# THE PROGRAMMER'S CP/M® HANDBOOK

# THE PROGRAMMER'S CP/M® HANDBOOK

Andy Johnson-Laird

Osborne/McGraw-Hill
Berkeley, California

Published by
Osborne/McGraw-Hill
2600 Tenth Street
Berkeley, California 94710
U.S.A.

For information on translations and book distributors outside of the U.S.A., please write to Osborne/McGraw-Hill at the above address.

**THE PROGRAMMER'S CP/M® HANDBOOK**

1234567890 DODO 89876543

ISBN 0-88134-103-7 (Paperback Edition)
ISBN 0-88134-119-3 (Hardcover Edition)

Mary Borchers, Acquisitions Editor
Ralph Baumgartner, Technical Editor
Susan Schwartz, Copy Editor
Judy Wohlfrom, Text Design
Yashi Okita, Cover Design

## Dedication

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 (formerly 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

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 1—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.

- ^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.

An excellent book for those who do not know how to program in C is *The C Programming Language* by Brian Kernighan and Dennis Ritchie (Prentice-Hall). Appendix A of this book is the C Reference Manual.

## 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.

# 2 The Structure of CP/M

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.

## CP/M from Digital Research

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.

*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

fall into two groups: One group is used only to rebuild CP/M, while the other set is general-purpose programming tools.

