}

Figure 11-5. (Continued)
if (dir_entry -> de_userno == 0x05) &&
(dir_entry -> de_extent == 0) &&
(comp_fname(scb, dir_entry) == NAME_EQ)
{
    mcount++; /* Update matched counts */
dcount++; /* Per disk count */
if (map_flag) /* Check map option */
{
    /* Update disk map */
    disk_map[scb.scb_disk][dir_entry -> de_userno]++;
}
else /* Display names */
{
    conv_dfnname(scb, scb_disk, dir_entry, file_name);
    printf("%,file_name}
    /* Check if need to start new line */
    if ((dcount % 4))
    {
        putchar(\n');
        if (++lcount > 18)
        {
            lcount = 0;
            printf("Press Space Bar to continue....");
getchar();
            putchar(\n');
        }
    }
}
} /* End of directory */
/* All disks searched */
if (map_flag)
{
    printf("n Numbers show files in each user number."));
    printf("n ---- User Numbers ---- Dir. Entries"());
    dmDisp(disk_map, scb.scb_adisks); /* Display disk map */
}
if (mcount == 0)
    printf("n --- File Not Found --- ");
bdos(SETDISK, cur_disk); /* Reset to current disk */
}
chk_usage(argc); /* check usage */
/* This function checks that the correct number of
   parameters has been specified, outputting instructions
   if not. */
/* Entry parameter */
int argc; /* Count of the number of arguments on the command line */
{
    /* The minimum value of argc is 1 (for the program name itself),
       so argc is always one greater than the number of parameters
       on the command line */
    if (argc == 1 || argc > 3)
    {
        printf("nUsage ");
        printf("n\t\tFIND :filename péri (NAMES)\t");
        printf("n\t \tfilename péri (All disks)\t");
        printf("n\t \tSELECTED disks peri (Selected Disks)\t");
        printf("n\t \tNAMES option shows actual names rather than map.\t");
        exit();
    }
}

Figure 11-5. (Continued)
SPACE — Show Used Disk Space

The SPACE utility shown in Figure 11-6 scans the specified logical disks and displays a disk map that shows, for each user number on each logical disk, how many Kbytes of storage have been used. It also displays the total number of Kbytes used and free on each logical disk.

Here is an example console dialog showing SPACE in operation:

```
P38> space<CR>
SPACE Version 1.0 02/11/83 (Library 1.0)
Usage :
    SPACE * (All disks)
    SPACE ABCD..OP (Selected Disks)
```

```
P38> space (CR>
SPACE Version 1.0 02/11/83 (Library 1.0)
Searching disk : A
Searching disk : B
Searching disk : C
Numbers show space used in kilobytes.

--- User Numbers ---

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>...</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>18</td>
<td>202</td>
<td>38</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>258</td>
<td>1196</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>692</td>
<td>432</td>
<td>656</td>
<td>548</td>
<td>36</td>
<td></td>
<td></td>
<td>2364</td>
<td>996</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>140</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>140</td>
<td>204</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Figure 11-6. SPACE.C, a utility that displays how much disk storage is used or available
print("\nSPACE Version Xs (Library Xs), VN, LIBVN);\nchk_use(argc); /* Check usage */\ncur_disk = bdos(GETDISK); /* Get current default disk */\nde_clr(disk_map); /* Reset disk map */\nsetscb(scb,argv[1]); /* Special version : set disks, name, type */\nfor (scb.scb_disk = 0; \n    scb.scb_disk < 16; \n    scb.scb_disk++) /* Move to next logical disk */\n{\n    /* Check if current disk has been selected for search */\n    if (!((scb.scb_adisks & (1 << scb.scb_disk)))\n        continue; /* No, so bypass this disk */\n    print("\nSearching disk X", scb.scb_disk + 'A');\ndir_pb.dp_disk = scb.scb_disk; /* Set to disk to be searched */\n    /* Set the directory to "closed", and force the get_nde\n    function to open it */\n    dir_pb.dp_open = 0;\n    /* While not at the end of the directory, set a pointer\n    to the next entry in the directory */\n    while (dir_entry = get_nde(dir_pb))\n    {\n        if (dir_entry -> de_userno == OxE5)\n            continue; /* Bypass inactive entries */\n        for (count = 0; \n            count < dir_pb.dp_nabpde; \n            count++) /* For number of alloc. blks. per dir. entry */\n        {\n            if (dir_pb.dp_nabpde == 8) /* Assume 8 2-byte numbers */\n                {\n                    disk_map[scb.scb_disk][dir_entry -> de_userno]\n                        += dir_pb.dp_absize; \n                        /* Build up sum for this disk */\n                    disk_map[scb.scb_disk][USED_COUNT] += disk_map[scb.scb_disk][user]; \n                }\n            else /* Assume 16 1-byte numbers */\n                {\n                    disk_map[scb.scb_disk][dir_entry -> de_userno]\n                        += dir_pb.dp_shorttcount; \n                        /* Build up sum for this disk */\n                    disk_map[scb.scb_disk][FREE_COUNT] += disk_map[scb.scb_disk][user]; \n                }\n        } /* All allocation blocks processed */\n    } /* End of directory for this disk */\n    /* Compute the storage used by multiplying the number of\n    allocation blocks counted by the number of Kbytes in\n    each allocation block. */\n    for (user = 0; \n    user < 16; \n    user++) /* Move to next user number */\n    {\n        /* Compute size occupied in Kbytes */\n        disk_map[scb.scb_disk][user] *= dir_pb.dp_absize; \n        /* Build up sum for this disk */\n        disk_map[scb.scb_disk][USED_COUNT] += disk_map[scb.scb_disk][user]; \n    } /* All disks processed */\n}\nprint("\nNumbers show space used in kilobytes.");\nprint("\n--- User Numbers ---\nSpace (Kb)");\ndisp(disk_map,scb,scb_adisks); /* Display disk map */\n\nFigure 11-6. (Continued)
The CP/M Programmer's Handbook

```c
bdos(SETDISK.cur_disk); /* Reset to current disk */
}
ssetscb(scb,ldisks); /* Special version of set search control block */
/* This function sets up a search control block according to just the logical disks specified. The disk are specified as a single string of characters without any separators. An asterisk means "all disks." For example --

    ABGH (disks A, B, G and H )

    * (all disks for which SELDSK has tables)

It sets the bit map according to which disks should be searched. For each selected disk, it checks to see if an error is generated when selecting the disk (i.e. if there are disk tables in the BIOS for the disk). The file name, type, and extent number are all set to "?" to match all possible entries in the directory. */
/* Entry parameters */
struct _scb *scb; /* Pointer to search control block */
char *ldisks; /* Pointer to the logical disks */
/* Exit parameters */
None.
*/

int disk; /* Disk number currently being checked */
unsigned adisks; /* Bit map for active disks */
adisks = 0; /* Assume no disks to search */
if (*ldisks) /* Some values specified */
{
    if (*ldisks == '*') /* Check if "all disks" */
    {
        adisks = 0xFFFF; /* Set all bits */
    }
    else /* Set specific disks */
    {
        while(*ldisks) /* Until end of disks reached */
        {
            /* Build the bit map by getting the next disk id. (A - P), converting it to a number in the range 0 - 15, and shifting a 1-bit left that many places and OR ing it into the current active disks. */
            adisks |= 1 << (toupper(*ldisks) - 'A');
            ++ldisks; /* Move to next character */
        }
    }
}
else /* Use only current default disk */
{
    /* Set just the bit corresponding to the current disk */
adisks = 1 << bdos(GETDISK);
}
/* Set the user number, file name, type, and extent to "?" so that all active directory entries will match */
/* 0123456789012 */
strncpy(&scb->scb_userno,"?????????????");
/* Make calls to the BIOS SELDSK routine to make sure that all of the active disk drives have disk tables for them in the BIOS. If they don't, turn off the corresponding bits in the bit map. */

for (disk = 0; /* Start with disk A */
disk < 16; /* Until disk P */
disk++) /* Use next disk */
{
    if ( !(1 << disk) & adisks))
        continue; /* Avoid selecting unspecified disks */
```
if (biosh(SELDISK,disk) == 0)  /* Make BIOS SELDSK call */
  { /* Returns 0 if invalid disk */
    /* Turn OFF corresponding bit in mask
    by AND-ing it with bit mask having
    all the other bits set = 1. */
    adisks &= (((1 << disk) ^ OxFFFF));
  }
  scb -> scb_adisks = adisks; /* Set bit map in scb */
} /* End ssetscb */

chk_use(argc)  /* Check usage */
/* This function checks that the correct number of
parameters has been specified, outputting instructions
if not. */
/* Entry parameter */
int argc;  /* Count of the number of arguments on the command line */
{
  /* The minimum value of argc is 1 (for the program name itself),
  so argc is always one greater than the number of parameters
  on the command line */
  if (argc != 2)
    { /* End chk_use */
      printf("Usage: ");
      printf("SPACE = (All disks)");
      printf("(Selected Disks)");
      exit();
    }
}

Figure 11-6. (Continued)

MOVE — Move Files Between User Numbers

The MOVE utility shown in Figure 11-7 moves files from one user number to
another on the same logical disk. The movement is achieved by changing the user
number in all the relevant directory entries. This is much faster than copying the
files. It also avoids having multiple copies of the same file on the disk.

Here is a console dialog showing MOVE in operation:

P3B>move(CR>
MOVE Version 1.0 02/10/83 (Library 1.0)
Usage:
  MOVE d:filename.typ to_user {from_user} {NAMES}
  =filename.typ (All disks)
  ABCD..OP:filename.typ (Selected Disks)
NAMES option shows names of files moved.

P3B>dir *.com(CR>
B: ERASE COM : FUNKEY COM : DATE COM : FIND COM
B: SPACE COM : UNERASE COM : MAKE COM : MOVE COM
B: TIME COM : ASSIGN COM : SPEED COM : PROTOCOL COM

P3B>move *.com 0 names(CR>
MOVE Version 1.0 02/10/83 (Library 1.0)

Moving file(s) 3/B:?????????.COM -> User 0.
#define VN "1.0 02/10/83"

/* MOVE -- This utility transfers file(s) from one user number to another, but on the same logical disk. Files are not actually copied -- rather, their directory entries are changed. */

#include <LIBRARY.H>

struct _dirpb dir_pb;  /* Directory management parameter block */
struct _dir *dir_entry;  /* Pointer to directory entry */
struct _scb scb;  /* Search control block */

#define DIR_BSZ 128  /* Directory buffer size */
char dir_buffer[DIR_BSZ];  /* Directory buffer */
char file_name[20];  /* Formatted for display: un/d:FILENAME,TYP */
short name_flag;  /* NZ to display names of files moved */
short cur_disk;  /* Current logical disk at start of program */
int from_user;  /* User number from which to move files */
int to_user;  /* User number to which files will be moved */
int mcount;  /* Match count (no. of file names matched) */
int dcount;  /* Per-disk match count */
int lcount;  /* Line count (for lines displayed) */

main(argc,argv)
short argc;
char argv[];
{
printf("MOVE Version %s (Library %s)\n",VN,LBN);

chk_use(argc);  /* Check usage */

to_user = atoi(argv[2]);  /* Convert user no. to integer */

/* Set and check destination user number */
if(to_user > 15)
{
printf("Error -- the destination user number cannot be greater than 15.\n");
exit0;
}

/* Set the current user number */
from_user = bdos(OETUSER,0xFF);

/* Check if source user number specified */
if(isdigit(argv[3][0]))
{
/* Set and check source user number */

if((from_user = atoi(argv[3])) > 15)
{
printf("Error -- the source user number cannot be greater than 15.\n");
exit();
}

/* Set name suppress flag from parameter #4 */
name_flag = usstrcmp("NAMES",argv[4]);
}
else
/* No source user specified */
}

Figure 11-7. MOVE.C, a utility program that changes files' user numbers
/* Set name suppress flag from parameter #3 */
name_flag = usstrcmp("NAMES",argv[3]);

/* To simplify the logic below, name_flag must be made
NZ if it is equal to NAME_EQ, 0 if it is any other value */
name_flag = (name_flag == NAME_EQ ? 1 : 0);

if (to_user == from_user) /* To = from */
{
    printf("Error - 'to' user number is the same as 'from' user number.");
    exit(1);
}

/* Set the search control block file name, type, user number,
extent number, and length -- length matches user number, file
name, and type. As the extent number does not enter into the
comparison, all extents of a given file will be found. */
setscb(scb, argv[11], from_user, '/', 13);

cur_disk = bdos(GETDISK); /* Get current default disk */
lcount = dmcount = mcount = 0; /* Initialize counts */
for (scb.scb_disk = 0; /* Starting with logical disk A */
    scb.scb_disk < 16; /* Until logical disk P */
    scb.scb_disk++) /* Move to next logical disk */
{
    /* Check if current disk has been selected for search */
    if ((scb.scb_disk & (1 << scb.scb_disk)))
        continue; /* No, so bypass this disk */
    /* Convert search user number and name for output */
    conv_dfname(scb.scb_disk, scb.file_name);
    printf("\nMoving file(s) %s -> User %d., file_name, to_user);

    lcount++;
    /* Update line count */
    dir_pb.dp_disk = scb.scb_disk; /* Set to disk to be searched*/
    dmcount = 0; /* Reset disk matched count */
    if (name_flag) /* If file names are to be displayed */
        putchar('\n'); /* Move to column 1 */
    /* Set the directory to "closed" to force the getade
    function to open it. */
    dir_pb.dp_open = 0;

    /* While not at the end of the directory, set a pointer
to the next directory entry */
    while(dir_entry = get_nde(dir_pb))
    {
        /* Match those entries that have the correct
        user number, file name, type, and any
        extent number. */
        if (/*
            (dir_entry -> de_userno != OXE5) &
            (comp_fname(scb.dir_entry) == NAME_EQ)
        */
            dir_entry -> de_userno == to_user; /* Move to new user */
            /* Request sector to be written back */
            dir_pb.dp_write = 1;
            mcount++; /* Update matched counts */
            dmcount++; /* Per-disk count */
            if (name_flag) /* Check map option */
                {
                    conv_dfname(scb.scb_disk, dir_entry, file_name);
                    printf("%s ", file_name);
                    /* Check if need to start new line */
                    if (! (dmcount % 4))
                        {putchar('
');
                        if (++lcount > 18)
Other Utilities

The utility programs described in this section are by no means a complete set. You may want to develop many other specialized utility programs. Some possibilities are:

FILECOPY

A more specialized version of PIP could copy ambiguously specified groups of files. Of special importance would be the ability to read a file containing the names of the files to be copied. A useful option would be the ability to detect the setting of the unused file attribute bit and copy only files that have been changed.

PROTECT/UNPROTECT

This pair of utilities would allow you to “hide” files in user numbers greater than 15. Files so hidden could not be accessed other than by UNPROTECTing them, thereby moving them back into the normal user number range.
RECLAIM
This utility would read all sectors on a disk (using the BIOS). Any bad sectors encountered could then be logically removed by creating an entry in the file directory, with allocation block numbers that would effectively "reserve" the blocks containing the bad sectors.

OWNER
This utility, given a track or sector number, would access the directory and determine which file or files were using that part of the disk. This is useful if you have a bad sector or track on a disk. You then can determine which files have been damaged.

Utility Programs for the Enhanced BIOS
This section describes several utility programs that work with the enhanced BIOS shown in Figure 8-10. Several of these utilities work directly with the physical devices on the computer system, which can vary from computer to computer. The library header contains #define declarations for device numbers and names for physical devices (Figure 11-2, f and Figure 11-2, g).

These #define statements are used to build a physical-device code table. If you have more physical devices or want to change the names by which you refer to the devices, you will need to change these definitions.

All of these utilities share some common features in the way that they are invoked. If they are called without any parameters, they display instructions on the console regarding what parameters are available. If they are called with the word "SHOW" (or "S", "SH", and so forth) as a parameter, they display the current settings of whatever attribute the utility controls.

MAKE — Make Files "Invisible" or "Visible"

The MAKE utility shown in Figure 11-8 is designed to operate in conjunction with the public files option implemented in the enhanced BIOS of Figure 8-10. It has two modes of operation — making files "invisible" or "visible."

An invisible file is one in user 0 which has been set to Read-Only and System status. When the public files option is enabled, these files cannot be seen when you use the DIR command, nor can they be erased accidentally.

A visible file is one that has been set to Read/Write and Directory status.

When files are made invisible, they are transferred from the current user number to user 0. When files are made visible, they are transferred from user 0 to the current user number.

Here is an example console dialog showing MAKE in operation:

P3B>make<CR>
MAKE Version 1.0 02/12/83 (Library 1.0)
430 The CP/M Programmer's Handbook

```c
#count++;
/

/* Update matched counts */

if (invisible)
{ /* Set ms bits */
    dir_entry -> de_fname[8] |= 0x80;
    dir_entry -> de_fname[9] |= 0x60;
}
else /* Visible */
{ /* Clear ms bits */
    dir_entry -> de_fname[8] &= 0x7F;
    dir_entry -> de_fname[9] &= 0x7F;
}

/* Move to correct user number */
/* Indicate sector to be written back */
/* Check if name to be displayed */
if (name_flag)
{
    conv_dfname(scb, scb_disk, dir_entry, file_name);
    printf("\n\%s made \%s in User \%d.",
           file_name, operation, to_user);
}
/* All directory entries processed */
/* All disks processed */
if (count == 0)
    printf("\n --- No Files Processed --- ");
bdos(setdisk, cur_disk); /* Reset to current disk */
}

chk_use(argc) /* Check usage */
/* This function checks that the correct number of
   parameters has been specified.  Outputting instructions
   if not. */
/* Entry parameter */
int argc; /* Count of the number of arguments on the command line */
{
    /* The minimum value of argc is 1 (for the program name itself),
       so argc is always one greater than the number of parameters
       on the command line */
    if (argc == 3 || argc == 4)
        return;
    else
        printf("\nUsage :");
        printf("\n\tMAKE dirfilename.type INVISIBLE (NAMES) ");
        printf("\n\t VISIBLE ");
        printf("\n\t filename.type (All disks) ");
        printf("\n\t ABCD..OPTfilename.type (Selected Disks) ");
        printf("\n\t NAMES option shows names of files Processed.");
        exit(1);
    }

Figure 11-8. (Continued)"
SPEED — Set Baud Rates

The SPEED utility shown in Figure 11-9 sets the baud rate for a specific serial device. Here is an example console dialog that shows several of the options:

P3B>speed<CR>
SPEED 1.0 02/17/83
The SPEED utility sets the baud rate speed for each physical device.
Usage is: SPEED physical-device baud-rate, or
SPEED SHOW (to show current settings)

Valid physical devices are:
TERMIAL
PRINTER
MODEM

Valid baud rates are:
300
600
1200
2400
4800
9600
19200

P3B>speed show<CR>
SPEED 1.0 02/17/83
Current Baud Rate settings are:
TERMINAL set to 9600 baud.
PRINTER set to 9600 baud.
MODEM set to 9600 baud.