## 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

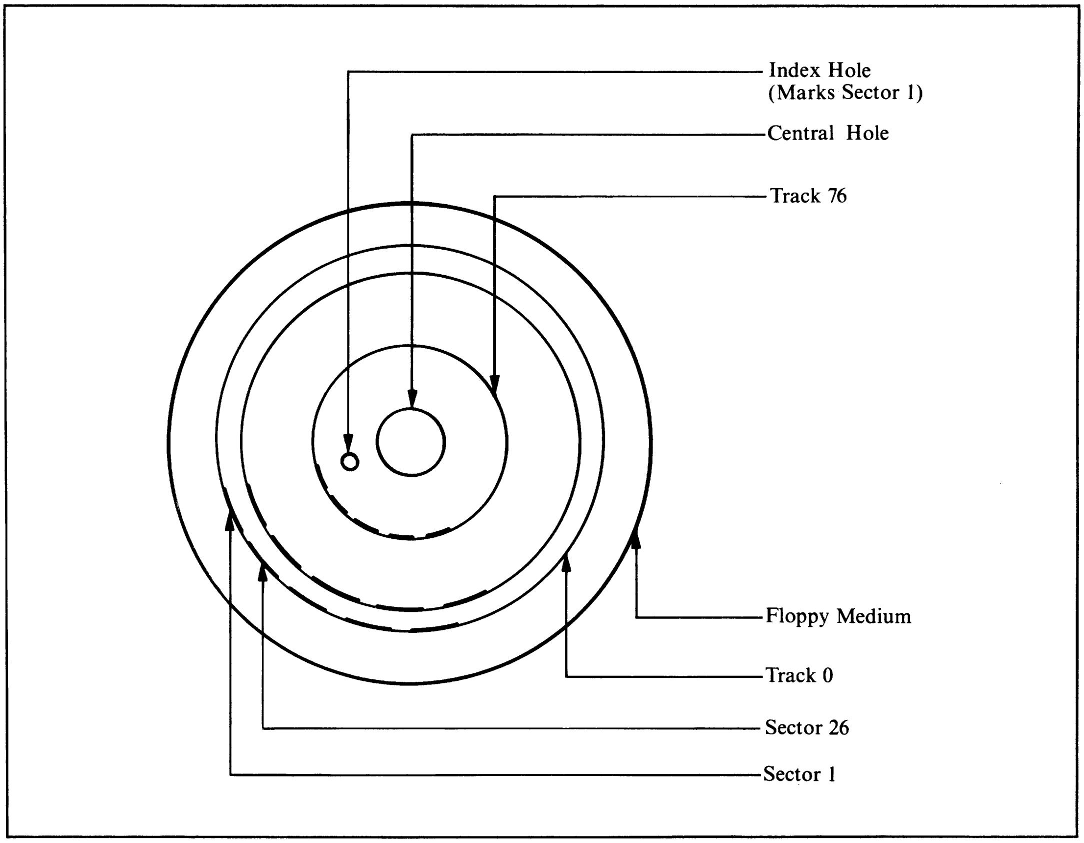


**Figure 2-1.** Floppy disk layout

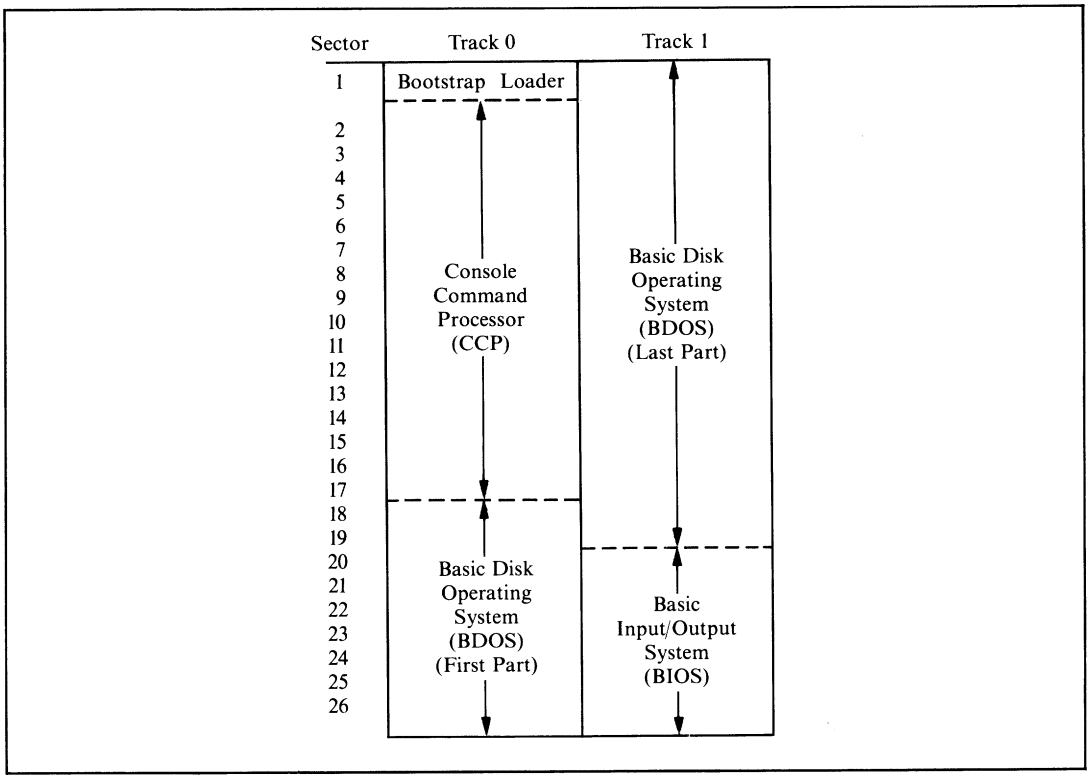

Sector
Track 0
Track 1
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
Bootstrap Loader
Console Command Processor (CCP)
Basic Disk Operating System (BDOS) (First Part)
Basic Disk Operating System (BDOS) (Last Part)
Basic Input/Output System (BIOS)

**Figure 2-2.** Layout of CP/M on tracks 0 and 1 of floppy disk

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.