P3B>speed m 19<CR>
SPEED 1.0 02/17/83
Current Baud Rate settings are:
TERMINAL set to 9600 baud.
PRINTER set to 9600 baud.
MODEM set to 19200 baud.

P3B>speed XYZ 12<CR>
SPEED 1.0 02/17/83
Physical Device 'XYZ' is invalid or ambiguous.
Legal Physical Devices are:
TERMIAL
PRINTER
MODEM

#define VN "\nSPEED 1.0 02/17/83"
/* This utility sets the baud rate speed for each of the physical devices. */
#include <LIBRARY.H>
struct ct cl_pdevc(MAXPDEV + 2); /* Physical device table */
/* Hardware specific items */

Figure 11-9. SPEED.C, a utility that sets the baud rate for a specific device
define B300 0x35 /* 300 baud */
define B600 0x36 /* 600 baud */
define B1200 0x37 /* 1200 baud */
define B2400 0x3A /* 2400 baud */
define B4800 0x3C /* 4800 baud */
define B9600 0x3E /* 9600 baud */
define B19200 0x3F /* 19200 baud */
struct _ct ct_br[10]: /* Code table for baud rates (+ spare entries) */

/* Parameters on the command line */
define PDEV argv[1] /* Physical device */
define BAUD argv[2] /* Baud rate */

main(argc, argv)
int argc;
char *argv[];

print(VN); /* Display sign-on message */
setup(); /* Set up code tables */
chk_use(argc); /* Check correct usage */

  */ Check if request to show current settings */
if (strstr(argv[1], "SHOW"))
{
  /* No -- assume setting is required */
  set_baud(get_pdev(PDEV), get_baud(BAUD)); /* Set baud rate */
}

show_baud(); /* Display current settings */

} /* end of program */

setup() /* set up the code tables for this program */
{
  */ Initialize the physical device table */
  ct_init(ct_pdev[0], T_DEVN, PN_T); /* Terminal */
  ct_init(ct_pdev[1], P_DEVN, PN_P); /* Printer */
  ct_init(ct_pdev[2], M_DEVN, PN_M); /* Modem */
  ct_init(ct_pdev[3], CT_SNF, "); /* Terminator */

  */ Initialize the baud rate table */
  ct_init(ct_br[0], B300, "300");
  ct_init(ct_br[1], B600, "600");
  ct_init(ct_br[2], B1200, "1200");
  ct_init(ct_br[3], B2400, "2400");
  ct_init(ct_br[4], B4800, "4800");
  ct_init(ct_br[5], B9600, "9600");
  ct_init(ct_br[6], B19200, "19200");
  ct_init(ct_br[7], CT_SNF, "); /* Terminator */
}

unsigned get_pdev(pdev)
/* Get physical device */
/* This function returns the physical device code
specified by the user in the command line. */
char *pdev; /* Pointer to character string */
{
  unsigned retval; /* Return value */

  retval = ct_paro(ct_pdev, pdev); /* Get code for ASCII string */
  if (retval == CT_SNF)
    { /* If string not found */
      printf("\n\n07 Physical Device '%s' is invalid or ambiguous.",
        pdev);
      printf("\nLegal Physical Devices are:");
      ct_disps(ct_pdev); /* Display all values */
      exit(0);
    }

  return retval; /* Return code */
}

unsigned get_baud(pbaud)
/* This function returns the baud rate time constant for
the baud rate specified by the user in the command line */

Figure 11-9. (Continued)
char *pbaud;  /* Pointer to character string */
{
    unsigned retval;  /* Return value */
    retval = ct_mpc(ct_br, pbaud);  /* Get code for ASCII string */
    if (retval != CT_SNFF)  /* If string not found */
    {
        printf("\n\007Baud Rate 'x' is invalid or ambiguous.",
               pbaud);
        printf("\nInvalid Baud Rates are : ");
        ct_disps(ct_br);  /* Display all values */
        exit(1);
    }
    return retval;  /* Return code */
}

set_baud(pdevc, baudc)  /* Set the baud rate of the specified device */
int pdevc;
/* Physical device code */
short baudc;
/* Baud rate code */
{  /* On some systems this may have to be a
two-byte (unsigned) value */
    short *baud_rc;
    /* Pointer to the baud rate constant */
    /* On some systems this may have to be a
two-byte (unsigned) value */
    /* Notes: the respective codes for accessing the baud rate constants
via the get_cba (get configuration block address) function are:
Device 00 = 19, 01 = 21, 02 = 23. This function uses this
mathematical relationship */
    /* Set up pointer to the baud rate constant */
    baud_rc = get_cba(CB_DO_BRC + (pdevc << 1));
    /* Then set the baud rate constant */
    *baud_rc = baudc;
    /* Then call the BIOS initialization routine */
bios(CIOINIT, pdevc);
}

show_baud()  /* Show current baud rate */
{
    int pdevn;  /* Physical device number */
    short baudc;  /* Baud rate code */
    /* On some systems this may have to be a
two-byte (unsigned) value */
    short *baud_rc;
    /* Pointer to the baud rate constant */
    /* On some systems this may have to be a
two-byte (unsigned) value */
    /* Notes: the respective codes for accessing the baud rate constants
via the get_cba (get configuration block address) function are:
Device 00 = 19, 01 = 21, 02 = 23. This function uses this
mathematical relationship */
    printf("\nCurrent baud rate settings are: ");
    for (pdevn = 0; pdevn <= MAXPDEV; pdevn++)  /* All physical devices */
    {
        /* Set up pointer to the baud rate constant --
the code for the get_cba function is computed
by adding the physical device number »2 to
the Baud Rate code for device »0 */
        baud_rc = get_cba(CB_DO_BRC + (pdevn << 1));
        /* Then set the baud rate constant */
        *baud_rc = baudc;
        printf("\n\%d set to %s baud.",
               ct_strc(ct_pdev, pdevn),  /* Get ptr. to device name */
               ct_strc(ct_br, baudc));  /* Get ptr. to baud rate */
    }
}

chk_use(argc)  /* Check correct usage */
int argc;  /* Argument count */
{
if (argc == 1)
{
    printf("The SPEED utility sets the baud rate speed for each physical device.");
    printf("Usage is: SPEED physical-device baud rate, or ");
    printf("SPEED SHOW (to show current settings )");
    printf("Valid physical devices are: ");
    ct_disps(ct_pdev);
    printf("Valid baud rates are: ");
    ct_dlsps(ct_br);
    exit(0);
}

Figure 11-9. (Continued)

PROTOCOL — Set Serial Line Protocols

The PROTOCOL utility shown in Figure 11-10 is used to set the protocol for a specific serial device.

The drivers for each physical device can support several serial line protocols. The protocols are divided into two groups, depending on whether they apply to data output by or input to the computer.

Note that the output DTR and input RTS protocols can coexist with other protocols. The strategy is first to set the required character-based protocol and then to set the DTR/RTS protocol. There is an example of this in the following console dialog:

P3B>protocol<CR>
PROTOCOL Vn 1.0 02/17/83
PROTOCOL sets the physical device's serial protocols.
PROTOCOL physical-device direction protocol {message-length}

Legal physical devices are:
TERMINAL
PRINTER
MODEM

Legal direction/protocols are:
Output DTR
Output XON
Output ETX
Input RTS
Input XON

Message length can be specified with Output ETX.

P3B>protocol show<CR>
PROTOCOL Vn 1.0 02/17/83
Protocol for TERMINAL - None.
Protocol for PRINTER - Output XON
Protocol for MODEM - Input RTS

P3B>protocol m o g 128<CR>
PROTOCOL Vn 1.0 02/17/83
Protocol for TERMINAL - None.
Protocol for PRINTER - Output XON
Protocol for MODEM - Output ETX Message Length 128 bytes.

P3B>protocol m o d<CR>
PROTOCOL Vn 1.0 02/17/83
Protocol for TERMINAL - None.
Protocol for PRINTER - Output XON
Protocol for MODEM - Output DTR Output ETX Message Length 128 bytes.

#define VN "\nPROTOCOL Vn 1.0 02/17/83"
/* PROTOCOL -- This utility sets the serial port protocol for the
specified physical device. Alternatively, it displays the
current protocols for all of the serial devices. */
#include <LIBRARY.H>

/* Code tables used to relate ASCII strings to code values */
struct _ct ct_iproto[3]; /* Code table for input protocols */
struct _ct ct_oproto[3]; /* Code table for output protocols */
struct _ct ct_pdev[MAXDEV + 2]; /* Physical device table */
struct _ct ct_io[3]; /* Input, output */

/* Parameters on the command line */
#define PDEV argv[1] /* Physical device */
#define IO argv[2] /* Input/output */
#define PROTO argv[3] /* Protocol */
#define PROTOL argv[4] /* Protocol message length */

main(argc, argv)
int argc;
char argv[];
{
printf(VN); /* Display sign-on message */
setproto(argv[1]); /* Protocol */

/* End of program */
}

setproto() /* Set up the code tables for this program */
{
/* Initialize the physical device table */
ct_init(ct_pdev[0], 0, PDEV); /* Terminal */
ct_init(ct_pdev[1], 1, PDEV); /* Printer */
ct_init(ct_pdev[2], 2, PDEV); /* Modem */
ct_init(ct_pdev[3], T_SNF, "*"); /* Terminator */

/* Initialize the input/output table */
ct_init(ct_io[0], 0, "INPUT");
ct_init(ct_io[1], 1, "OUTPUT");
ct_init(ct_io[2], T_SNF, "*"); /* Terminator */

/* Initialize the output protocol table */
c_t_init(ct_oprotot[0], DT_DTR, "DTR");
c_t_init(ct_oprotot[1], DT_OXON, "XON");
c_t_init(ct_oprotot[2], DT_ETX, "ETX");

Figure 11-10. PROTOCOL.C, a utility that sets the protocol governing input and output of a specified serial device
ct_init(CT_OPyoto[3],CT_SNF,"*");  // Terminator */
/* Initialize the input protocol table */
ct_init(CT_Iproto[0],DT_IRTS,"RTS");
ct_init(CT_Iproto[1],DT_IIXON,"XON");
ct_init(CT_Iproto[2],CT_SNF,"*");  // Terminator */
/* Initialize the display protocol */
ct_init(CT_Dproto[0],DT_ODTR,"Output DTR");
ct_init(CT_Dproto[1],DT_OXON,"Output XON");
ct_init(CT_Dproto[2],DT_OETX,"Output ETX");
c_t_init(CT_Dproto[3],CT_SNF,"*");

unsigned getpdev(ppdev)  /* Get physical device */
/* This function returns the physical device code specified by the user in the command line. */
char *pdev;  /* Pointer to character string */
{ unsigned retval;  /* Return value */
  retval = ct_parct(CT_pdev, pdev);  /* Get code for ASCII string */
  if (retval == CT_SNF)  /* If string not found */
    {
      printf("\n\007Physical Device 'xs' is invalid or ambiguous.", pdev);
      printf("\nLegal Physical Devices are: ");
      ct_disps(ct_pdev);  /* Display all values */
      exit();
    }
  return retval;  /* Return code */
}

unsigned getio(pio)  /* Get input/output parameter */
char *pio;  /* Pointer to character string */
{ unsigned retval;  /* Return value */
  retval = ct_parct(CT_io, pio);  /* Get code for ASCII string */
  if (retval == CT_SNF)  /* If string not found */
    {
      printf("\n\007Input/Output direction 'xs' is invalid or ambiguous.", pio);
      printf("\nLegal values are: ");
      ct_disps(ct_io);  /* Display all values */
      exit();
    }
  return retval;  /* Return code */
}

unsigned getproto(output,ppproto)  /* This function returns the protocol code for the protocol specified by the user in the command line. */
  int output;  /* =1 for output, =0 for input */
char *ppproto;  /* Pointer to character string */
{ unsigned retval;  /* Return value */
  if (output)  /* OUTPUT specified */
    {
      /* Get code for ASCII string */
      retval = ct_parct(CT_oproto, pproto);  /* If string not found */
      if (retval == CT_SNF)
        {
          printf("\n\007Output Protocol 'xs' is invalid or ambiguous.", pproto);
          printf("\nLegal Output Protocols are: ");
          ct_disps(ct_oproto);  /* Display valid protocols */
          exit();
        }
    }
}

Figure 11-10. (Continued)
Figure 11-10. (Continued)
for (pdevc = 0; pdevc <= MAXPDEV; pdevc++)
{
    /* Set pointer to device table */
    dt = pdt[pdevc];
    if (dt) /* Check if pointer in array is valid */
    {
        printf("Protocol for %s = ", ct_strc(ct_pdev, pdevc));
        /* Check if any protocols set */
        if (! (dt -> dt_stl & ALLPROTO))
            printf("None.");
        continue;
    }

    /* Set pointer to display protocol table */
    dproto = ct_dproto;
    while (dproto -> _ct_code != CT_SNF)
    {
        /* Check if protocol bit set */
        if (dproto -> _ct_code & dt -> dt_stl)
            
            printf("%s ", dproto -> _ct_sp);
        ++dproto; /* Move to next entry */
    }
    /* Check if ETX/ACK protocol and message length to be displayed */
    if (dt -> dt_stl & DT_OETX)
        printf("Message length %d bytes.",
            dt -> dt_etxml);
    }
}

chk_use(argc) /* Check for correct usage */
int argc; /* Argument count on command line */
{
    if (argc == 1)
    {
        printf("\nPROTOCOL sets the physical device's serial protocols.");
        printf("\nPROTOCOL physical-device direction protocol (message-length)\n");
        printf("\nLegal physical devices are ");
        ct_disps(ct_pdev);
        printf("\nLegal direction/protocols are ");
        ct_disps(ct_dproto);
        printf("\nMessage length can be specified with Output ETX, \n\n");
        exit();
    }
}

Figure 11-10. (Continued)

ASSIGN — Assign Physical to Logical Devices

The ASSIGN utility shown in Figure 11-11 sets the necessary bits in the physical input/output redirection bits in the BIOS. It assigns a logical device's input and output to physical devices. Input can only be derived from a single physical device, while output can be directed to multiple devices.

Here is an example console dialog showing ASSIGN in action:

P3B>assign<CR>
ASSIGN Vn 1.0 02/17/83
ASSIGN sets the Input/Output redirection.
ASSIGN logical-device INPUT physical-device
ASSIGN logical-device OUTPUT physical-dev1 {phy_dev2...}
ASSIGN SHOW (to show current assignments)
Legal logical devices are:

- CONSOLE
- AUXILIARY
- LIST

Legal physical devices are:

- TERMINAL
- PRINTER
- MODEM

P3B> **assign show**<CR>
ASSIGN Vn 1.0 02/17/83
Current Device Assignments are:

- CONSOLE INPUT is assigned to - TERMINAL
- console output is assigned to - TERMINAL
- AUXILIARY INPUT is assigned to - MODEM
- AUXILIARY OUTPUT is assigned to - MODEM
- LIST INPUT is assigned to - PRINTER
- LIST OUTPUT is assigned to - PRINTER

P3B> **assign a o t m**<CR>
ASSIGN Vn 1.0 02/17/83
Current Device Assignments are:

- CONSOLE INPUT is assigned to - TERMINAL
- console output is assigned to - TERMINAL
- AUXILIARY INPUT is assigned to - MODEM
- AUXILIARY OUTPUT is assigned to - TERMINAL PRINTER MODEM
- LIST INPUT is assigned to - PRINTER
- LIST OUTPUT is assigned to - PRINTER

```c
#define VN "\nASSIGN Vn 1.0 02/17/83"
#include <LIBRARY.H>
struct _ct ct_pdev[MAXPDEV + 2]; /* Physical device table */

    /* Names of logical devices */
#define LN_C "CONSOLE"
#define LN_A "AUXILIARY"
#define LN_L "LIST"
struct _ct ct_ldev[4]; /* Logical device table */
struct _ct ct_iot[3]; /* Input, output */

    /* Parameters on the command line */
#define LDEV argv[1] /* Logical device */
#define IO argv[2] /* Input/output */

main(argc, argv)
int argc;
char *argv[];
{
    printf(VN); /* Display sign-on message */
    setup(); /* Set up code tables */
    chk_use(argc); /* Check correct usage */

    /* Check if request to show current settings */
    if (usstream("SHOW", argv[1]))
    { /* No, assume a set is required */
```

Figure 11-11. ASSIGN.C, a utility that assigns a logical device’s input and output to two physical devices
NOTE: the number of physical devices to process is given by argc - 3

set_assign(get_ldev(LDEV), get_io(IO), argc - 3, argv);
}

show_assign();

setup(); /* Set up the code tables for this program */
{
    /* Initialize the physical device table */
    ct_init(ct_pdev[0], 0, PN_T);  /* Terminal */
    ct_init(ct_pdev[1], 1, PN_P);  /* Printer */
    ct_init(ct_pdev[2], 2, PN_M);  /* Modem */
    ct_init(ct_pdev[3], CT_SNF, "snf");  /* Terminator */

    /* Initialize the logical device table */
    ct_init(ct_ldev[0], 0, LN_C);  /* Terminal */
    ct_init(ct_ldev[1], 1, LN_A);  /* Auxiliary */
    ct_init(ct_ldev[2], 2, LN_L);  /* List */
    ct_init(ct_ldev[3], CT_SNF, "snf");  /* Terminator */

    /* Initialize the input/output table */
    ct_init(ct_io[0], 0, "INPUT");
    ct_init(ct_io[1], 1, "OUTPUT");
    ct_init(ct_io[2], CT_SNF, "snf");  /* Terminator */
}

unsigned
get_ldev(pldev) /* Get logical device */
{
    /* This function returns the logical device code specified by the user in the command line. */
    char *pldev;  /* Pointer to character string */

    unsigned retval;  /* Return value */
    retval = ct_par(c(ct_ldev, pldev));  /* Get code for ASCII string */
    if (retval == CT_SNF)  /* If string not found */
    {
        printf("\n\nLogical device \"%s\" is invalid or ambiguous.",
pldev);
        printf("\nLegal logical devices are: ");
        ct_disps(ct_ldev);  /* Display all values */
        exit();
    }
    return retval;  /* Return code */
}

unsigned
get_io(pio) /* Get input/output parameter */
{
    char *pio;  /* Pointer to character string */

    unsigned retval;  /* Return value */
    retval = ct_par(c(ct_io, pio));  /* Get code for ASCII string */
    if (retval == CT_SNF)  /* If string not found */
    {
        printf("\nInput/output direction \"%s\" is invalid or ambiguous.",
pio);
        printf("\nLegal values are: ");
        ct_disps(ct_io);  /* Display all values */
        exit();
    }
    return retval;  /* Return code */
}

set_assign(ldevc, output, argc, argv) /* Set assignment (I/O redirection) */
{
    int ldevc;  /* Logical device code */
    int output;  /* I/O redirection code */
    int argc;  /* Count of arguments to process */
    char *argv[1];  /* Replica of parameter to main function */

    unsigned *redir;  /* Pointer to redirection word */
    int pdevc;  /* Physical device code */
    unsigned rd_val;  /* Redirection value */

    /* Get the address of the I/O redirection word.

Figure 11-11. (Continued)
This code assumes that get_cba code values are ordered:
Device 0, input & output
Device 1, input & output
Device 2, input & output

The get_cba code is computed by multiplying the logical device code by 2 (that is, shift left 1) and added onto the code for Device 0. output
Then the output variable (0 = input, 1 = output) is added on:

```c
redir = get_cba(CB_CI + (ldev << 1) + output);
```

```c
rd_val = 0;  // Initialize redirection value /

    // For output, assignment can be made to several physical devices, so this code may be executed several times /
    do
        
            // Get code for ASCII string /
            /# Notes: the physical device parameters start
            with parameter #3 (argv[3]). However argc is a decreasing count of the number of physical devices to be processed. Therefore, argc + 2 causes them to be processed in reverse order (i.e. from right to left on the command line) #/
            pdevc = ct_part(ct_pdev, argv[argc + 2]);
            if (pdevc == CT_SNF)  // If string not found #/
                {
                    printf("\n%07Phyiscal device 'X' is invalid or ambiguous.\n",
                       argv[argc + 2]);
                    printf("\nLegal physical devices are 1 \n");
                    ct_disp(ct_pdev);  // Display all values #/
                    exit();
                }
            else
                {
                    // Repeat this loop for as long as there are
                    more parameters (for output only) #/
                    rd_val = (1 << pdevc);  
                }
    } while (--argc && output);
`redir = rd_val;  // Set the value into the config. block #/
```

```c
show_assign()  // Show current baud rate #/
{
    int rd_code;  // Redirection code for get_cba #/
    int ldevn;  // Logical device number #/
    int pdevn;  // Physical device number #/
    unsigned rd_val;  // Redirection value #/
    unsigned *prd_val;  // Pointer to the redirection value #/
```

```c
    // Notes: the respective codes for accessing the redirection values
    // via the get_cba (get configuration block address) function are:
    Device 0 console input -- 5
    Device 0 console output -- 6
    Device 1 auxiliary input -- 7
    Device 1 auxiliary output -- 8
    Device 2 list input -- 9
    Device 2 list output -- 10
```

```c
    This function uses this mathematical relationship #: printf("\nCurrent device assignments are i\n");
```

```c
    // For all get_cba code values
    for (rd_code = CB_CI; rd_code <= CB_LO; rd_code++)
    {
        // Set pointer to redirection value #/
        prd_val = get_cba(rd_code);
        // Get the input redirection value #/
```

\*Figure 11-11. (Continued)\*
The CP/M Programmer's Handbook

```c
rd_val = *rd_val;  /* This also performs byte reversal */

/* Display device name. The rd_code is converted to a
device number by subtracting the first code number
from it and dividing by 2 (shift right one place).
The input/output direction is derived from the
least significant bit of the rd_code. */
printf("%s %s is assigned to - ",
    ct_strct(ct_ldev,(rd_code - CB_CI) >> 1),
    ct_strct(ct_io,((rd_code & 0x01) * 1)));

/* For all physical devices */
for (pdevn = 0; pdevn < 16; pdevn++)
{
    /* Check if current physical device is
     assigned by AND-ing with a 1-bit shifted left pdevn times */
    if (rd_val & (1 << pdevn))  /* Is device active? */
    {
        /* Display physical device name */
        printf("%s",ct_strct(ct_pdev,pdevn));
    }
}
}
chk_use(argc)  /* Check for correct usage */
int argc;    /* Argument count on command line */
{
    if (argc == 1)
    {
        printf("ASSIGN sets the Input/Output redirection.");
        printf("ASSIGN logical-device INPUT physical-device");
        printf("ASSIGN logical-device OUTPUT physical-device");
        printf("ASSIGN SHOW (to show current assignments)");
        printf("%s",ct_strct(ct_ldev));
        printf("%s",ct_strct(ct_pdev));
        exit();
    }
}
```

Figure 11-11. (Continued)

**DATE — Set the System Date**

The DATE utility shown in Figure 11-12 sets the system date in the configuration
block, along with a flag that indicates that the DATE utility has been used.
Other utility programs can use this flag as a primitive test of whether the system
date is current.

Here is an example console dialog:

```text
P3B>date<CR>
DATE Vn 1.0 02/18/83
DATE sets the system date. Usage is:
    DATE mm/dd/yy
    DATE SHOW (to display current date)

P3B>date show<CR>
DATE Vn 1.0 02/18/83
Current Date is 12/18/82

P3B>date 2/23/83<CR>
DATE Vn 1.0 02/18/83
Current Date is 02/23/83
```
#define VN "\n DATE Vn 1.0 02/18/83"

/* This utility accepts the current date from the command tail, validates it, and sets the internal system date in the BIOS. Alternatively, it can be requested just to display the current system date. */
#include <LIBRARY.H>

char *date; /* Pointer to the date in the config. block */
char *date_flag; /* Pointer to date-set flag */
int mm, dd, yy; /* Variables to hold month, day, year */
int mcount; /* Match count of numeric values entered */
int count; /* Count used to add leading 0's to date */

main(argc, argv[])
char argv[1];
{ /* Pointer to date */
  if (argc != 2) /* Check if help requested (or needed) */
    show_use(); /* Display correct usage and exit */
  if (strcmp(argv[1], "SHOW") == 0) /* Check if not SHOW option */
    if (mcount = sscanf(argv[1], "%d/%d/%d", &mm, &dd, &yy))
      if (mcount != 3) /* Input not numeric */
        show_use(); /* Display correct usage and exit */
    if (mm > 12 || mm < 1) /* Check valid month, day, year */
      if (dd > 31 || dd < 1)
        if (yy > 90 || yy < 83) /* <=-- NOTE! */
          printf("\n Month - Xd is illegal.", mm);
          show_use(); /* Display correct usage and exit */
      printf("\n Day = Yd is illegal.", dd);
      show_use(); /* Display correct usage and exit */
    printf("\n Year - Yd is illegal.", yy);
    show_use(); /* Display correct usage and exit */
    /* Convert integers back into a formatted string */
    sprintf(date, "%d/%d/%d", mm, dd, yy);
    date[8] = '0'; /* New string terminator */
    date[9] = '0'; /* New string terminator */
    date[10] = '0';
    /* Change "1/2/3" into "01/02/03" */
    for (count = 0; count < 7; count++)
      if (date[count] == ' ')
        date[count] = '0';
    /* Turn flag on to indicate that user has set date */
    date_flag = DATE_SET;
  printf("\n \n Current Date is Xs", date);
}

show_use() /* Display correct usage and exit */
{ /* PRINTS */
  printf("\n VNDATE sets the system date. Usage is ");
  printf("\n DATE mm/dd/yy
 DATE SHOW (to display current date) ");
  exit();
}

Figure 11-12. DATE.C, a utility that makes the current date part of the system
TIME — Set the System Time

The TIME utility shown in Figure 11-13 sets the current system time. Like DATE, TIME sets a flag so that other utilities can test that the system time is likely to be current.

Here is an example console dialog:

P3B>time<CR>
TIME Vn 1.0 02/18/83
TIME sets the system time. Usage is:
TIME hh:mm:ss
TIME SHOW (to display current time)

P3B>time show<CR>
TIME Vn 1.0 02/18/83
Current Time is 13:08:44

P3B>time 5:47<CR>
TIME Vn 1.0 02/18/83
Current Time is 05:47:00

#define VN "\nTIME Vn 1.0 02/18/83"

/* This utility accepts the current time from the command tail.
   validates it, and sets the internal system time in the BIOS.
   Alternatively, it can just display the current system time. */
#include <LIBRARY.H>

char *time;
char *time_set;
int hh,mm,ss;
int count;
char argv[3];

main(argc,argv)
char argv[];
{   printf(VN);               /* Display sign-on message */
    time = get_cbe(CB_TIME);  /* Set pointer to time */
    time_set = get_cbe(CB_DTFLAGS); /* Set pointer to the */
    /* time-set flag */
    hh = mm = ss = 0;           /* Initialize the time if seconds or */
    minutes are not specified */
    if (argc != 2)             /* Check if help requested (or needed) */
        show_use();           /* Display correct usage and exit */
    if (strstrp(argv[1], "SHOW")) /* Check if not SHOW option */
    {   /* Convert time into hours, minutes, seconds */
        count = sscanf(argv[1], "%d:%d:%d", &hh, &mm, &ss);
        if (count > 3)           /* Input not numeric */
            show_use();           /* Display correct usage and exit */
        if (hh > 12)             /* Check valid hours, minutes, seconds */
        {   printf("\n007Hours = Xd is illegal.", hh);
            show_use();           /* Display correct usage and exit */
        }
    }
}

Figure 11-13. TIME.C, a utility that makes the current time part of the system
if (mm > 59)
    printf("\n\007Minutes = %d is illegal.", mm);
    show_use(); // Display correct usage and exit */
}  
if (ss > 59)
    show_use(); // Display correct usage and exit */
    printf("\n\007Seconds = %d is illegal.", ss);

/* Convert integers back into formatted string */
sprintf(time, "%2d:%2d:%2d", hh, mm, ss);
if (time == 0x0)
    /* Terminate with line feed */
time[9] = '\0';  /* New string terminator */
    /* Convert " 1: 2: 3" into "01:02:03" */
for (count = 0; count < 7; count += 3)
    if (time[count] == ' ')  
        time[count] = '0';
    /* Turn bit on to indicate that the time has been set */
    time_flag |= TIME_SET;

printf("\n\007Current Time is %s", time);
}
/* Display correct usage and exit */
show_use();
    printf("\nTIME sets the system time. Usage is :");
    printf("\nTIME hh:mm:ss")
    printf("\nTIME SHOW (to display current time)\n");
    exit();
}

Figure 11-13. TIME.C, a utility that makes the current time part of the system (continued)

**FUNKEY — Set the Function Keys**

The FUNKEY utility shown in Figure 11-14 sets the character strings associated with specific function keys. In the specified character string, the character "<" is converted into a LINE FEED character. Here is an example console dialog:

```
P3B>funkey<CR>
FUNKEY sets a specific function key string.
FUNKEY key-number "string to be programmed"  
   (Note: '<' is changed to line feed.)  
   (key-number is from 0 to 17.)  
   (string can be up to 16 chars.)
   FUNKEY SHOW  
   (displays settings for all keys)
```

```
P3B>funkey show<CR>
FUNKEY Vn 1.0 02/18/83
   Key #0 = 'Function Key 1'
   Key #1 = 'Function Key 2'
```

```
P3B>funkey 0 "PIP B:=A;=[V]<"<CR>
P3B>funkey show<CR>
FUNKEY Vn 1.0 02/18/83
   Key #0 = 'PIP B:=A;=[V]<'
   Key #1 = 'Function Key 2'
```
```c
#define VN "nFUNKEY Vn 1.0 02/18/83"

#include <LIBRARY.H>

int fnum;        /* Function key number to be programmed */
char fstring[20]; /* String for function key */
struct _fkt *pfk; /* Pointer to function key table */

main(argc, argv)
int argc;
char argv[];
{
    if (argc == 1 || argc > 3)
        show_use();

    pfk = get_cba(CB_FKT); /* Set pointer to function key table */
    if (strcmp(argv[1], "SHOW") == 0)
        show_use();

    fnum = atoi(argv[1]); /* Convert function key number */
    if (fnum > FK_ENTRIES)
        printf("Function key number %d too large.", fnum);
    show_use();

    if (get_fs(fstring) > FK_LENGTH)
        printf("Function key string is too long.");
    show_use();

    pfk += fnum; /* Update pointer to string */
    /* Copy string into function key table */
    if (!pfk->fk_input) /* Check if function key input present */
        printf("Error: Function Key #%d is not set up to be programmed.", fnum);
    show_use();

    strcpy(pfk->fk_output, fstring);
    /* SHOW function specified */

    get_fs(string); /* Get function string from command tail */
    char string[];  /* Pointer to character string */
    char *tail;     /* Pointer to command tail */
    short tcount;   /* Count of TOTAL character in command tail */
    int slen;      /* String length */

    tail = 0x80;    /* Command line is in memory at 0080H */
    tcount = *tail++; /* Set TOTAL count of characters in command tail */
    slen = 0;       /* Initialize string length */

    while(tcount--) /* For all characters in the command tail */
        if (*tail++ == '"') /* Scan for first quotes */
            break;
}

Figure 11-14. FUNKEY.C, a utility that sets the character strings associated with specific function keys.
if (!tcount)  /* No quotes found */
    printf("\n\007No leading quotes found.");
    show_use();
}

++tcount;    /* Adjust tail count */
while(tcount--) /* For all remaining characters in tail */
{
    if (#tail == '"')
        {                      /* Add terminator */
            string[strlen] = '\\0';
            break;                   /* Exit from loop */
        }
    string[strlen] = #tail++; /* Move char. from tail into string */
    if (#string[strlen] == '<' )
        {                      /* Move char. into string */
            string[strlen] = 'xOA';
            ++strlen;
        }
    if (!tcount)  /* No terminating quotes found */
        {                 /* No terminating quotes found */
            printf("\n\007No trailing quotes found.");
            show_use();
        }
    return strlen;        /* Return string length */
}

show_funct()   /* Display settings for all function keys */
{
    struct fkt *pfkt;      /* Local pointer to function keys */
    int count;             /* Count to access function keys */
    char *If;              /* Pointer to "<" character (LINE FEED) */
    pfkt = get_cba(CB_FKT); /* Set pointer to function key table */
    for (count = 0; count < FK_ENTRIES; count++)
        {
            if (pfkt -> fk_input[0]) /* Key is programmed */
                {                      /* Key is programmed */
                    /* Check if at physical end of table */
                    if (pfkt -> fk_input == OxFF)    /* Yes -- break out of for loop */
                        strcpy(fstring,pfkt -> fk_output);    /* Convert all 0xOA chars to "<" */
                        while (If = strncsc(fstring,"\012"))
                            {                                 /* If string can be up to 32 chars. */
                                ++pfkt;                           /* Move to next entry */
                            }
                        printf("\n\007Key 0%ld = 'x%02x',count,fstring);
                    ++pfkt;                        /* Move to next entry */
                }
        }
    printf("\nFUNKEY sets a specific function key string.");
    printf("\nFUNKEY key-number \042string to be programmed\042");
    printf("\nIt (Note: '<' is changed to line feed.)");
    printf("\nIt (key-number is from 0 to %d.)",FK_ENTRIES-1);
    printf("\nIt (string can be up to %d chars.)",FK_LENGTH);
    printf("\nFUNKEY SHOW (displays settings for all keys)"英特尔,exit());
}
Other Utilities

Because of space limitations, not all of the possible utility programs for the BIOS features can be shown in this chapter. Others that would need to be developed in order to have a complete set are

PUBLIC/PRIVATE
This pair of utilities would turn the public files flag on or off, making the files in user 0 available from other user numbers or not, respectively.

SETTERM
This program would program the CONOUT escape table, setting the various escape sequences as required. It could also program the characters in the function key table that match with those emitted by the terminal currently in use.

SAVESYS
This utility would save the current settings in the long term configuration block.

LOADSYS
This would load the long term configuration block from a previously saved image.

DO
This utility would copy the command tail into the multi-command buffer, changing "\" into LINE FEED, and then set the forced input pointer to the multi-command buffer. As a result, characters from the multi-command buffer would be fed into the console input stream as though they had been typed one command at a time.

SPARE
This utility would work in conjunction with the hard-disk bad-sector management in your disk drivers. It would spare out bad sectors or tracks on the hard disk. This done, all subsequent references to the sectors or tracks would be redirected to a different part of the disk.
This chapter lists the error messages that emanate from standard CP/M and its utility programs. It does not include any error messages from the BIOS; these messages, if any, are the individualized product of the programmers who wrote the various versions of the BIOS.

The error messages are shown in alphabetical order, followed (in parentheses) by the name of the program or CP/M component outputting the message. Messages are shown in uppercase even if the actual message you will see contains lowercase letters. Additional characters that are displayed to “pretty up” the message have been omitted. For example, the message “** ABORTED **” will be listed as “ABORTED”.

Following each message is an explanation and, where possible, some information to help you deal with the error.

The last section of the chapter deals with known errors or peculiarities in CP/M and its utilities. Read this section so that you will recognize these problems when they occur.
Error Messages Displayed

? (CCP)

The CCP displays a question mark if you enter a command name and there is no corresponding "command.COM" file on the disk.

It is also displayed if you omit the number of pages required as a parameter in the SAVE command.

? (DDT)

DDT outputs a question mark under several circumstances. You must use context (and some guesswork) to determine what has gone wrong. Here are some specific causes of problems:

- DDT cannot find the file that you have asked it to load into memory. Exit from DDT and investigate using DIR or STAT (the file may be set to System status and therefore invisible with DIR).

- There is a problem with the data in the HEX file that you have asked DDT to load. The problem could be a bad check-sum on a given line or an invalid field somewhere in the record. Try typing the HEX file out on a console, or use an editor to examine it. It is rare to have only one or two bad bits or bytes in a HEX file; large amounts of the file are more likely to have been corrupted. Therefore, you may be able to spot the trouble fairly readily. If you have the source code for the program, reassemble it to produce another copy of the HEX file. If you do not have the source code, there is no reliable way around this problem unless you are prepared to hand-create the HEX file—a difficult and tedious task.

- DDT does not recognize the instruction you have entered when using the "A" (assemble) command to convert a source code instruction into hexadecimal. Check the line that you entered. DDT does not like tabs in the line (although it appears to accept them) or hexadecimal numbers followed by "H". Check that the mnemonic and operands are valid, too.

?? = (DDT)

This cryptic notation is used by DDT when you are using the "L" (list disassembled) command to display some part of memory in DDT's primitive assembly language form. DDT cannot translate all of the 256 possible values of a byte. Some of them are not used in the 8080 instruction set. When DDT encounters an untranslatable value, it displays this message as the instruction code, followed by the actual value of the byte in hexadecimal.

You will see this if you try to disassemble code written for the Z80 CPU, which
Chapter 12: Error Messages

uses unassigned 8080 instructions. You will also see it if you try to disassemble bytes that contain ASCII text strings rather than 8080 instructions.

**ABORTED (STAT)**

If you enter any keyboard character while STAT is working its way down the file directory setting files to $DIR (Directory), $SYS (System), $R/W (Read/Write), or $R/O (Read-Only) status, then it will display this message, stop what it is doing, and execute a warm boot.

By contrast, if you enter the command