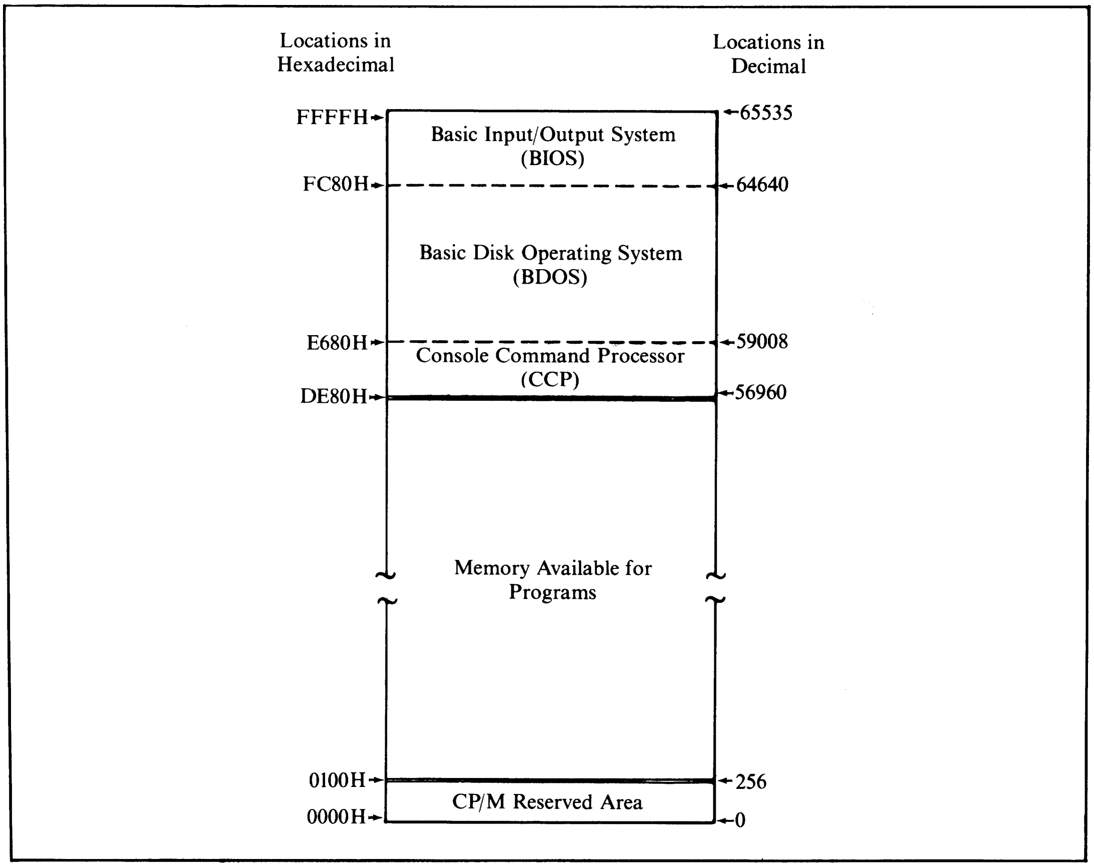

Locations in Hexadecimal
Locations in Decimal
FFFFH
65535
Basic Input/Output System (BIOS)
FC80H
64640
Basic Disk Operating System (BDOS)
E680H
59008
Console Command Processor (CCP)
DE80H
56960
Memory Available for Programs
0100H
256
CP/M Reserved Area
0000H
0

**Figure 2-3.** Memory layout with CP/M loaded

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.

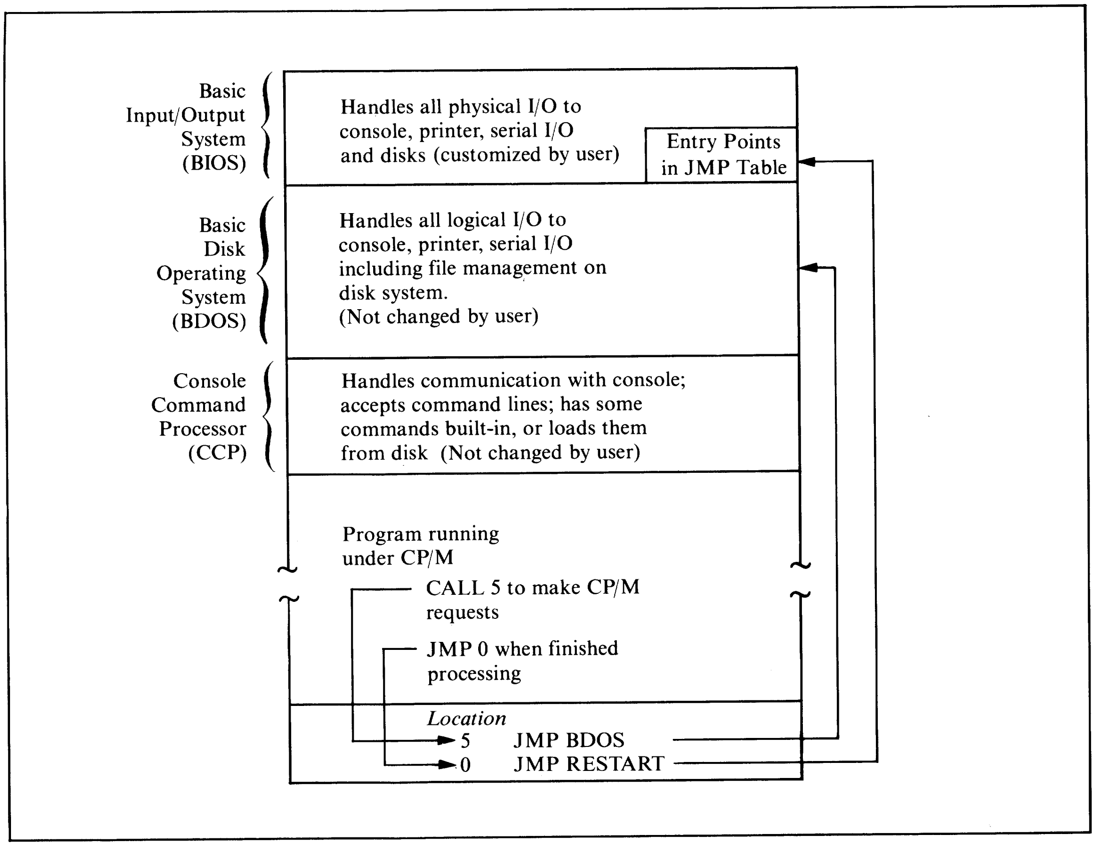

Basic Input/Output System (BIOS)
Handles all physical I/O to console, printer, serial I/O and disks (customized by user)
Entry Points in JMP Table
Basic Disk Operating System (BDOS)
Handles all logical I/O to console, printer, serial I/O including file management on disk system. (Not changed by user)
Console Command Processor (CCP)
Handles communication with console; accepts command lines; has some commands built-in, or loads them from disk (Not changed by user)
Program running under CP/M
CALL 5 to make CP/M requests
JMP 0 when finished processing
Location
5 JMP BDOS
0 JMP RESTART