```
A>stat *.*<cr>
```

to display all of the files on a disk, there is no way that the process can be aborted.

**ABORTED (PIP)**

This message is displayed if you press any keyboard character while PIP is copying a file to the list device.

**BAD DELIMITER (STAT)**

If your BIOS uses the normal IOBYTE method of assigning physical devices to logical devices, you use STAT to perform the assignment. The command has this format:

```
STAT RDR:=PTR:
```

STAT displays this message if it cannot find the “=” in the correct place.

**BAD LOAD (CCP)**

This is probably the most obscure error message that emanates from CP/M. You will get this message if you attempt to load a COM file that is larger than the transient program area. Your only recourse is to build a CP/M system that has a larger TPA.

**BAD PARAMETER (PIP)**

PIP accepts certain parameters in square brackets at the end of the command line. This message is displayed if you enter an invalid parameter or an illegal numeric value following a parameter letter.

**BDOS ERROR ON d: BAD SECTOR (BDOS)**

The BDOS displays this message if the READ and WRITE functions in your BIOS ever return indicating an error. The only safe response to this message is to type CONTROL-C. CP/M will then execute a warm boot. If you type CARRIAGE RETURN, the error will be ignored—with unpredictable results.
A well-implemented BIOS should include disk error recovery and control so that the error will never be communicated to the BDOS. If the BIOS gives you the option of ignoring an error, do so only when you are reasonably sure of the outcome or have adequate backup copies so that you can recreate your files.

**BDOS ERROR ON d: FILE R/O (BDOS)**

You will see this message if you attempt to erase (ERA) a file that has been set to Read-Only status. Typing any character on the keyboard causes the BDOS to perform a warm boot operation. Note that the BDOS does not tell you which file is creating the problem. This can be a problem when you use ambiguous file names in the ERA command. Use the STAT command to display all the files on the disk; it will tell you which files are Read-Only.

This message is also displayed if a program tries to delete a Read-Only file. Again, it can be difficult to determine which file is causing the problem. Your only recourse is to use STAT to try to infer which of the Read-Only files might be causing the problems.

**BDOS ERROR ON d: R/O (BDOS)**

This looks similar to the previous message, but it refers to an entire logical disk instead of a Read-Only file. However, it is rarely output because you have declared a disk to be Read-Only. Usually, it occurs because you changed diskettes without typing a CONTROL-C; CP/M will detect the new diskette and, without any external indication, will set the disk to Read-Only status.

If you or a program attempts to write any data to the disk, the attempt will be trapped by the BDOS and this message displayed. Typing any character on the keyboard causes a warm boot—then you can proceed.

**BDOS ERROR ON d: SELECT (BDOS)**

The BDOS displays this message if you or a program attempts to select a logical disk for which the BIOS lacks the necessary tables. The BDOS uses the value returned by SELDSK to determine whether a logical disk “exists” or not.

If you were trying to change the default disk to a nonexistent one, you will have to press the RESET button on your computer. There is no way out of this error.

However, if you were trying to execute a command that accessed the nonexistent disk, then you can type a CONTROL-C and CP/M will perform a warm boot.

**BREAK x AT y (ED)**

This is another cryptic message whose meaning you cannot guess. The list that follows explains the possible values of “x.” The value “y” refers to the command ED was executing when the error occurred.
**Meaning**

- x  Search failure. ED did not find the string you asked it to search for.
- #  Unrecognized command.
- ?  File not found.
- >  ED's internal buffer is full.
- E  Command aborted.
- F  Disk or directory full. You will have to determine which is causing the problem.

**CANNOT CLOSE, READ/ONLY? (SUBMIT)**

SUBMIT displays this message if the disk on which it is trying to write its output file, "$$.SUB", is physically write protected. Do not confuse this with the disk being *logically* write protected.

The standard version of SUBMIT writes the output file onto the current default disk, so if your current default disk is other than drive A:, you may be able to avoid this problem if you switch the default to A: and then enter a command of the form:

```
A>submit blsubfile<cr>
```

**CANNOT CLOSE DESTINATION FILE (PIP)**

PIP displays this message if the destination disk is physically write protected. Check the destination disk. If it is write protected, remove the protection and repeat the operation.

If the disk is not protected, you have a hardware problem. The directory data written to the disk is being written to the wrong place, even the wrong disk, or is not being recorded on the medium.

**CANNOT CLOSE FILES (ASM)**

ASM displays this message if it cannot close its output files because the disk is physically write protected, or if there is a hardware problem that prevents data being written to the disk. See the paragraph above.

**CANNOT READ (PIP)**

PIP displays this message if you attempt to read information from a logical device that can only output. For example:

```
A>pip diskfile=L$T1<cr>
```

PIP also will display this message if you confuse it sufficiently, as with the following instruction:

```
A>pip file1=file2;file3<cr>
```
The CP/M Programmer’s Handbook

CANNOT WRITE (PIP)

PIP displays this message if you attempt to output (write) information to a logical device that can only be used for input, such as the RDR: (reader, the anachronistic name for the auxiliary input device).

CHECKSUM ERROR (LOAD)

LOAD displays this message if it encounters a line in the input HEX file that does not have the correct check sum for the data on the line.

LOAD also displays information helpful in pinpointing the problem:

CHECKSUM ERROR
LOAD ADDRESS 0110 <- First address on line in file
ERROR ADDRESS 0112 <- Address of next byte to be loaded
BYTES READ:
0110: 00 33 22 2B 02 21 27 02 <- Bytes preceding error

Note that LOAD does not display the check-sum value itself. Use TYPE or an editor to inspect the HEX file in order to see exactly what has gone wrong.

CHECKSUM ERROR (PIP)

If you ask PIP to copy a file of type HEX, it will check each line in the file, making sure that the line’s check sum is valid. If it is not, PIP will display this message. Unfortunately, PIP does not tell you which line is in error—you must determine this by inspection or recreate the HEX file and try again.

COMMAND BUFFER OVERFLOW (SUBMIT)

SUBMIT displays this message if the SUB file you specified is too large to be processed. SUBMIT’s internal buffer is only 2048 bytes. You must reduce the size of the SUB file; remove any comment lines, or split it into two files with the last line of the first file submitting the second to give a nested SUBMIT file.

COMMAND TOO LONG (SUBMIT)

The longest command line that SUBMIT can process is 125 characters. There is no way around this error other than reducing the length of the offending line. You will have to find this line by inspection—SUBMIT does not identify the line.

One way that you can remove a few characters from a command line is to rename the COM file you are invoking to a shorter name, or use abbreviated names for parameters if the program will accept these.

CORRECT ERROR, TYPE RETURN OR CTL-Z (PIP)

This message is a carryover from the days when PIP used to read hexadecimal data from a high-speed paper tape reader. If PIP detected the end of a physical roll
of paper tape, it would display this message. The user could then check to see if the paper tape had torn or had really reached its end. If there was more tape to be read, the user could enter a CARRIAGE RETURN to resume reading tape or enter a CONTROL-Z to serve as the end-of-file character.

Needless to say, it is unlikely that you will see this message if you do not have a paper tape reader.

**DESTINATION IS R/O, DELETE (Y/N)? (PIP)**

PIP displays this message if you try to overwrite a disk file that has been set to Read-Only status. If you type “Y” or “y”, PIP will overwrite the destination file. It leaves the destination file in Read/Write status with its Directory/System status unchanged. Typing any character other than “Y” or “y” makes PIP abandon the copy and display the message

**NOT DELETED**

You can avoid this message altogether if you specify the “w” option on PIP’s command line. For example:

```
A>pip destfile=srcfile[w]<cr>
```

PIP will then overwrite Read-Only files without question.

**DIRECTORY FULL (SUBMIT)**

This message is displayed if the BDOS returns an error when SUBMIT tries to create its output file, “$$$.SUB”. As a rough and ready approximation, use “STAT *.*” to see how many files and extents you have on the disk. Erase any unwanted ones. Then use “STAT DSK:” to find out the maximum number of directory entries possible for the disk.

You may also see this message if the file directory has become corrupted or if the disk formatting routine leaves the disk with the file directory full of some pattern other than E5H.

You can assess whether the directory has been corrupted by using “STAT USR:”. STAT then displays which user numbers contain files. If the directory is corrupt, you will normally see user numbers greater than 15.

It is not easy to repair a corrupted directory. “ERA *.*” erases only the files for the current user number, so you will have to enter the command 16 times, once for each user number from 0 to 15. Alternatively, you can reformat the disk.

**DISK OR DIRECTORY FULL (ED)**

Self-explanatory.
DISK READ ERROR (PIP)
DISK WRITE ERROR (SUBMIT)
DISK WRITE ERROR (PIP)

These messages will normally be preceded by a BIOS error message. They will only be displayed if the BIOS returns indicating an error. As was described earlier, this is unlikely if the BIOS has any kind of error recovery logic.

END OF FILE, CTL-Z? (PIP)

PIP displays this message if, while copying a HEX file, it encounters a CONTROL-Z (end of file). Again, the underlying idea is based on the concept of physical paper tape. When you saw this message, you could look at the tape in the reader, and if it really was at the end of the roll, enter a CONTROL-Z on the keyboard to terminate the file. Given any other character, PIP would read the next piece of tape.

ERROR: CANNOT CLOSE FILES (LOAD)

LOAD displays this message if you have physically write protected the disk on which it is trying to write the output COM file.

ERROR: CANNOT OPEN SOURCE (LOAD)

LOAD displays this message if it cannot open the HEX file that you specified in the command tail.

ERROR: DISK READ (LOAD)
ERROR: DISK WRITE (LOAD)

These two messages would normally be preceded by a BIOS error message. If your BIOS includes disk error recovery, you would not normally see these messages; the error would have been handled by the BIOS.

ERROR: INVERTED LOAD ADDRESS (LOAD)

LOAD displays this message if it detects a load address less than 0100H in the input HEX file. It also displays the actual address input from the file, so you can examine the HEX file looking for this address to determine the likely cause of the problem.

Note that DDT, when asked to load the same HEX file, will do so without any error—and will probably damage the contents of the base page in so doing.

ERROR: NO MORE DIRECTORY SPACE (LOAD)

Self-explanatory.
Chapter 12: Error Messages

ERROR ON LINE N (SUBMIT)

SUBMIT displays this message if it encounters a line in the SUB file that it does not know how to process. Most likely you have a file that has type .SUB but does not contain ASCII text.

The first line of the SUB file is number 001.

FILE EXISTS (CCP)

The CCP displays this message if you attempt to use the REN command to rename an existing file to a name already given to another file.

Use “STAT *.” to display all of the files on the disk. DIR will show only those files that have Directory status, and you may not be able to see the file causing the problem.

FILE IS READ/ONLY (ED)

ED displays this message if you attempt to edit a file that has been set to Read-Only status.

FILE NOT FOUND (STAT)
FILENAME NOT FOUND (PIP)

STAT and PIP display their respective messages if you specify a nonexistent file. This applies to both specific and ambiguous file names.

INVALID ASSIGNMENT (STAT)

STAT can be used to assign physical devices to logical devices using the IOBYTE system described earlier. It will display this message if you enter an illogical assignment. Use the “STAT VAL:” command to display the valid assignments.

INVALID CONTROL CHARACTER (SUBMIT)

SUBMIT is supposed to be able to handle a control character in the SUB file—the notation being “^x”, where “x” is the control letter. In fact, the standard release version of SUBMIT cannot handle this notation. A patch is available from Digital Research to correct this problem.

Given that this patch has been installed, SUBMIT will display this message if a character other than “A” to “Z” is specified after the circumflex character.

INVALID DIGIT (PIP)

PIP displays this message if it encounters non-numeric data where it expects a numeric value.
INVALID DISK ASSIGNMENT (STAT)

STAT displays this message if you try to set a logical disk to Read-Only status and you specify a parameter other than "R/O." Note that there is no leading "$" in this case (as there is when you want to set a file to Read-Only).

INVALID DRIVE NAME (USE A, B, C, OR D) (SYSGEN)

SYSGEN displays this message if you attempt to load the CP/M system from, or write the system to, a disk drive other than A, B, C, or D.

INVALID FILE INDICATOR (STAT)

STAT outputs this message if you specify an erroneous file attribute. File attributes can only be one of the following:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$DIR</td>
<td>Directory</td>
</tr>
<tr>
<td>$SYS</td>
<td>System</td>
</tr>
<tr>
<td>$R/O</td>
<td>Read-Only</td>
</tr>
<tr>
<td>$R/W</td>
<td>Read/Write</td>
</tr>
</tbody>
</table>

INVALID FORMAT (PIP)

PIP displays this message if you enter a badly formatted command; for example, a "+" character instead of an "=" (on some terminals these are on the same key).

INVALID HEX DIGIT (LOAD)

LOAD displays this message if it encounters a nonhexadecimal digit in the input HEX file, where only a hex digit can appear. LOAD then displays additional information to tell you where in the file the problem occurred:

```
INVALID HEX DIGIT
LOAD ADDRESS 0110 <- First address on line in file
ERROR ADDRESS 0112 <- Address of byte containing non-hex
BYTES READ:
0110:
0110: 00 33 <- Bytes preceding error
```

INVALID MEMORY SIZE (MOVCPM)

MOVCPM displays this message if you enter an invalid memory size for the CP/M system size you want to construct.

INVALID SEPARATOR (PIP)

PIP displays this message if you try to concatenate files using something other than a comma between file names.
INVALID USER NUMBER (PIP)

PIP displays this message if you enter a user number outside the range 0 to 15 with the “[gn]” option (where “n” is the user number).

NO ‘SUB’ FILE PRESENT (SUBMIT)

SUBMIT displays this message if it cannot find a file with the file name that you specified and with a type of .SUB.

NO DIRECTORY SPACE (ASM)
NO DIRECTORY SPACE (PIP)

Self-explanatory.

NO FILE (CCP)

The CCP displays this message if you use the REN (rename) command and it cannot find the file you wish to rename.

NO FILE (PIP)

PIP displays this message if it cannot find the file that you specified.

NO MEMORY (ED)

ED displays this message if it runs out of memory to use for storing the text that you are editing.

NO SOURCE FILE ON DISK (SYSGEN)

This error message is misleading. SYSGEN does not read source code files. The message should read “INPUT FILE NOT FOUND”.

NO SOURCE FILE PRESENT (ASM)

In this case, ASM really does mean that the source code file cannot be found. Remember that ASM uses a strange form of specifying its parameters. ASM uses the file name that you enter and then searches for a file of that name, but with file type .ASM. The three characters of the file type that you specify are used to represent the logical disks on which the source, hex, and list files, respectively, are to be placed.

NO SPACE (CCP)

The CCP displays this message if you use the SAVE command and there is insufficient room on the disk to accommodate the file.
NOT A CHARACTER SOURCE (PIP)

PIP displays this message if you attempt to copy characters from a character output device, such as the auxiliary output device (known to PIP as PUN:).

OUTPUT FILE WRITE ERROR (ASM)

ASM will display this message if the BDOS returns an error from a disk write operation. If your BIOS has disk error recovery logic, you should never see this message.

PARAMETER ERROR (SUBMIT)

SUBMIT uses the “$” to mark points where parameter values are to be substituted. If you have a single “$” followed by an alphabetic character, SUBMIT will display this message. Use “$$” to represent a real “$”.

PERMANENT ERROR, TYPE RETURN TO IGNORE (SYSGEN)

SYSGEN displays this message if the BIOS returns an error from a disk read or write operation. If your BIOS has disk error recovery logic, you should never see this message.

QUIT NOT FOUND (PIP)

PIP displays this message when it cannot find the string specified in the “[Qcharacter string^Z]” option, meaning “Quit copying when you encounter this string.”

READ ERROR (CCP)

The CCP displays this message if the BIOS returns an error from a disk read or write operation. If your BIOS includes disk error recovery logic, you should not see this error message.

RECORD TOO LONG (PIP)

PIP displays this message if it encounters a line longer than 80 characters while copying a HEX file. Inspect the HEX file using the TYPE command or an editor.

REQUIRES CP/M 2.0 OR NEWER FOR OPERATION (PIP)
REQUIRES CP/M VERSION 2.0 OR LATER (XSUB)

Self-explanatory.
SOURCE FILE INCOMPLETE (SYSGEN)

SYSGEN displays this message if the file that you have asked it to read is too short. Use STAT to check the length of the file.

SOURCE FILE NAME ERROR (ASM)

ASM displays this message if you specify an ambiguous file name: that is, one that contains either "*" or "?".

SOURCE FILE READ ERROR (ASM)

ASM displays this message if it encounters problems reading the input source code file. Check the input file using the TYPE command or an editor.

START NOT FOUND (PIP)

PIP displays this message when it cannot find the string specified in the "[Scharacter string^Z]" option, meaning "Start copying when you encounter this string."

SYMBOL TABLE OVERFLOW (ASM)

ASM displays this message when you have too many symbols in the source code file. Your only recourse is to split the source file into several pieces and arrange for ORG (origin) statements to position the generated object code so that the pieces fit together.

SYNCRONIZATION ERROR (MOVCPM)

Apart from the spelling error, this message is designed to be cryptic. MOVCPM displays it when the Digital Research serial number embedded in MOVCPM does not match the serial number in the version of CP/M that you are currently running.

SYSTEM FILE NOT ACCESSIBLE (ED)

ED displays this message if you attempt to edit a file that has been set to System status. Use STAT to set the file to Directory status.

TOO MANY FILES (STAT)

STAT displays this message if there is insufficient memory available to sort and display all of the files on the specified disk. Try limiting the number of files it has to sort by judicious use of ambiguous file names.

UNRECOGNIZED DESTINATION (PIP)

PIP displays this message if you specify an "illegal" destination device.
VERIFY ERROR (PIP)

If you use the "[v]" (verify) option of PIP when copying to a disk file, PIP will write a sector to the disk, read it back, and compare the data. PIP displays this message if the data does not match.

If there is a problem with your disk system, you should have seen some form of disk error message preceding this one. If there is no preceding message, then you have a problem with the main memory on your system.

Wrong CP/M Version (Requires 2.0) (STAT)

Self-explanatory.

(XSUB ACTIVE) (XSUB)

This is not really an error message, but you may mistake it for one. XSUB is the eXtended SUBMIT program. Without it, SUBMIT can only feed command lines to the Console Command Processor. XSUB allows character-by-character input into any program that uses the BDOS to read console input.

XSUB is initiated by being the first command in a SUB file. Once initiated it stays in memory until the end of the SUB file has been reached. Until that happens, XSUB will output this message every time a warm boot occurs as a reminder that it is still in memory.

XSUB Already Present (XSUB)

XSUB will display this message if it is already active and you attempt to load it again.

Miscellaneous Errors

This section deals with errors that are not accompanied by any error message. It is included here to help you recognize a problem after it has already occurred. The errors are shown grouped by product.

ASM: Fails to Detect Unterminated IF Clause

If you use the IF pseudo-operation, it must be followed by a matching ENIF. ASM fails to detect the case that the end of the source file is encountered before the ENIF.

If the condition specified on the IF line is false, you could have a situation in which ASM would ignore the majority of the source file without comment.
**ASM: Creates HEX File That Cannot Be Loaded**

If you omit the ORG statement at the front of a source file, ASM will assemble the code origined at location 0000H. This file will crash the system if you try to load it with DDT. The message “ERROR: INVERTED ADDRESS” will be shown from LOAD.

**CP/M: Signs On and Then Dies Without A > Prompt**

After the BIOS has signed on, it transfers control to the Console Command Processor. The CCP then attempts to log in the system disk, reading the file directory and building the allocation vector. If your file directory has been badly corrupted, it can cause the system to crash. Use another system disk and try to display the directory on the bad disk.

**DDT: Loads HEX File and Then Crashes the System**

DDT does not check the addresses specified in a HEX file. If you have forgotten to put an ORG statement at the front of the source file, or more subtly, if your source program has “wrapped around” by having addresses up at 0FFFFH and “above,” the assembler will start assembling at 0000H again.

**DIR: Shows Odd-Looking File Names**

If you have odd-looking file names, or the vertical lines of “:” that DIR uses to separate the file names are misaligned, then the file directory has been corrupted. One strategy is to format a new disk, copy all of the valid files to it, and discard the corrupted disk.

**DIR: Shows More than One Entry with the Same Name**

This can happen if you use a program that creates a new file without asking the BDOS to delete any existing files of the same name. It can also happen if you use the custom MOVE utility carelessly.

To remedy the situation proceed as follows:

- Use PIP to copy the specific file to another disk. Do not use an ambiguous file name; specify the duplicated file name exactly. PIP will copy the first instance of the file it encounters in the directory.
- Use the ERA command to erase the duplicated file. *This will erase both copies of the file.*
- Use PIP to copy back the first instance of the file.
STAT: User Numbers > 15

If you use the "STAT USR:" command to display which user numbers contain active files, and user numbers greater than 15 are displayed, then the file directory on the disk has been corrupted.

Use PIP to copy the valid files from legitimate user numbers, and then discard the corrupted disk.

SUBMIT: Fails to Start Submit Procedure

There are several reasons why SUBMIT will not initiate a SUB file:

- You are using the standard release version of SUBMIT and your current default disk is other than drive A:. SUBMIT builds its "$$.SUB" file on the default disk, but the CCP only looks on drive A: for "$$.SUB". Use the following procedure to modify SUBMIT to build its "$$.SUB" file on drive A:

```
A> DDT SUBMIT.COM<cr>
DDT VERS 2.2
NEXT PC
0600 0100
-$$bb <- Change 5bb
05BB 01 00<cr> <- from 00 (default drive)
05BC 24 <cr> to 01 (drive A:)
-^c
A>SAVE 5 SUBMIT.COM<cr>
A>_
```

- If you forgot to terminate the last line of the SUB file with a CARRIAGE RETURN.
- If your SUB file contains a line with nothing but a CARRIAGE RETURN on it (that is, a blank line).
The American Standard Code for Information Interchange (ASCII) consists of a set of 96 displayable characters and 32 nondisplayed characters. Most CP/M systems use at least a subset of the ASCII character set. When CP/M stores characters on a diskette as text, the ASCII definitions are used.

Several of the CP/M utility programs use the ASCII Character Code. Text created using ED is stored as ASCII characters on diskette. DDT, when displaying a “dump” of the contents of memory, displays both the hexadecimal and ASCII representations of memory’s contents.

ASCII does not use an entire byte of information to represent a character. ASCII is a seven-bit code, and the eighth bit is often used for parity. Parity is an error-checking method which assures that the character received is the one transmitted. Many microcomputers and microcomputer devices ignore the parity bit, while others require one of the following two forms of parity:

**Even Parity**

The number of binary 1’s in a byte is always an even number. If there is an odd number of 1’s in the character, the parity bit will be a 1; if there is an even number of 1’s in the character, the parity bit is made a 0.

**Odd Parity**

The number of binary 1’s in a byte is always an odd number. If there is an
even number of 1’s in the character, the parity bit will be a 1; if there is an odd number of 1’s in the character, the parity bit is made a 0.

Alternative ways of coding the information stored by the computer include the 8-bit EBCDIC (Extended Binary Coded Decimal Interchange Code), used by IBM, and a number of packed binary schemes, primarily used to represent numerical information.

**Table A-1. ASCII Character Codes**

<table>
<thead>
<tr>
<th>b7</th>
<th>b6</th>
<th>b5</th>
<th>b4</th>
<th>b3</th>
<th>b2</th>
<th>b1</th>
<th>Col.</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>NUL</td>
<td>0</td>
<td>@</td>
<td>P</td>
<td>a</td>
<td>p</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>SOH</td>
<td>1</td>
<td>A</td>
<td>Q</td>
<td>a</td>
<td>q</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>DC1</td>
<td>2</td>
<td>B</td>
<td>R</td>
<td>b</td>
<td>r</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>DC2</td>
<td>3</td>
<td>C</td>
<td>S</td>
<td>c</td>
<td>s</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>DC3</td>
<td>4</td>
<td>D</td>
<td>T</td>
<td>d</td>
<td>t</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>NAK</td>
<td>5</td>
<td>E</td>
<td>U</td>
<td>e</td>
<td>u</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>SYN</td>
<td>6</td>
<td>F</td>
<td>V</td>
<td>f</td>
<td>v</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>ETB</td>
<td>7</td>
<td>G</td>
<td>W</td>
<td>g</td>
<td>w</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>CAN</td>
<td>8</td>
<td>H</td>
<td>X</td>
<td>h</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>EM</td>
<td>9</td>
<td>I</td>
<td>Y</td>
<td>i</td>
<td>y</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>SUB</td>
<td>10</td>
<td>J</td>
<td>Z</td>
<td>j</td>
<td>z</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>ESC</td>
<td>11</td>
<td>K</td>
<td>M</td>
<td>k</td>
<td>m</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>12</td>
<td>12</td>
<td>12</td>
<td>FS</td>
<td>12</td>
<td>L</td>
<td>N</td>
<td>l</td>
<td>n</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>13</td>
<td>13</td>
<td>13</td>
<td>GS</td>
<td>13</td>
<td>M</td>
<td>O</td>
<td>m</td>
<td>o</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>14</td>
<td>14</td>
<td>14</td>
<td>RS</td>
<td>14</td>
<td>N</td>
<td>P</td>
<td>n</td>
<td>p</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>15</td>
<td>15</td>
<td>15</td>
<td>US</td>
<td>15</td>
<td>O</td>
<td>DEL</td>
<td>o</td>
<td>DEL</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **NUL** Null
- **SOH** Start of heading
- **STX** Start of text
- **ETX** End of text
- **EOT** End of transmission
- **ENQ** Enquiry
- **ACK** Acknowledge
- **BEL** Bell or alarm
- **BS** Backspace
- **HT** Horizontal tabulation
- **LF** Line feed
- **VT** Vertical tabulation
- **FF** Form feed
- **CR** Carriage return
- **SO** Shift out
- **SI** Shift in
- **DLE** Data link escape
- **DC1** Device control 1
- **DC2** Device control 2
- **DC3** Device control 3
- **DC4** Device control 4
- **NAK** Negative acknowledge
- **SYN** Synchronous idle
- **ETB** End of transmission block
- **CAN** Cancel
- **EM** End of medium
- **SUB** Substitute
- **ESC** Escape
- **FS** File separator
- **GS** Group separator
- **RS** Record separator
- **US** Unit separator
- **SP** Space
- **DEL** Delete
### Table A-2. ASCII Character Codes in Ascending Order

<table>
<thead>
<tr>
<th>Hexadecimal</th>
<th>Binary</th>
<th>ASCII</th>
<th>Hexadecimal</th>
<th>Binary</th>
<th>ASCII</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>000 0000</td>
<td>NUL</td>
<td>30</td>
<td>011 0000</td>
<td>0</td>
</tr>
<tr>
<td>01</td>
<td>000 0001</td>
<td>SOH</td>
<td>31</td>
<td>011 0001</td>
<td>1</td>
</tr>
<tr>
<td>02</td>
<td>000 0010</td>
<td>STX</td>
<td>32</td>
<td>011 0010</td>
<td>2</td>
</tr>
<tr>
<td>03</td>
<td>000 0011</td>
<td>ETX</td>
<td>33</td>
<td>011 0011</td>
<td>3</td>
</tr>
<tr>
<td>04</td>
<td>000 0100</td>
<td>EOT</td>
<td>34</td>
<td>011 0100</td>
<td>4</td>
</tr>
<tr>
<td>05</td>
<td>000 0101</td>
<td>ENQ</td>
<td>35</td>
<td>011 0101</td>
<td>5</td>
</tr>
<tr>
<td>06</td>
<td>000 0110</td>
<td>ACK</td>
<td>36</td>
<td>011 0110</td>
<td>6</td>
</tr>
<tr>
<td>07</td>
<td>000 0111</td>
<td>BEL</td>
<td>37</td>
<td>011 0111</td>
<td>7</td>
</tr>
<tr>
<td>08</td>
<td>000 1000</td>
<td>BS</td>
<td>38</td>
<td>011 1000</td>
<td>8</td>
</tr>
<tr>
<td>09</td>
<td>000 1001</td>
<td>HT</td>
<td>39</td>
<td>011 1001</td>
<td>9</td>
</tr>
<tr>
<td>0A</td>
<td>000 1010</td>
<td>LF</td>
<td>3A</td>
<td>011 1010</td>
<td>:</td>
</tr>
<tr>
<td>0B</td>
<td>000 1011</td>
<td>VT</td>
<td>3B</td>
<td>011 1011</td>
<td>;</td>
</tr>
<tr>
<td>0C</td>
<td>000 1100</td>
<td>FF</td>
<td>3C</td>
<td>011 1100</td>
<td>&lt;</td>
</tr>
<tr>
<td>0D</td>
<td>000 1101</td>
<td>CR</td>
<td>3D</td>
<td>011 1101</td>
<td>=</td>
</tr>
<tr>
<td>0E</td>
<td>000 1110</td>
<td>SO</td>
<td>3E</td>
<td>011 1110</td>
<td>&gt;</td>
</tr>
<tr>
<td>0F</td>
<td>000 1111</td>
<td>SI</td>
<td>3F</td>
<td>011 1111</td>
<td>?</td>
</tr>
<tr>
<td>10</td>
<td>001 0000</td>
<td>DLE</td>
<td>40</td>
<td>100 0000</td>
<td>A</td>
</tr>
<tr>
<td>11</td>
<td>001 0001</td>
<td>DC1</td>
<td>41</td>
<td>100 0001</td>
<td>B</td>
</tr>
<tr>
<td>12</td>
<td>001 0010</td>
<td>DC2</td>
<td>42</td>
<td>100 0010</td>
<td>C</td>
</tr>
<tr>
<td>13</td>
<td>001 0011</td>
<td>DC3</td>
<td>43</td>
<td>100 0011</td>
<td>D</td>
</tr>
<tr>
<td>14</td>
<td>001 0100</td>
<td>DC4</td>
<td>44</td>
<td>100 0100</td>
<td>E</td>
</tr>
<tr>
<td>15</td>
<td>001 0101</td>
<td>NAK</td>
<td>45</td>
<td>100 0101</td>
<td>F</td>
</tr>
<tr>
<td>16</td>
<td>001 0110</td>
<td>SYN</td>
<td>46</td>
<td>100 0110</td>
<td>G</td>
</tr>
<tr>
<td>17</td>
<td>001 0111</td>
<td>ETB</td>
<td>47</td>
<td>100 0111</td>
<td>H</td>
</tr>
<tr>
<td>18</td>
<td>001 1000</td>
<td>CAN</td>
<td>48</td>
<td>100 1000</td>
<td>I</td>
</tr>
<tr>
<td>19</td>
<td>001 1001</td>
<td>EM</td>
<td>49</td>
<td>100 1001</td>
<td>J</td>
</tr>
<tr>
<td>1A</td>
<td>001 1010</td>
<td>SUB</td>
<td>4A</td>
<td>100 1010</td>
<td>K</td>
</tr>
<tr>
<td>1B</td>
<td>001 1011</td>
<td>ESC</td>
<td>4B</td>
<td>100 1011</td>
<td>L</td>
</tr>
<tr>
<td>1C</td>
<td>001 1100</td>
<td>FS</td>
<td>4C</td>
<td>100 1100</td>
<td>M</td>
</tr>
<tr>
<td>1D</td>
<td>001 1101</td>
<td>GS</td>
<td>4D</td>
<td>100 1101</td>
<td>N</td>
</tr>
<tr>
<td>1E</td>
<td>001 1110</td>
<td>RS</td>
<td>4E</td>
<td>100 1110</td>
<td>O</td>
</tr>
<tr>
<td>1F</td>
<td>001 1111</td>
<td>US</td>
<td>4F</td>
<td>100 1111</td>
<td>P</td>
</tr>
<tr>
<td>20</td>
<td>010 0000</td>
<td>SP</td>
<td>50</td>
<td>101 0000</td>
<td>Q</td>
</tr>
<tr>
<td>21</td>
<td>010 0001</td>
<td>&quot;</td>
<td>51</td>
<td>101 0001</td>
<td>R</td>
</tr>
<tr>
<td>22</td>
<td>010 0010</td>
<td>#</td>
<td>52</td>
<td>101 0010</td>
<td>S</td>
</tr>
<tr>
<td>23</td>
<td>010 0011</td>
<td>$</td>
<td>53</td>
<td>101 0011</td>
<td>T</td>
</tr>
<tr>
<td>24</td>
<td>010 0100</td>
<td>%</td>
<td>54</td>
<td>101 0100</td>
<td>U</td>
</tr>
<tr>
<td>25</td>
<td>010 0101</td>
<td>&amp;</td>
<td>55</td>
<td>101 0101</td>
<td>V</td>
</tr>
<tr>
<td>26</td>
<td>010 0110</td>
<td>'</td>
<td>56</td>
<td>101 0110</td>
<td>W</td>
</tr>
<tr>
<td>27</td>
<td>010 0111</td>
<td>(</td>
<td>57</td>
<td>101 0111</td>
<td>X</td>
</tr>
<tr>
<td>28</td>
<td>010 1000</td>
<td>)</td>
<td>58</td>
<td>101 1000</td>
<td>Y</td>
</tr>
<tr>
<td>29</td>
<td>010 1001</td>
<td>*</td>
<td>59</td>
<td>101 1001</td>
<td>Z</td>
</tr>
<tr>
<td>2A</td>
<td>010 1010</td>
<td>+</td>
<td>5A</td>
<td>101 1010</td>
<td>[</td>
</tr>
<tr>
<td>2B</td>
<td>010 1011</td>
<td>.</td>
<td>5B</td>
<td>101 1011</td>
<td>\</td>
</tr>
<tr>
<td>2C</td>
<td>010 1100</td>
<td>-</td>
<td>5C</td>
<td>101 1100</td>
<td>]</td>
</tr>
<tr>
<td>2D</td>
<td>010 1101</td>
<td>.</td>
<td>5D</td>
<td>101 1101</td>
<td>^</td>
</tr>
<tr>
<td>2E</td>
<td>010 1110</td>
<td>/</td>
<td>5E</td>
<td>101 1110</td>
<td>_</td>
</tr>
<tr>
<td>2F</td>
<td>010 1111</td>
<td>\</td>
<td>5F</td>
<td>101 1111</td>
<td></td>
</tr>
</tbody>
</table>
Table A-2. ASCII Character Codes in Ascending Order (Continued)

<table>
<thead>
<tr>
<th>Hexadecimal</th>
<th>Binary</th>
<th>ASCII</th>
<th>Hexadecimal</th>
<th>Binary</th>
<th>ASCII</th>
</tr>
</thead>
<tbody>
<tr>
<td>60</td>
<td>110 0000</td>
<td></td>
<td>70</td>
<td>111 0000</td>
<td>p</td>
</tr>
<tr>
<td>61</td>
<td>110 0001</td>
<td>a</td>
<td>71</td>
<td>111 0001</td>
<td>q</td>
</tr>
<tr>
<td>62</td>
<td>110 0010</td>
<td>b</td>
<td>72</td>
<td>111 0010</td>
<td>r</td>
</tr>
<tr>
<td>63</td>
<td>110 0011</td>
<td>c</td>
<td>73</td>
<td>111 0011</td>
<td>s</td>
</tr>
<tr>
<td>64</td>
<td>110 0100</td>
<td>d</td>
<td>74</td>
<td>111 0100</td>
<td>t</td>
</tr>
<tr>
<td>65</td>
<td>110 0101</td>
<td>e</td>
<td>75</td>
<td>111 0101</td>
<td>u</td>
</tr>
<tr>
<td>66</td>
<td>110 0110</td>
<td>f</td>
<td>76</td>
<td>111 0110</td>
<td>v</td>
</tr>
<tr>
<td>67</td>
<td>110 0111</td>
<td>g</td>
<td>77</td>
<td>111 0111</td>
<td>w</td>
</tr>
<tr>
<td>68</td>
<td>110 1000</td>
<td>h</td>
<td>78</td>
<td>111 1000</td>
<td>x</td>
</tr>
<tr>
<td>69</td>
<td>110 1001</td>
<td>i</td>
<td>79</td>
<td>111 1001</td>
<td>y</td>
</tr>
<tr>
<td>6A</td>
<td>110 1010</td>
<td>j</td>
<td>7A</td>
<td>111 1010</td>
<td>z</td>
</tr>
<tr>
<td>6B</td>
<td>110 1011</td>
<td>k</td>
<td>7B</td>
<td>111 1011</td>
<td>{</td>
</tr>
<tr>
<td>6C</td>
<td>110 1100</td>
<td>l</td>
<td>7C</td>
<td>111 1100</td>
<td></td>
</tr>
<tr>
<td>6D</td>
<td>110 1101</td>
<td>m</td>
<td>7D</td>
<td>111 1101</td>
<td>}</td>
</tr>
<tr>
<td>6E</td>
<td>110 1110</td>
<td>n</td>
<td>7E</td>
<td>111 1110</td>
<td>~</td>
</tr>
<tr>
<td>6F</td>
<td>110 1111</td>
<td>o</td>
<td>7F</td>
<td>111 1111</td>
<td>DEL</td>
</tr>
</tbody>
</table>
This appendix summarizes the command line format and the function of each CP/M built-in and transient command. The commands are listed in alphabetical order.

**ASM Command Lines**

**ASM filename<cr>**  Assembles the file filename.ASM; uses the currently logged disk for all files.

**ASM filename.opt<cr>**  Assembles the file filename.ASM on drive o: (A:,B:,..,P:). Writes HEX file on drive p: (A:,B:,..,P:), or skips if p: is Z:.

  Writes PRN file on drive t: (A:,B:,..,P:), sends to console if p: is X:, or skips if p: is Z:.
DDT Command Lines

**DDT<cr>**  Loads DDT and waits for DDT commands.

**DDT x:filename.typ<cr>**  Loads DDT into memory and also loads filename.typ from drive x: into memory for examination, modification, or execution.

DDT Command Summary

**Assss**  Enters assembly language statements beginning at hexadecimal address ssss.

**D**  Displays the contents of the next 192 bytes of memory.

**Dssss,ffff**  Displays the contents of memory starting at hexadecimal address ssss and finishing at hexadecimal address ffff.

**Fssss,ffff,cc**  Fills memory with the 8-bit hexadecimal constant cc starting at hexadecimal address ssss and finishing with hexadecimal address ffff.

**G**  Begins execution at the address contained in the program counter.

**G,bbbb**  Sets a breakpoint at hexadecimal address bbbb, then begins execution at the address contained in the program counter.

**G,bbbb,ccccc**  Sets breakpoints at hexadecimal addresses bbbb and cccc, then begins execution at the address contained in the program counter.

**Gssss**  Begins execution at hexadecimal address ssss.

**Gssss,bbbb**  Sets a breakpoint at hexadecimal address bbbb, then begins execution at hexadecimal address ssss.

**Hx,y**  Hexadecimal sum and difference of x and y.

**Ifilename.typ**  Sets up the default file control block using the name filename.typ.

**L**  Lists the next eleven lines of assembly language program disassembled from memory.

**Lssss**  Lists eleven lines of assembly language program disassembled from memory starting at hexadecimal address ssss.

**Lssss,ffff**  Lists the assembly language program disassembled from memory starting at hexadecimal address ssss and finishing at hexadecimal address ffff.
Appendix B: CP/M Command Summary

**Mssss,ffff,dddd** Moves the contents of the memory block starting at hexadecimal address ssss and ending at hexadecimal address ffff to the block of memory starting at hexadecimal address dddd.

**R** Reads a file from disk into memory (use “I” command first).

**Rnnnn** Reads a file from disk into memory beginning at the hexadecimal address nnnn higher than normal (use “I” command first).

**Sssss** Displays the contents of memory at hexadecimal address ssss and optionally changes the contents.

**Tnnnn** Traces the execution of (hexadecimal) nnnn program instructions.

**Unnnn** Executes (hexadecimal) nnnn program instructions, then stops and displays the CPU register’s contents.

**X** Displays the CPU register’s contents.

**Xr** Displays the contents of CPU or Flag r and optionally changes them.

**DIR Command Lines**

**DIR x:<cr>** Displays directory of all files on drive x:. Drive x: is optional; if omitted, the currently logged drive is used.

**DIR x:filename.typ<cr>** Displays directory of all files on drive x: whose names match the ambiguous or unambiguous filename.typ. Drive x: is optional; if omitted, the currently logged drive is used.

**DUMP Command Line**

**DUMP x:filename.typ <cr>** Displays the hexadecimal representations of each byte stored in the file filename.typ on drive x:. If filename.typ is ambiguous, displays the first file which matches the ambiguous file name.

**ED Command Line**

**ED x:filename.typ <cr>** Invokes the editor, which then searches for filename.typ on drive x: and creates a temporary file x:filename.$$$ to store the edited text. The filename.typ is unambiguous. Drive x: is optional; if omitted, the currently logged drive is assumed.

**ED Command Summary**

**NOTE:** Non-alphabetic commands follow the “Z” command.
nA

Append lines. Moves “n” lines from original file to edit buffer. 0A moves lines until edit buffer is at least half full.

+/-B

Begin/Bottom. Moves CP.
  +B moves CP to beginning of edit buffer
  -B moves CP to end of edit buffer.

+/-nC

Move by characters. Moves CP by “n” character positions.
  + moves forward
  - moves backward.

+/-nD

Delete characters. Deletes “n” characters before or after the CP in the edit buffer.
  + deletes before the CP
  - deletes after the CP.

E

End. Ends edit, closes files, and returns to CP/M; normal end.

nFstring^Z

Find string. Finds the “n”th occurrence of string, beginning the search after the CP.

H

Move to head of edited file. Ends edit, renames files, and then edits former temporary file.

I<cr>

Enter insert mode. Text from keyboard goes into edit buffer after the CP; exit with CONTROL-Z.

Istring^Z

Insert string. Inserts string in edit buffer after the CP.

Istring<cr>

Insert line. Inserts string and CRLF in the edit buffer after the CP.

nJfindstring^Zinsertstring^Zendstring^Z

Juxtaposition. Beginning after the CP, finds findstring, inserts insertstring after it, then deletes all following characters up to but not including endstring; repeats until performed “n” times.

+/-nK

Kill lines. Deletes “n” lines.
  + deletes after the CP
  - deletes before the CP.

+/-nL

Move by lines. Moves the CP to the beginning of the line it is in, then moves the CP “n” lines forward or backward.
  + moves forward
  - moves backward.

nMcommandstring^Z

Macro command. Repeats execution of the ED commands in
commandstring “n” times. “n” = 0, “n” = 1, or “n” absent repeats execution until error occurs.

### nNstring^Z
Find string with autoscan. Finds the “n”th occurrence of string, automatically appending from original file and writing to temporary file as necessary.

### 0
Return to original file. Empties edit buffer, empties temporary file, returns to beginning of original file, ignores previous ED commands.

### +/-nP
Move CP and print pages. Moves the CP forward or backward one page, then displays the page following the CP. “nP” displays “n” pages, pausing after each.

### Q
Quit edit. Erases temporary file and block move file, if any, and returns to CP/M; original file is not changed.

### R<cr>
Read block move file.Copies the entire block move file X$$$$$$$.LIB from disk and inserts it in the edit buffer after the CP.

### Rfilename<cr>
Read library file. Copies the entire file filename with extension LIB from the disk and inserts it in the edit buffer after the CP.

### nSfindstring^Zreplacestring^Z
Substitute string. Starting at the CP, repeats “n” times: finds findstring and replaces it with replacestring.

### +/-nT
Type lines. Displays “n” lines.

+ displays the “n” lines after the CP
- displays the “n” lines before the CP

If the CP is not at the beginning of a line

0T displays from the beginning of the line to the CP
T displays from the CP to the end of the line
0TT displays the entire line without moving the CP.

### +/-U
Uppercase translation. After +U command, alphabetic input to the edit buffer is translated from lowercase to uppercase; after –U, no translation occurs.

### OV
Edit buffer free space/size. Displays the decimal number of free (empty) bytes in the edit buffer and the total size of the edit buffer.

### +/-V
Verify line numbers. After +V, a line number is displayed with each line displayed; ED’s prompt is then preceded by the number of the line containing the CP. After –V, line numbers are not displayed, and ED’s prompt is “*”. 
Write lines. Writes first “n” lines from the edit buffer to the temporary file; deletes these lines from the edit buffer.

Block transfer (Xfer). Copies the “n” lines following the CP from the edit buffer to the temporary block move file X$$$$$$$.LIB; adds to previous contents of that file.

Sleep. Delays execution of the command which follows it. Larger “n” gives longer delay, smaller “n” gives shorter delay.

Move CP to line number “n.” Moves the CP to the beginning of the line number “n” (see “+/-V”).

Continue through line number “m.” A command prefix which gives the ending point for the command which follows it. The beginning point is the location of the CP (see “+/-V”).

Move and display one line. Abbreviated form of +/-nLT.

ERA Command Lines

\textbf{ERA} x:filename.typ<cr> \hspace{1cm} Erases the file filename.typ on the disk in drive x:. The filename and/or typ can be ambiguous. Drive x: is optional; if omitted, the currently logged drive is used.

\textbf{ERA} x:*.*<cr> \hspace{1cm} Erases all files on the disk in drive x:. Drive x: is optional; if omitted, the currently logged drive is used.