**Figure 2-4.** CP/M's functional breakdown

3

# The CP/M File System

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 1K 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 0FFFFH (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 (1K) 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 1K-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.

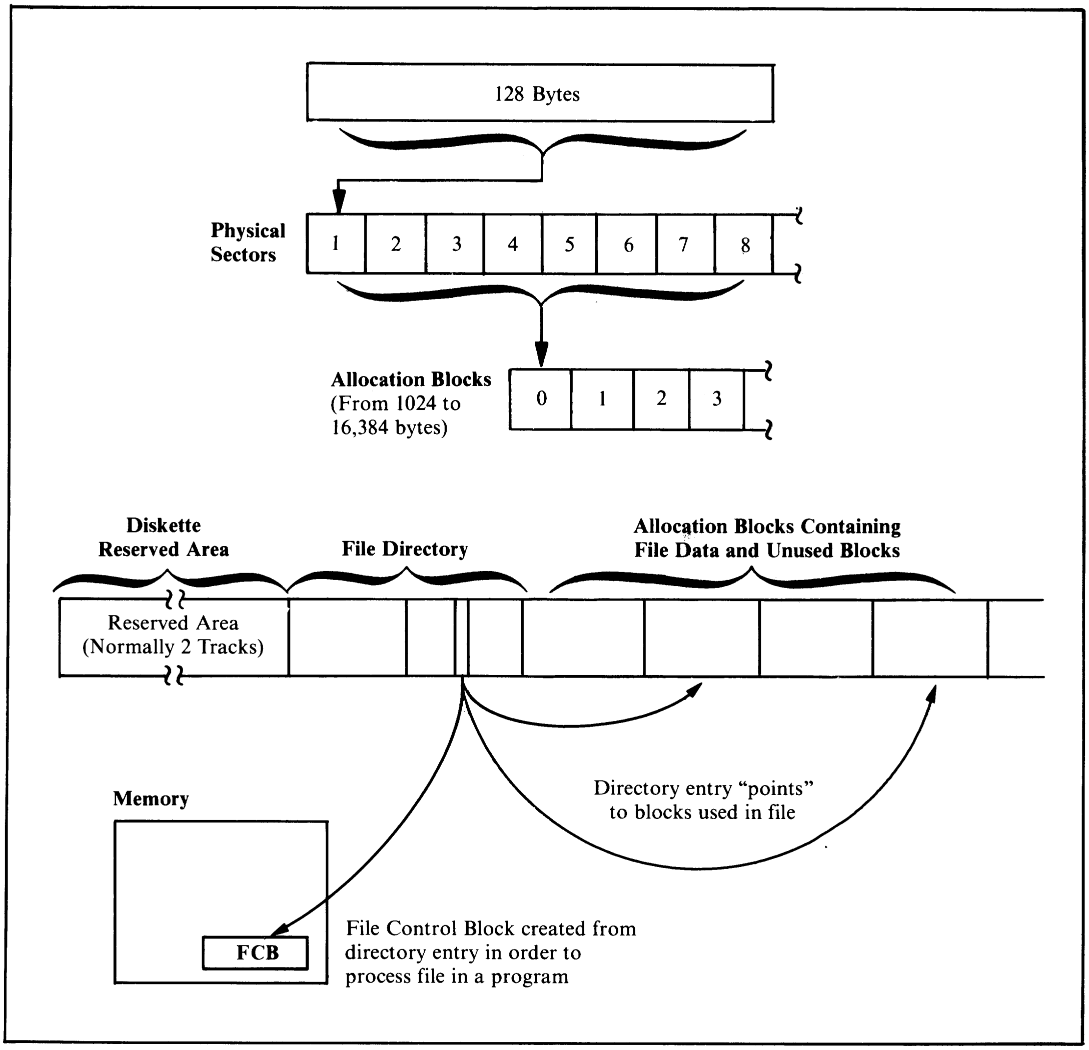

128 Bytes
Physical Sectors
1
2
3
4
5
6
7
8
Allocation Blocks (From 1024 to 16,384 bytes)
0
1
2
3
Diskette Reserved Area
File Directory
Allocation Blocks Containing File Data and Unused Blocks
Reserved Area (Normally 2 Tracks)
Memory
FCB
Directory entry "points" to blocks used in file
File Control Block created from directory entry in order to process file in a program

**Figure 3-1.** The hierarchical relationship between sectors, allocation blocks, directory entires, and FCBs

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.