Line Editing Commands

\textbf{CONTROL-C} \hspace{1cm} Restarts CP/M if it is the first character in command line. Called \textit{warm start}.

\textbf{CONTROL-E} \hspace{1cm} Moves to the beginning of next line. Used for typing long commands.

\textbf{CONTROL-H} or \textbf{BACKSPACE} \hspace{1cm} Deletes one character and erases it from the screen (CP/M version 2.0 and newer).

\textbf{CONTROL-J} or \textbf{LINE FEED} \hspace{1cm} Same as \textit{carriage return} (CP/M version 2.0 and newer).

\textbf{CONTROL-M} \hspace{1cm} Same as \textit{carriage return} (<cr>).

\textbf{CONTROL-P} \hspace{1cm} Turns on the list device (usually your printer). Type it again to turn off the list device.
Appendix B: CP/M Command Summary

**CONTROL-R** Repeats current command line (useful with version 1.4); it verifies the line is corrected after you delete several characters (CP/M version 1.4 and newer).

**CONTROL-S** Temporarily stops display of data on the console. Press any key to continue.

**CONTROL-U** or **CONTROL-X** Cancels current command line (CP/M version 1.4 and newer).

**RUBOUT** (RUB) or **DELETE** (DEL) Deletes one character and echoes (repeats) it.

**Load Command Line**

**LOAD** `<filename>` Reads the file filename.HEX on drive x: and creates the executable program file filename.COM on drive x:.

**MOVCPM Command Lines**

**MOVCPM** `<cr>` Prepares a new copy of CP/M which uses all of memory; gives control to the new CP/M, but does not save it on disk.

**MOVCPM** `nn` `<cr>` Prepares a new copy of CP/M which uses “nn” K bytes of memory; gives control to the new CP/M, but does not save it on disk.

**MOVCPM** `**` `<cr>` Prepares a new copy of CP/M that uses all of memory, to be saved with SYSGEN or SAVE.

**MOVCPM** `nn` `*` `<cr>` Prepares a new copy of CP/M that uses “nn” K bytes of memory, to be saved with SYSGEN or SAVE.

The “nn” is an integer decimal number. It can be 16 through 64 for CP/M 1.3 or 1.4. For CP/M 2.0 and newer “nn” can be 20 through 64.

**PIP Command Lines**

**PIP** `<cr>` Loads PIP into memory. PIP prompts for commands, executes them, then prompts again.

**PIP pipcommandline** `<cr>` Loads PIP into memory. PIP executes the command pipcommandline, then exits to CP/M.

**PIP Command Summary**

**x:new.typ=y:old.typ[p]** `<cr>` Copies the file old.typ on drive y: to the file new.typ on drive x:, using parameters p.

**x:new.typ=y:old1.typ[p],z:old2.typ[q]** `<cr>` Creates a file new.typ on drive x: that
The CP/M Programmer’s Handbook

consists of the contents of file old1.typ on drive y: using parameters p followed by the contents of file old2.typ on drive z: using parameters q.

\[ x:\text{filename.ty}p=\text{dev:}[p]<\cr> \]

Copies data from device dev: to the file filename.typ on drive x:.

\[ \text{dev:}=x:\text{filename.ty}p[p]<\cr> \]

Copies data from filename.typ on drive x: to device dev:.

\[ \text{dst:}=\text{src:}[p]<\cr> \]

Copies data to device dst: from device src:.

**PIP Parameter Summary**

- **B**: Specifies block mode transfer.
- **Dn**: Deletes all characters after the “n”th column.
- **E**: Echoes the copying to the console as it is being performed.
- **F**: Removes form feed characters during transfer.
- **Gn**: Directs PIP to copy a file from user area “n.”
- **H**: Checks for proper Intel Hex File format.
- **I**: Ignores any .00 records in Intel Hex File transfers.
- **L**: Translates uppercase letters to lowercase.
- **N**: Adds a line number to each line transferred.
- **O**: Object file transfer (ignores end-of-file markers).
- **Pn**: Issues page feed after every “n”th line.
- **Qs\^Z**: Specifies quit of copying after the string “s” is encountered.
- **R**: Directs PIP to copy from a system file.
- **Ss\^Z**: Specifies start of copying after the string “s” is encountered.
- **Tn**: Sets tab stops to every “n”th column.
- **U**: Translates lowercase letters to uppercase.
- **V**: Verifies copy by comparison after copy finished.
- **W**: Directs PIP to copy onto an R/O file.
- **Z**: Zeroes the “parity” bit on ASCII characters.

**PIP Destination Devices**

- **CON**: Logical devices
- **PUN**: Logical devices
- **LST**: Logical devices
- **TTY**: Logical devices
- **PTP**: Logical devices
- **LPT**: Logical devices
- **CRT**: Logical devices
- **UP1**: Logical devices
- **UL1**: Logical devices
- **UC1**: Physical devices
- **UP2**: Physical devices
- **OUT**: Special PIP devices
- **PRN**: Special PIP devices
Appendix B: CP/M Command Summary 477

**PIP Source Devices**

CON: RDR: Logical devices
TTY: PTR:
CRT: URI:
UCI: UR2: Physical devices
NUL: EOF: INP: Special PIP devices

**REN Command Line**

REN newname.typ=oldname.typ<cr> Finds the file oldname.typ and renames it newname.typ.

**SAVE Command Line**

SAVE nnn x:filename.typ<cr> Saves a portion of the Transient Program Area of memory in the file filename.typ on drive x: where nnn is a decimal number representing the number of pages of memory. Drive x: is the option drive specifier.

**STAT Command Lines**

STAT<cr> Displays attributes and amount of free space for all diskette drives accessed since last warm or cold start.

STAT x:<cr> Displays amount of free space on the diskette in drive x:.

STAT x:filename.typ<cr>(CP/M 2.0 and newer) Displays size and attributes of file(s) filename.typ on drive x:. filename.typ may be ambiguous. x: is optional; if omitted, currently logged drive is assumed.

STAT x:filename.typ $atr<cr> Assigns the attribute atr to the file(s) filename.typ on drive x:. File filename.typ may be ambiguous. Drive x: is optional; if omitted, currently logged drive is assumed.

STAT DEV:<cr> Reports which physical devices are currently assigned to the four logical devices.

STAT VAL:<cr> Reports the possible device assignments and partial STAT command line summary.

STAT log:=phy:<cr> Assigns the physical device phy: to the logical device log: (may be more than one assignment on the line; each should be set off by a comma).

STAT USR:<cr>(CP/M 2.0 and newer) Reports the current user number as well as all user numbers for which there are files on currently logged disks.
STAT x:DSK<cr> (CP/M 1.4 and newer) Assigns a temporary write-protect status to drive x:.

SUBMIT Command Lines

SUBMIT filename<cr> Creates a file $$$.SUB which contains the commands listed in filename.SUB; CP/M then executes commands from this file rather than the keyboard.

SUBMIT filename parameters<cr> Creates a file $$$.SUB which contains commands from the file filename.SUB; certain parts of the command lines in filename.SUB are replaced by parameters during creation of $$$.SUB. CP/M then gets commands from this file rather than the keyboard.

SYSGEN Command Line

SYSGEN<cr> Loads the SYSGEN program to transfer CP/M from one diskette to another.

TYPE Command Line

TYPE x:filename.typ<cr> Displays the contents of file filename.typ from drive x: on the console.

USER Command Line

USER n<cr> Sets the User Number to “n,” where “n” is an integer decimal number from 0 to 15, inclusive.

x: Command Line

x:<cr> Changes the currently logged disk drive to drive x:. Drive x: can be “A” through “P.”
# Summary of BDOS Calls

**Table C-1.** BDOS Function Definitions for CP/M-80 Version 2.2

<table>
<thead>
<tr>
<th>Function No.</th>
<th>Function Name</th>
<th>Entry Parameter(s)</th>
<th>Exit Parameter(s)</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>SYSTEM RESET</td>
<td>None</td>
<td>None</td>
<td>Restarts CP/M-80 by returning control to the CCP after reinitializing the disk subsystem.</td>
</tr>
<tr>
<td>01</td>
<td>CONSOLE INPUT</td>
<td>None</td>
<td>A = ASCII character</td>
<td>Returns the next character typed to the character calling program. Any non-printable character is echoed to the screen (like BACKSPACE, TAB, or CARRIAGE RETURN). Execution does not return to the calling program until a character has been typed. Standard CCP control characters are recognized and their actions performed (CONTROL-P begins or ends printer echoing and so on).</td>
</tr>
<tr>
<td>Function</td>
<td>No.</td>
<td>Name</td>
<td>Entry Parameter(s)</td>
<td>Exit Parameter(s)</td>
</tr>
<tr>
<td>----------</td>
<td>-----</td>
<td>---------------------</td>
<td>-----------------------------</td>
<td>-------------------</td>
</tr>
<tr>
<td>02</td>
<td>CONSOLE OUTPUT</td>
<td>E = ASCII character</td>
<td>None</td>
<td>Displays the character in the E register on the console device. Standard CCP control characters are recognized and their actions performed (CONTROL-P begins or ends printer echoing and so on.).</td>
</tr>
<tr>
<td>03</td>
<td>READER INPUT</td>
<td>None</td>
<td>A = ASCII character</td>
<td>Returns the next character received from the reader device to the calling program. Execution does not return to the calling program until a character is received.</td>
</tr>
<tr>
<td>04</td>
<td>PUNCH OUTPUT</td>
<td>E = ASCII character</td>
<td>None</td>
<td>Transmits the character in the E register to the punch device.</td>
</tr>
<tr>
<td>05</td>
<td>LIST OUTPUT</td>
<td>E = ASCII character</td>
<td>None</td>
<td>Transmits the character in the E register to the list device.</td>
</tr>
<tr>
<td>06</td>
<td>DIRECT CONSOLE IN</td>
<td>E = FF hex</td>
<td>A = ASCII</td>
<td>If register E contains an FF hex, the console device is interrogated to see if a character is ready. If no character is ready, a 00 is returned to the calling program in register A; otherwise the character detected is returned in register A. If register E contains any character other than an FF hex, that character is passed to the console display. All CCP control characters are ignored. The user must protect the program against nonsensical characters being sent from or received by the console device.</td>
</tr>
<tr>
<td>07</td>
<td>GET IOBYTE</td>
<td>None</td>
<td>A = IOBYTE</td>
<td>Places a copy of the byte stored at location 0003 hex in the A register before returning control to the calling program.</td>
</tr>
<tr>
<td>08</td>
<td>SET IOBYTE</td>
<td>E = IOBYTE</td>
<td>None</td>
<td>Places a copy of the value in register E into the memory location of 0003 hex before returning control to the calling program.</td>
</tr>
<tr>
<td>09</td>
<td>PRINT STRING</td>
<td>DE = String address</td>
<td>None</td>
<td>Sends the string of characters stored beginning at the address stored in the DE register pair to the console device. All characters in subsequent addresses are sent until BDOS encounters a memory location which contains a 24 hex (an ASCII &quot;$&quot;). The CCP control characters are checked for and performed if encountered.</td>
</tr>
</tbody>
</table>

**Note:** CP/M-80 always copies the contents of the H register in the A register if nothing is to be specifically returned in the A register. Some manufacturers, specifically Microsoft, make use of such information to reduce movement of information between the H and A registers.
## Appendix C: Summary of BDOS Calls

### Table C-1. (Continued)

<table>
<thead>
<tr>
<th>Function No.</th>
<th>Name</th>
<th>Entry Parameter(s)</th>
<th>Exit Parameter(s)</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0A</td>
<td>READ CONSOLE BUFFER</td>
<td>DE = Buffer address</td>
<td>Data in buffer</td>
<td>This function performs essentially the same as the CCP would in that it takes the characters the user types and stores them into the buffer that begins at the address stored in the DE register pair. The first byte in the buffer pointed to by the DE pair must be the maximum length of the command. BDOS will place the number of characters encountered in the second byte, with the typed command beginning with the third byte pointed to by the DE pair. All standard CCP editing characters are recognized during the command entry.</td>
</tr>
<tr>
<td>0B</td>
<td>GET CONSOLE STATUS</td>
<td>None</td>
<td>A = Status</td>
<td>BDOS checks the status of the console device and returns a 00 hex if no character is ready, FF hex if a character has been typed.</td>
</tr>
<tr>
<td>0C</td>
<td>GET VERSION NUMBER</td>
<td>None</td>
<td>HL = Version</td>
<td>If the byte returned in the H register is 00 hex then CP/M is present, if 01, then MP/M is present. The byte returned in the L register is 00 if the version is previous to CP/M 2.0, 20 hex if the version is 2.0, 21 hex if 2.1 and so on.</td>
</tr>
<tr>
<td>0D</td>
<td>RESET DISK SYSTEM</td>
<td>None</td>
<td></td>
<td>Used to tell CP/M to reset the disk subsystem. Should be used any time diskettes are changed.</td>
</tr>
<tr>
<td>0E</td>
<td>SELECT DISK E = Disk number</td>
<td>None</td>
<td></td>
<td>Selects the disk to be used for subsequent disk operations. A 00 hex in the E register indicates disk A, a 01 hex indicates disk B, etc.</td>
</tr>
<tr>
<td>0F</td>
<td>OPEN FILE DE = FCB address</td>
<td>A = 'Found' / not found code</td>
<td></td>
<td>Used to activate a file on the current disk drive and current user area. BDOS scans the first 14 bytes of the designated FCB block and attempts to find a match to the filename in the block. A 3F hex (ASCII &quot;?&quot;&quot;) can be used in any of the filename positions to indicate a &quot;don't care&quot; character. If a match is found, the relevant information about that file is filled into the rest of the FCB by CP/M-80. A value of 00 hex to 03 in register A upon return indicates the open operation was successful, while an FF hex indicates that the file could not be found. If question marks are used to identify a file, the first matching entry is used.</td>
</tr>
</tbody>
</table>

**Note:** CP/M-80 always copies the contents of the H register in the A register if nothing is to be specifically returned in the A register. Some manufacturers, specifically Microsoft, make use of such information to reduce movement of information between the H and A registers.
CONIN (continued)
Console input, in the BIOS, 151
Recognizing incoming function key characters, 221
Use with forced input, 219

CONOUT:
Console output, in the BIOS, 151
Escape sequences to input date and time, 223
Processing output escape sequences, 222

CONST:
Console input status, in the BIOS, 50
Problems with programs that "gobble" characters, 218
Use with forced input, 219

CP/M:
Bringing up a new system, 350

CP/M file system:
Concepts, 17

CP/M records as 128-byte sectors: 71

CRC:
See Cyclic Redundancy Check

CRF:
Example of Random Write, 135

Cancel command line:
CONTROL-U, 49

Captions:
For debug subroutines, 322

CARRIAGE RETURN:
CONTROL-M, 48

Changed diskette:
Size of buffer for detection, in disk parameter block, 36
Work area in disk parameter header, 32

Changing disks:
Need to force disk log-in, 96

Changing user number:
USER, 53

Code table:
Definition of structure in LIBRARY.H, 394
Display all strings, ct_disps, Code, 385, Narrative, 407
Get string for code, ct_strc, Code, 386, Narrative, 407
Get string for index, ct_stri, Code, 386, Narrative, 407
Initialize, ct_init, Code, 384, Narrative, 407
Prompt and return code, ct_parc, Code, 384, Narrative, 407
Return code, ct_code, Code, 385, Narrative, 407
Return index, ct_index, Code, 386, Narrative, 407
Used for command tail parameters, 406

Cold Boot:
BIOS functions, 149
Concepts, 12

Command line:
Canceling, CONTROL-U, 49
Deleting last character typed, 49
Repeating, CONTROL-R, 49

Command Line Editing:
By the CCP, 46

Command tail:
Code tables, C functions, 405
Example program to process parameters, 63
In base page, 60
Input to the CCP, 46
Processing, C functions, 405

Communications:
Using Reader/Punch (Auxiliary), 151

Comp__fname:
Compare file name, Code, 374, Narrative, 401

Compare file name:
Comp__fname, Code, 374, Narrative, 401

Configuration Block:
Accessing from C, 396
Concepts, 211
Suggestion for utility program, 448
Default disk:
Changing, 50
In base page, 59
In CCP prompt, 46
Default File Control Blocks:
In base page, 60
Deferred writes:
In conjunction with track buffering, 231
Delete character:
Rubout/Del, 49
Deleting files:
ERA, 51
Device table:
Accessing from C, 398
Displaying for debugging, 356
Structure, 225
Digital Research:
Manuals, 6
Direct BIOS calls:
Example code, 156
Examples, 65
When to avoid, 15
Directory code:
As returned by BDOS calls, 71
As returned from Create (Make) File, 114
As returned from Rename File, 116
Returned by BDOS Close File, 103
Returned by BDOS Open File, 99
Returned by Search for First Name Match, 103
Returned by Search for Next Name Match, 107
Directory entry: 99
Definition in LIBRARY.H, 394
Directory Parameter Block:
Definition in LIBRARY.H, 393
Disk Drivers:
Debugging, 364
Disk I/O:
Enhancements, 231
In the BIOS, 152
Disk Map:
In file directory entry, 26
Disk Parameter Block:
Accessing from C, 398
Adding extra information, 41
Block shift, mask, and extent mask, 33
Definition in LIBRARY.H, 394
Details, 33
Finding the address of, 125
Maximum allocation block number, 34
Disk Parameter Block (continued)
Number of directory entries — 1, 35
Number of tracks before directory, 36
Pointer in disk parameter header, 31
Reserving allocation blocks for file directory, 35
Sectors per track, 33
Size of buffer for detecting changed diskettes, 36
Worked example for hard disk, 39
Disk Parameter Header:
Details, 28
Disk buffer, 31
Disk parameter block, 31
Pointer to allocation vector, 32
Sector skewing, 28
Work area for changed diskette detection, 32
Disk buffer:
In disk parameter header, 31
Disk definition tables:
Concept, 18
Details, 27
Disk drivers:
Example testbed code, 365
Disk errors:
Strategy, 303
Disk full:
Error returned from Sequential Write, 112
Disk layout:
CP/M on diskettes, 189
Disk map:
As used in C functions, 402
Disk map clear:
Dm__clr, Code, 382, Narrative, 403
Disk map display:
Dm__disp, Code, 382, Narrative, 403
Diskette:
Layout of standard CP/M diskette, 37
Diskette format:
Concepts, 9
Display $-Terminated String:
BDOS Function 9, 88
Display directory error:
Err__dir, Code, 381, Narrative, 400
Displaying an ASCII file:
TYPE, 52
Displaying current user number: 54
Dm__clr:
Disk map clear, Code, 382, Narrative, 403
Dm__disp:
Disk map display, Code, 382, Narrative, 403
Control characters (continued)
CONTROL-C:
Used to abort after BDOS error, 98
CONTROL-P:
Errors generated, 299
CONTROL-Z:
If no Reader driver in BIOS, 75
Used to indicate end of file, 110
Used to terminate prior to BDOS Close File, 103
Conv_dfname:
Convert directory file name, Code, 375, Narrative, 402
Conv_fname:
Convert file name, Code, 375, Narrative, 408
Convert directory file name:
Conv_dfname, Code, 375, Narrative, 402
Convert file name:
Conv_fname, Code, 375, Narrative, 408
Create (Make) file:
BDOS Function 22, 112
Ct_code:
Code table, return code, Code, 385, Narrative, 407
Ct_disps:
Code table, display all strings, Code, 385, Narrative, 407
Ct_index:
Code table, return index, Code, 386, Narrative, 407
Ct_init:
Code table, initialize, Code, 384, Narrative, 407
Ct_parcc:
Code table, prompt and return code, Code, 384, Narrative, 407
Ct_strc:
Code table, get string for code, Code, 386, Narrative, 407
Ct_stri:
Code table, get string for index, Code, 386, Narrative, 407
Current default drive: 97
Current logical disk:
In base page, 59
Current record number:
In FCB, unchanged for Random Read, 132
In FCB, unchanged for Random Write, 132
Current user number:
Displaying, 54
In base page, 59
Customization:
Of CP/M, an overview, 8

Cyclic Redundancy Check:
As used in disk errors, 303

D

DDT:
Dynamic Debug Tool, 185, 329
Manual, 6
I Command used for building new CP/M system, 195
R Command used for building new CP/M system, 195
Used for checking CP/M images, 204
Used for debugging character drivers, 354
Used to create CP/M memory image, 194
Used to debug disk drivers, 364

DESPOOL:
Use of LISTST BIOS entry, 156

DIR:
Display directory of files, 50

DMA buffer:
Default in base page, 60

DPB:
See Disk Parameter Block

DPH:
See Disk Parameter Header

DTR:
PROTOCOL, C program to set protocols, 434
See Data Terminal Ready

Data storage area:
Concept, 17

Data Terminal Ready:
Explanation of DTR protocol, 219

DATE:
C program, sets the date, 442

Date:
Keeping the current date in the BIOS, 224
Reading the date from the console driver, 223

Debug output:
Controlling when it occurs, 324