- 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 1 - 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

```
0000 =          FDE$USER        EQU     0       ;File user number (LS 4 bits)
0001 =          FDE$NAME        EQU     1       ;File name (8 bytes)
0009 =          FDE$TYP         EQU     9       ;File type
                                                ;Offsets for bits used in type
0009 =          FDE$RO          EQU     9       ;Bit 7 = 1 - Read only
000A =          FDE$SYS         EQU     10      ;Bit 7 = 1 - System status
000B =          FDE$CHANGE      EQU     11      ;Bit 7 = 0 = File Written To
                                                ;
000C =          FDE$EXTENT      EQU     12      ;Extent number
                                                ;13, 14 reserved for CP/M
000F =          FDE$RECUSED     EQU     15      ;Records used in this extent
0010 =          FDE$ABUSED      EQU     16      ;Allocation blocks used
                ;
                ;
                ;
0000            FD$USER:        DS              ;File user number
0001            FD$NAME:        DS      8       ;File name
0009            FD$TYP:         DS      3       ;File type
000C            FD$EXTENT:      DS      1       ;Extent
000D            FD$RESV:        DS      2       ;Reserved for CP/M
000F            FD$RECUSED:     DS      1       ;Records used in this extent
0010            FD$ABUSED:      DS      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

*.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 thie 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
                                                        ; (used to access the headers)
0000 1000               DW      SKEWTABLE               ;Pointer to logical-to-physical
                                                        ; sector conversion table
0002 0000               DW      0                       ;Scratch pad areas used by CP/M
0004 0000               DW      0
0006 0000               DW      0
0008 2A00               DW      DIRBUF                  ;Pointer to Directory Buffer
                                                        ; work area
000A AA00               DW      DPBO                    ;Pointer to disk parameter block
000C B900               DW      WACD                    ;Pointer to work area (used to
                                                        ; check for changed diskettes)
000E C900               DW      ALVECO                  ;Pointer to allocation vector
                ;
                ;
                ;       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 directery entries 1
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.
                                                        ; Indexed by logical sector
0010 01070D13           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 1218040A           DB      18,24,04,10             ;20,21,22,23
0028 1016               DB      16,22                   ;24,25
                ;
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
                                                        ;Only used for removable media
00C9            ALVECO: DS      (NOAB/8)+1              ;Allocation vector #0
                                                        ;Needs 1 bit per allocation
                                                        ; block
```

**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

| **Physical Sector** | **Logical Sector** | | | | | |
|---|---|---|---|---|---|---|
| | **Pass** 1 | **Pass** 2 | **Pass** 3 | **Pass** 4 | **Pass** 5 | **Pass** 6 |
| 1 | 0 | | | | | |
| 2 | | | | 13 | | |
| 3 | | | 9 | | | |
| 4 | | | | | | 22 |
| 5 | | 5 | | | | |
| 6 | | | | | 18 | |
| 7 | 1 | | | | | |
| 8 | | | | 14 | | |
| 9 | | | 10 | | | |
| 10 | | | | | | 23 |
| 11 | | 6 | | | | |
| 12 | | | | | 19 | |
| 13 | 2 | | | | | |
| 14 | | | | 15 | | |
| 15 | | | 11 | | | |
| 16 | | | | | | 24 |
| 17 | | 7 | | | | |
| 18 | | | | | 20 | |
| 19 | 3 | | | | | |
| 20 | | | | 16 | | |
| 21 | | | 12 | | | |
| 22 | | | | | | 25 |
| 23 | | 8 | | | | |
| 24 | | | | | 21 | |
| 25 | 4 | | | | | |
| 26 | | | | 17 | | |
| 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 (DPB0)** 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

```
                       SKEWTABLE:                                ;Logical sector
0000 01070D13                  DB        01,07,13,19             ;0,1,2,3
0004 19050B11                  DB        25,05,11,17             ;4,5,6,7
0008 1703090F                  DB        23,03,09,15             ;8,9,10,11
000C 1502080E                  DB        21,02,08,14             ;12,13,14,15
0010 141A060C                  DB        20,26,06,12             ;16,17,18,19
0014 1218040A                  DB        18,24,04,10             ;20,21,22,23
0018 1016                      DB        16,22                   ;24,25
                       ;
                       ;
                       ;       The code to translate logical sectors to physical
                       ;        sectors is as follows:
                       ;
                       ;       On entry, the logical sector will be transferred from
                       ;       CP/M as a 16-bit value in registers BC.
                       ;       CP/M also transfers the address of the skew table
                       ;       in registers DE (it finds the skew table by looking in
                       ;       the disk parameter header entry).
                       ;
                       ;       On return, the physical sector will be placed
                       ;       in registers HL.
                       ;
                       SECTRAN:
001A EB                        XCHG                       ;HL -> skew table base address
001B 09                        DAD       B                ;HL -> physical sector
                                                          ;      entry in skew table
001C 6E                        MOV       L,M              ;L = physical sector
001D 60                        MOV       H,0              ;HL = Physical Sector
001E C9                        RET                        ;Return to BDOS
```

**Figure 3-5.** Data declarations for the standard skewtable for standard diskettes

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 (somewhat out of place) in the disk parameter block.

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

## 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.

```
                     DPBO:
0000 1A00                       DW      26          ;Sectors per track
0002 03                         DB      3           ;Bloqk shift
0003 07                         DB      7           ;Block mask
0004 03                         DB      3           ;Extent mask
0005 F200                       DW      242         ;Max. allocation block number
0007 3F00                       DW      63          ;Number of directory entries 1
0009 C0                         DB      1100$0000B  ;Bit map for allocation blocks
000A 00                         DB      0000$0000B  ; used for directory
000B 1000                       DW      16          ;No. of bytes in dir. check buffer
000D 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

| Allocation Block Size | Block Shift | Block Mask |
|---|---|---|
| 1,024 | 3 | 7 |
| 2,048 | 4 | 15 |
| 4,096 | 5 | 31 |
| 8,192 | 6 | 63 |
| 16,384 | 7 | 127 |

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

| Allocation Block Size | Number of Allocation Blocks | |
|---|---|---|
| | 1 to 255 | 256 and Above |
| 1,024 | 0 | (Impossible) |
| 2,048 | 1 | 0 |
| 4,096 | 3 | 1 |
| 8,192 | 7 | 3 |
| 16,384 | 15 | 7 |

**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.

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

| **Physical characteristics:** | |
|---|---|
| 77 | Tracks/Diskette |
| 26 | Sectors/Track |
| 128 | Bytes/Sector |
| 2 | Tracks Reserved for CP/M |
| 1024 | Bytes/Allocation Block |

| **Calculate:** | |
|---|---|
| 77 | Tracks/Diskette |
| − 2 | Tracks Reserved for CP/M |
| 75 | Tracks for File Storage |
| ×26 | Number of Sectors |
| 1950 | Sectors for File Storage |
| ×128 | Bytes per Sector |
| 249,600 | Bytes for File Storage |
| ÷1024 | Bytes/Allocation Block |
| 243.75 | Total Number of Allocation Blocks |
| 242 | Number of the last allocation block (rounded and based on first block being Block 0) |

**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.

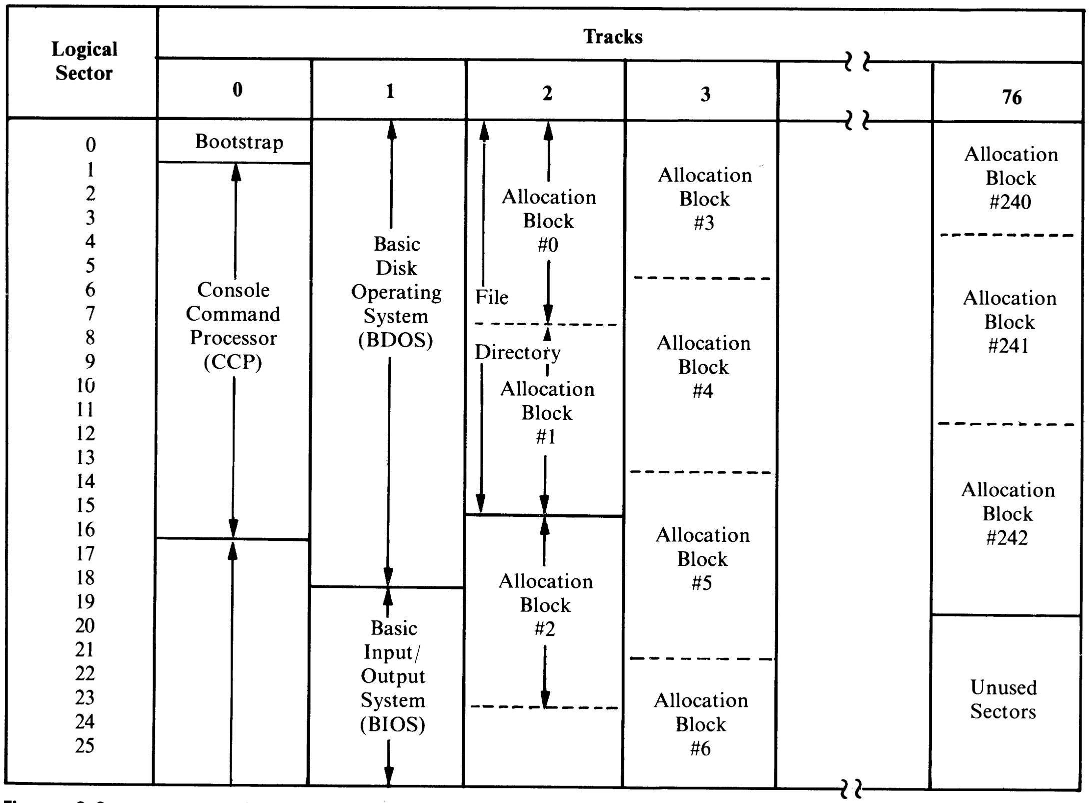


**Figure 3-8.** Layout of standard diskette

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

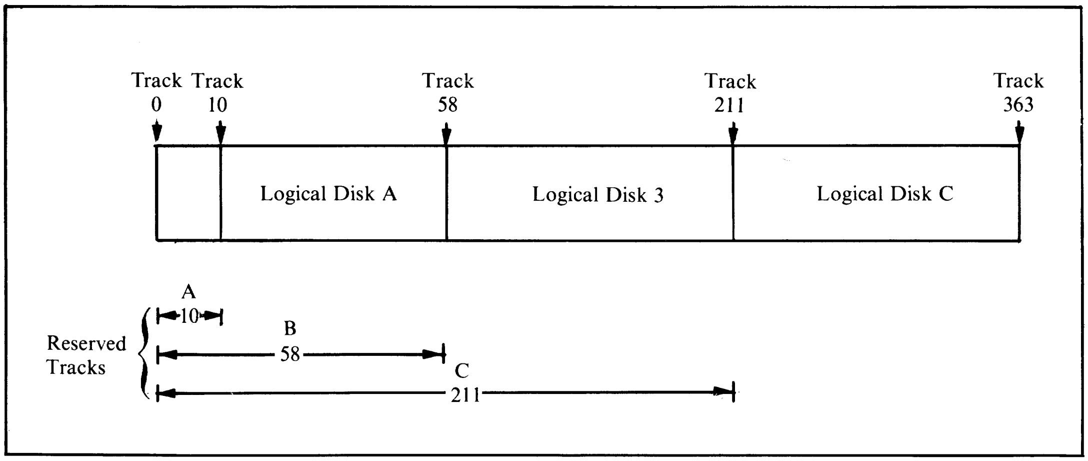


**Figure 3-9.** Dividing hard disks into logical disks

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 |

**Chosen Logical Characteristics:**

| | Tracks | Allocation Block Size |
|---|---|---|
| Reserved Area | 10 | n/a |
| Disk A: | 48 | 2048 |
| Disk B: | 153 | 4096 |
| Disk C: | 153 | 4096 |

**Calculate:**

| A: | B: and C: | |
|---|---|---|
| 48 | 153 | Tracks assigned to Disk |
| ×10,240 | ×10,240 | Bytes/Track |
| 491,520 | 1,566,720 | Bytes/Disk |
| ÷ 2048 | ÷ 4096 | Bytes/Allocation Block |
| 240 | 382.5 | Number of Allocation Blocks |
| 239 | 381 | Maximum Block Number |

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