Debug subroutines: 322
Overall design philosophy, 322

Debugging a new CP/M system, 319

Debugging checklist:
Character output, 361
Disk drivers, 367
Interrupt service routines, 359
Non-interrupt service routine, 359
Real Time Clock, 362

Default DMA Address: 118

Default DMA buffer:
In base page, 60
DO:
Suggestion for utility program, 448

DPB:
See Disk Parameter Block

DPH:
See Disk Parameter Header

E

ED:
Editor, manual, 6

ERA:
Erase (delete) files, 51

Echoing of keyboard characters:
Read Console Byte, 72

End of File:
Detection using Read Sequential, 110

Erase (Delete) File:
BDOS Function 19, 108

ERASE:
C program, a safer way to erase files, 409

Erased files:
Unerasing them, 26

Erasing a file:
ERA, 51
Logical deletion only, 23

Err_dir:
Display directory error, Code, 381, Narrative, 400

Error messages:
Debugging disk drivers, 368, Chapter 12

Errors:
Dealing with hardware errors, 295
Example printer error routine, 301
Handling disk errors, 303
Hardware, analysis, 297
Hardware, correction, 299
Hardware, detection strategy, 296
Hardware, indication, 297
Improved disk error messages, 312
Practical handling, character I/O, 299

Escape sequences:
Function keys, debugging character driver, 360
Incoming, debugging character driver, 360
Processing output sequences, 222
Recognizing function key sequences, 222
Suggestion for utility program, 448
Support via device table, 226

Etx/Ack (continued)
Protocol, C program to set protocols, 434

Example programs:
Ordering diskette, 4

Extent:
In file directory entry, 26
Of files, concepts, 18

Extent mask:
In disk parameter block, 33

F

FCB:
Default FCB's in base page, 60
See File Control Block

FDOS:
Rarely used term for BDOS/CCP combined

File Attributes: 99
Setting, 121
See File status

File Control Block:
Creating one from an ASCII file name, 100
Concepts, 18
Definition in LIBRARY.H, 393
Structure, 41
Used for random file operations, 43
Used for sequential file operations, 43
Used in BDOS Open File, 99
Used in BDOS Requests, 71

File Directory:
Accessing entries directly, 399
Processing, C functions, 402

File Organizations:
Concepts, 41

File Protection:
Special characters in file name, 114

File changed:
File status bit in file directory entry, 26

File directory:
Accessing, C functions, 400
Accessing, via BDOS & C functions, 408
Concept, 17
Details, 18
Disk map, 26
Displaying contents, DIR, 50
Entry structure, 22
Erasing files, ERA, 51
File extent, 26
File name and type in entry, 27
Matching names, C functions, 401
Number of entries — 1, in disk parameter block, 35
File directory (continued)
Number of tracks before, 36
Record number, 27
Status (attribute) bits, 26
User number in entry, 22
File extent:
Concepts, 18
In file directory entry, 26
Manipulation to achieve Random I/O, 110-12
Opening extent 0 for Random I/O, 133-34
File name/type:
In file directory entry, 23
File protection:
Suggestion for utility program, 426
File status:
In file directory entry, 26
File system:
Concepts, 17
File type:
Conventions for actual types, 24
Filecopy:
Suggestion for utility program, 426
Files:
Creating, sequence of operations, 20
Displaying a directory, DIR, 50
Find:
C program, finds lost files, 416
Flushing buffers:
Prior to BDOS Close File, 103
Forced input:
Concepts, 219
Debugging character driver, 360
Suggestion for utility program, 448
Framing error:
Character I/O, handling, 300
Function Key table:
Accessing from C, 397
Function keys:
Structure in LIBRARY.H, 392
Support with enhanced BIOS, 220
Testing in a live BIOS, 370
FUNKEY:
C program, sets the function keys, 445
G
GETC:
Example of Read Sequential, 111
GETDPB:
Example of Get Disk Parameter Block Address, 126
GFA:
Example of Get File Attributes, 122
GNF:
Example of Search First/Next File Name Match, 104
Get CP/M Version Number:
BDOS Function 12, 94
Get Current Default Disk:
BDOS Function 25, 118
Get Disk Parameter Block Address:
BDOS Function 31, 125
Get Disk Parameter Block Address:
Get_dpb, Code, 383
Get File Size:
BDOS Function 35, 142
Get I0BYTE Setting:
BDOS Function 7, 80
Get Read-Only Disks:
BDOS Function 29, 120
Get allocation vector:
BDOS Function 27, 119
Get configuration block address:
Get_cba, 372
Get next directory entry:
Get_nde, Code, 378, Narrative, 400
Get next file name:
Get_nfn, Code, 376, Narrative, 408
Get_cba:
Get configuration block address, 372
Get_dpb:
Get Disk Parameter Block Address, Code, 383
Get_nde:
Get next directory entry, Code, 378, Narrative, 400
Get_nfn:
Get next file name, Code, 376, Narrative, 408
H
HEX file structure: 195
HOME:
Home disk heads, in the BIOS, 153
HSTBUF:
In the BIOS, 152
Hard disk:
Division into several logical disks, 39
Special considerations, 36
Hardware errors:
Dealing with, 295, Chapter 9
Hardware reset:
Debugging character driver, 359
**Heath/Zenith:**
Special version of CP/M, 55

**Host Buffer:**
In the BIOS, 152

**Host sector size:**
In the BIOS, 152

**I/O Redirection:**
Assign, C program to assign physical devices, 439
Concepts, 214
I/OBYTE Structure, 57
IF/ENDIF directives:
Used for debug subroutines, 323
I/OBYTE:
Equates for bit fields, 86
Structure, 57
Use for polling communications line, 75
Use with Direct Console I/O for communications, 80

**Initialization of debug subroutines:** 323

**Input redirection:**
Debugging character driver, 359

**Input/Output:**
Fake I/O for debugging purposes, 327

**Interactions:**
Between CCP, BDOS, and BIOS, 15

**Interlace:**
See Sector skewing

**Interrupt service routines:**
Debugging checklist, 357

**Interrupts:**
Architecture, 216
Circular buffers, 217
Dealing with buffer overflow, 219
Debugging service routines, 329
Use for character input drivers, 215

**Johnson-Laird Inc.:**
Ordering diskette, 4

**Jump vector:**
Use for entering the BIOS, 15

**L**

**LIBRARY.C:**
Utility function library, 372

**LIBRARY.H:**
Header for LIBRARY.C functions, 390

**LIST:**
List output, in the BIOS, 151

**LISTST:**
List device output status, in the BIOS, 156

**LST:**
Logical list device, 56

**Line editing:**
Using Read Console String, 91

**Line feed:**
CONTROL-J, 48

**List Device Errors:**
Problems with BDOS Function 5, 78

**Loading CP/M:**
Overview, 11

**Loading programs:**
Via the CCP, 54

**Loadsys:**
Suggestion for utility program, 448

**Location 0000H:**
Use for warm boot, 13

**Location 0005H:**
Simple examples of use, 20
Use for BDOS function calls, 14

**Logging in a disk:**
Using BDOS Reset Disk System, 96

**Logical deletion of files:** 23

**Logical devices:**
CON:, LST:, AUX:, RDR:, PUN:, 56

**Logical disk:**
As represented in File Control Block, 42
Division of hard disk into several logical disks, 39
Selecting, 97

**Logical Input/Output:**
As afforded by the BIOS, 15

**Logical records:**
Concepts, 41

**Logical sectors to physical:** 28
SECTRAN, in the BIOS, 156

**Login Vector:**
See BDOS Function 24, 116

**Lowercase letters in file name:** 114

**M-disk:**
Using memory as an ultra-fast disk, 232

**M80:**
Macro Assembler, 185

**MAC:**
Macro Assembler, 185

**MAKE:**
C program, makes files visible/invisible, 427
MOVE:
C program, moves files between user numbers, 423

MOVCPM:
In conjunction with patches to CP/M, 234
Relocating the CCP and BDOS, 201
Use in building a new CP/M system, 182

MSGOUT:
Example of Write Console Byte, 74

MSGOUTI:
Example of Write Console Byte, 74

Manuals:
From Digital Research, 6

Maximum allocation block number:
In disk parameter block, 34

Memory:
Displaying in debug subroutines, 324
Finding size of area available for programs, 65
Use of hidden memory for buffers, 216
Used as an ultra-fast disk, 232

Memory dumps:
Base page, 61

Memory image:
Checking a new system, 204
Of new CP/M system, 185

Memory layout:
For example BIOS, 190
For input to SYSGEN, 187
With CP/M loaded, 13

Messages:
As an aid to debugging, 326

Note:
For example console dialog, 3

Number of file directory entries:
In disk parameter block, 35

OM:
Example of Display $-Terminated String, 89

OPENF:
Example of Open File, 100

Open File:
BDOS Function 15, 98

Open directory:
Open___dir, Code, 378, Narrative, 400

Output Escape sequence:
Debugging character output driver, 362

Overrun error:
Character I/O, handling, 300

Overwriting the CCP:
To gain memory, 45

Owner:
Suggestion for utility program, 426

P

PIP:
Used to test a new BIOS, 369

PROM Bootstrap:
Used to load CP/M, 11

PUN:
Logical Punch, 56

PUNCH:
Punch (Auxiliary) output, in the BIOS, 151

PUTC:
Example of Write Sequential, 113

PUTCPM:
Example program, 191
Writing a utility, 189

Parallel printers:
Error handling, 301

Parameters:
Example program to process command tail, 63

Parity error:
Character I/O, handling, 300

Pass counters:
Use in debug subroutines, 324

Patching CP/M:
General techniques, 234

Performance:
Effect of sector skewing, 29

Physical end of line:
CONTROL-E, 47

Physical sectors:
Relative, on a hard disk, 38

Polled Reader Input:
Problems and solutions, 75

Polled communications:
Using Direct Console I/O, 80

Printer echo:
CONTROL-P, 48

Printer errors:
Example routine, 301
Use of watchdog timer, 224
**Index**

<table>
<thead>
<tr>
<th>RST?</th>
<th>Use for debugging drivers, 356</th>
</tr>
</thead>
<tbody>
<tr>
<td>RTS</td>
<td>See also Buffer thresholds, Request to Send Protocol, C program to set protocols, 434</td>
</tr>
<tr>
<td>Random Read</td>
<td>Using Read Sequential, 110</td>
</tr>
<tr>
<td>Random Write</td>
<td>Using Write Sequential, 112</td>
</tr>
<tr>
<td>Random files</td>
<td>Concepts, 43</td>
</tr>
<tr>
<td></td>
<td>Creating an empty file, 144</td>
</tr>
<tr>
<td></td>
<td>Problem of sparse files, 44</td>
</tr>
<tr>
<td></td>
<td>Virtual size, 142</td>
</tr>
<tr>
<td>Random record number</td>
<td>In FCB, set for Random Read, 132</td>
</tr>
<tr>
<td></td>
<td>In FCB, set for Random Write, 132</td>
</tr>
<tr>
<td>Rd_disk</td>
<td>Read disk (via BIOS), Code, 377, Narrative, 400</td>
</tr>
<tr>
<td>Read Console Byte</td>
<td>BDOS Function 1, 72</td>
</tr>
<tr>
<td>Read Console Status</td>
<td>BDOS Function 1l, 94</td>
</tr>
<tr>
<td>Read Console String</td>
<td>BDOS Function 10, 90</td>
</tr>
<tr>
<td>Read Random</td>
<td>BDOS Function 33, 131</td>
</tr>
<tr>
<td>Read Reader Byte</td>
<td>BDOS Function 3, 75</td>
</tr>
<tr>
<td>Read Sequential</td>
<td>BDOS Function 20, 109</td>
</tr>
<tr>
<td>Read disk (via BIOS)</td>
<td>Rd_disk, Code, 377, Narrative, 400</td>
</tr>
<tr>
<td>Read-Only</td>
<td>Automatic setting after changing diskettes, 32</td>
</tr>
<tr>
<td></td>
<td>File status bit in file directory entry, 26</td>
</tr>
<tr>
<td>Read-Only Disks</td>
<td>120</td>
</tr>
<tr>
<td>Read-Only File</td>
<td>Attribute bit, 121</td>
</tr>
<tr>
<td>Read/write directory</td>
<td>Rw_dir, Code, 380, Narrative, 400</td>
</tr>
<tr>
<td>Reading/Writing disk</td>
<td>Direct BIOS calls from C, 399</td>
</tr>
<tr>
<td>Real Time Clock</td>
<td>Debugging, 362</td>
</tr>
<tr>
<td></td>
<td>Example testbed code, 363</td>
</tr>
<tr>
<td></td>
<td>TIME, C program to set the time, 444</td>
</tr>
</tbody>
</table>

**Printer timeout error:**
Handling, 300

**Program loading:**
Via the CCP, 54

**Program termination:**
Returning to CP/M, 66

**Prompt:**
From the CCP, 46

**Protect/Unprotect:**
Suggestion for utility program, 426

**PROTOCOL:**
C program, sets serial line protocols, 434

**Protocol:**
See also Data Terminal Ready, Request to Send, Xon/Xoff, Etx/Ack
Definitions in LIBRARY.H, 392
Support in enhanced BIOS, 218
Support via device table, 226
Xon/Xoff, used by TYPE, 52

**Public files:**
Patches to create this feature, 235
Suggestion for utility program, 448

**Public/Private:**
Suggestion for utility program, 448

**R**

**RAM-disk:**
Using memory as an ultra-fast disk, 232

**RCS:**
Example of Direct Console I/O, 81

**RDR:**
Logical Reader, 56

**READ:**
Read Sector, in the BIOS, 154

**READER:**
Reader input, in the BIOS, 152

**REN:**
Rename file, 52

**RF:**
Example of Rename File, 117

**RLS/RDR:**
Example of Read Reader Byte, 76

**RMAC:**
Relocatable Macro Assembler, 185

**RO:**
Example of Random File I/O, 136

**RSA:**
Example of Read Console String, 92
Reclaim:
Suggestion for utility program, 426

Record number:
In file directory entry, 26
Manipulation to achieve Random I/O, 110, 112

Registers:
Displaying in debug subroutines, 324

Relative page offset:
Use for making direct BIOS calls, 65

Relative physical sectors:
On a hard disk, 38

Release diskettes:
Files from Digital Research, 6

Rename File:
BDOS Function 23, 115

Renaming a file:
REN, 52

Repeat command line:
CONTROL-R, 48

Request to Send:
Explanation of RTS protocol, 219

Reserved area:
Concept, 17

Reset:
Signal used to start loading of CP/M, 11

Reset Disk System:
BDOS Function 13, 95

Reset Logical Disk Drive:
BDOS Function 37, 143

Resident CCP commands: 14

Restoring registers:
In interrupt service routine, 356

Rw_dir:
Read/write directory, Code, 380, Narrative, 400

SAVE:
Save memory image in disk file, 53
Use in building new CP/M system, 194

SECTRAN:
Logical sector to physical, in the BIOS, 156

SELDSK:
Debugging disk drivers, 367
Select disk, in the BIOS, 153

SETDMA:
Set DMA Address, in the BIOS, 154

SETSEC:
Set Sector, in the BIOS, 153

SETTRK:
Set Track, in the BIOS, 153

SETTRK/SEC:
Debugging disk drivers, 367

SFA:
Example of Set File Attributes, 122

SID:
Debugging tool, 330

STAT:
Use for displaying current user number, 54

SYSGEN:
System Generator, 185
Writing a new system to disk, 186

Savesys:
Suggestion for utility program, 448

Saving memory on disk:
SAVE, 53

Search First/Next:
Example use together, 107

Search for file:
Srch_file, Code, 376, Narrative, 408

Search for Next File Name Match:
BDOS Function 18, 107
Require for Search for First, 104

Sector interface:
See Sector skewing

Sector size:
Host, in the BIOS, 152

Sector skewing:
Effect on performance, 29
For CP/M image on disk, 190
In disk parameter header, 28

Sector skipping:
Concepts, 304

Sector sparing:
Concepts, 304

Sectors:
Use in allocation blocks, 18

Sectors per track:
In disk parameter block, 33

Select Logical Disk:
BDOS Function 14, 97

Sequential Files:
Concepts, 43

Set DMA (Read/Write) Address:
BDOS Function 26, 118
Required by Search for First Name Match, 104
Set File Attributes:
BDOS Function 30, 121

Set IOBYTE:
BDOS Function 8, 86

Set Logical Disk Read-Only:
BDOS Function 28, 120

Set Random Record Number:
BDOS Function 36, 142

Set disk parameters for rd/wrt_disk:
Setdisk, Code, 378, Narrative, 400

Set search control block:
Setscb, Code, 381, Narrative, 401

Set/Get User Number:
BDOS Function 32, 131

Set_disk:
Set disk parameters for rd/wrt_disk, Code, 378, Narrative, 401

Setscb:
Set search control block, Code, 381, Narrative, 401

Setterm:
Suggestion for utility program, 448

Shadow PROM:
Used to load CP/M, 11

Short:
Minor change to C Language, 395

Single-density, single-sided:
Diskette format, 10

Single disk reset, 143

Skewing:
See Sector skewing

Skipping:
Skipping bad sectors on disk, 304

SPACE:
C program, shows used/free disk space, 420

Spare:
Suggestion for utility program, 448

Spare directory:
Debugging disk drivers, 367

Sparing:
Use of spare sectors on disk, 304

Sparse Random Files:
Problem, 44

Special version of CP/M:
Heath/Zenith, 55

SPEED:
C program, sets baud rates, 431

Srch_file:
Search for file, Code, 376, Narrative, 408

Srtrcmp:
Substring compare, 373

Stack:
Filling with known pattern, 323

Stack overflow:
In interrupt service routine, 358

Standard BIOS:
Example code, 158

String scan:
Strscn, 372

String scan, uppercase:
Ustrscn, 372

Strscn:
String scan, 372

Structure:
Of CP/M, 5

Subroutine:
CCPM, Check if CP/M Version 2, 95

CDISK, Change Disk, 96

CRF, Create Random File, 135

DB$Blank, Display a blank, 344

DB$CAH, Convert A to ASCII Hex., 343

DB$CRLF, Display Carriage Return, Line Feed, 344

DB$Colon, Display a colon, 344

DB$Conin, Debug console input, 336

DB$Conout, Debug console output, 336

DB$DAH, Display A in Hex., 343

DB$DHLH, Display HL in Hex., 343

DB$Display$CALLA, Display call address, 343

DB$Display, Main debug display, 338

DB$Display, Main debug display, 338

DB$GHV, Get Hex. Value, 348

DB$Init, Debug initialize, 335

DB$Input, Debug Port Input, 346

DB$MEMORY, Debug display of memory/registers, 325

DB$MSG, Display Message, 345

DB$MSGI, Display Message (In-line), 345

DB$Off, Turn debug output off, 337

DB$On, Turn debug output on, 337

DB$Output, Debug Port Output, 347

DB$Pass, Decrement the pass counter, 337

DB$Set$Pass, Set pass counter, 337

DIVHL, Divide HL by DE, 129

FOLD, Fold lowercase to upper, 93

FSCMP, Folded String Compare, 93

GAB, Get Allocation Block given Track/Sector, 128

GDTAS, Get Directory Track/Sector, 127

GETC, Get Character from Sequential File, 111

GETDPB, Get Disk Parameter Block Address, 126

GFA, Get File Attributes, 122

GMTAS, Get Maximum Track/Sector, 127
Subroutine (continued)

GNF, Get Next File matching ambiguous name, 104
GNTAS, Get Next Track/Sector, 128
GTAS, Get Track/Sector from Allocation block No., 126
MSGOUT, Message Output, 74
MSGOUTI, Message Output In-Line, 74
MULHL, Multiply HL by DE, 129
OM, Output Message selected by A register, 89
OPENF, Open File given ASCII file name, 100
PUTC, Put Character to Sequential File, 113
RCS, Read Console String, 81
RF, Rename File, 117
RL$RDR, Read Line from Reader, 76
RO, Random File I/O (non-128-byte records), 136
RSA, Return Subprocessor Address, 93
SDLR, Shift DE,HL one bit right, 141
SFA, Set File Attributes, 122
SHLR, Shift HL right one bit, 130
SUBHL, Subtract DE from HL, 130
TERM, Terminal Emulator, 87
TOUPPER, Fold lowercase to upper, 84
WL$LST, Write Line to List Device, 79
WL$PUN, Write Line to Punch, 78

Substring compare:
Sstrcmp, 373
Ustrcmp, 372

System file:
Attribute bit, 121
File status bit in file directory entry, 26
Not displayed by DIR, 51

System Reset:
BDOS Function 0, 71

T

TERM:
Example of Set/Get IOBYTE, 87
TIME:
C program, sets the time, 442
TYPE:
Type an ASCII file, 52
Tab:
Interaction of tab characters and escape sequences, 222
Tab expansion:
Supported by Write Console Byte, 73
Using Display $-Terminated String, 89
Termination of programs, returning to CCP: 45
Testbed:
Use for new drivers, 353

Time:
Correct display during debugging, 364
Keeping the current time in the BIOS, 224
Reading the time from the console driver, 223

Top of RAM:
Finding, via base page, 60
Track buffering:
Enhancement to disk I/O, 231
Track offset:
See Tracks before directory
Tracks before directory:
In disk parameter block, 36

Transient Program Area:
Finding available size, 65

Typeahead:
Concepts, 217
Dealing with buffer overflow, 219

U

Undo command line:
CONTROL-U, 49
UNERASE:
C program, restores erased files, 412

User Number:
Changing under program control, 131
Changing using USER, 53
Displaying, 54
In base page, 59
In file directory entry, 22
Patches to make this appear in CCP prompt, 235
Suggestion for utility program, 426

Ustrcmp:
Uppercase substring compare, 373

Utility programs: 371

V

Variable record lengths:
Processing in Random Files, 133, 134

W

WLSLST:
Example of Write List Byte, 79

WRITE:
Example of Write Punch Byte, 78

WRITE:
Write sector, in the BIOS, 155

Warm Boot:
After BDOS Error, 98
Warm Boot (continued)
BIOS functions, 150
Initiated by CONTROL-C, 47
Initiated by pressing a key, 94
Initiated by System Reset BDOS Function, 72
JMP at location 0000H, 55
Reloading the CCP, 45
Resetting Read-Only disks, 120
Setting default DMA Address, 118
Technique for avoiding, 66
Use of location 0000H, 13
Watchdog timer:
Concepts, 225
Debugging Real Time Clock, 364
Use for detecting printer errors, 224
Write Console Byte:
BDOS Function 2, 73
Write List Byte:
BDOS Function 5, 77
Write Punch Byte:
BDOS Function 4, 77
Write Random:
BDOS Function 34, 133
Write Random with Zero-fill:
BDOS Function 40, 144
Write Sequential:
BDOS Function 21, 110
Write disk (via BIOS):
Wrt_disk, Code, 377, Narrative, 400
X
Xoff:
CONTROL-S, 48
Xon:
CONTROL-Q, 49
Xon/Xoff:
Debugging character driver, 358, 362
Explanation of protocol, 240
PROTOCOL, C program to set protocols, 434
Supported by Read Console Byte, 72
Use by TYPE, 53
Z
ZSID:
Z80 Symbolic Interactive Debugger, 185, 350