```
DPBA:       DPBB:       DPBC:
    4           4           4           ;128-byte sectors/"track"
    4           5           5           ;Block shift
    15          31          31          ;Block mask
    1           1           1           ;Extent mask
    239         381         381         ;Max. all. block #
    255         1023        1023        ;No. of directory entries
    11110000B   11111111B   11111111B   ;Bit Map for allocation blocks
    00000000B   00000000B   00000000B   ; used for directory
    0           0           0           ;No. of bytes in dir.check buffer
    (10)        (58)        (211)       ;Actual tracks before directory
    200         1160        4220        ;"Tracks" before directory
```

**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 skullduggery. 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

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$TYP        EQU     9       ;File type
                                                ;Offsets for bits used in type
0009 =          FCBE$RO         EQU     9       ;Bit 7 = 1 - read only
000A =          FCBE$SYS        EQU     10      ;Bit 7 = 1 - system status
000B =          FCBE$CHANGE     EQU     11      ;Bit 7 = 0 - file written to
                                                ;
000C =          FCBE$EXTENT     EQU     12      ;Extent number
                                                ;13, 14 reserved for CP/M
000F =          FCBE$RECUSED    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$RANREC     EQU     33      ;Random rec. to read/write
0023 =          FCBE$RANRECO    EQU     35      ;Random rec. overflow byte (MS)
                ;
                ;
                ;
0000 00         FCB$DISK:       DB      0       ;Search on default disk drive
0001 46494C454EFCB$NAME:        DB      'FILENAME'      ;File name
0009 545950     FCB$TYP:        DB      'TYP'   ;File type
000C 00         FCB$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$ABUSED:      DB      0,0,0,0,0,0,0,0 ;Allocation blocks used
0018 0000000000                 DB      0,0,0,0,0,0,0,0
0020 00         FCB$SEQREC:     DB      0       ;Sequential rec. to read/write
0021 0000       FCB$RANREC:     DW      0       ;Random rec. to read/write
0023 00         FCB$RANRECO:    DB      0       ;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.

# 4 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 rubout 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 PAYROLLLLORYAPSALES
             ^^^^^^^
             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 PAYROLLLLORYAPSALES#
  RUN SALES_
```

The "#" character is output by the CCP to indicate that the line has been repeated. 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               SYSFILE does not show
B>DIR JUNK<cr>
NO FILE                 JUNK does not exist
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.

```
0000 =          RAM             EQU     0         ;Start of RAM (and the base page)
                                                  ;You may need to change this to
                                                  ; some other value (e.g. 4300H)

0000            ;               ORG     RAM       ;Set location counter to RAM base
0000            WARMBOOT:       DS      3         ;Contains a JMP to warm boot entry
                                                  ; in BIOS Jump vector table
                ;
0002 =          BIOSPAGE        EQU     RAM+2     ;BIOS Jump vector page
                ;
0003            IOBYTE:         DS      1         ;Input/output redirection byte
                ;
0004            CURUSER:        DS      1         ;Current user (bits 7-4)
0004 =          CURDISK         EQU     CURUSER   ;Default logical disk (bits 3-0)
                ;
0005            BDOSE:          DS      3         ;Contains a JMP to BDOS entry
0007 =          TOPRAM          EQU     BDOSE+2   ;Top page of usable RAM
                ;
0005C                           ORG     RAM+5CH   ;Bypass unused locations
                ;
005C            FCB1:           DS      16        ;File control block #1
                                                  ;Note: if you use this FCB here
                                                  ; you will overwrite FCB2 below.
                ;
006C            FCB2:           DS      16        ;File control block #2
                                                  ;You must move this to another
                                                  ; place before using it
                ;
0080                            ORG     RAM+80H   ;Bypass unused locations
                ;
                COMTAIL:                          ;Complete command tail
0080            COMTAIL$COUNT:  DS      1         ;Count of the number of chars
                                                  ; in command tail (CR not incl.)
0081            COMTAIL$CHARS:  DS      127       ;Characters in command tail
                                                  ; converted to uppercase and
                                                  ; without trailing carriage ret.
                ;
0080                            ORG     RAM+80H   ;Redefine command tail area
                ;
0080            DMABUFFER:      DS      128       ;Default "DMA" address used
                                                  ; as a 128-byte record buffer
                ;
0100                            ORG     RAM+100H  ;Bypass unused locations
                TPA:                              ;Start of transient program area
                                                  ; into which programs are loaded.
```

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

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 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

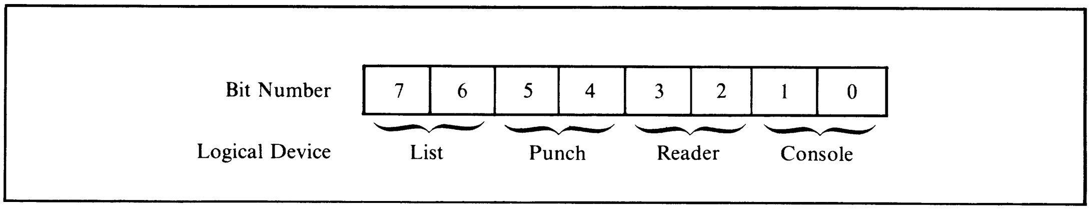


**Figure 4-2.** Arrangement of the IOBYTE

**Table 4-1.** IOBYTE Values

| Logical Device | Physical Device | | | |
|---|---|---|---|---|
| | 00 | 01 | 10 | 11 |
| Console (CON:) | TTY: | CRT: | BAT: | UC1: |
| Reader (RDR:) | TTY: | PTR: | UR1: | UR2: |
| Punch (PUN:) | TTY: | PTP: | UP1: | UP2: |
| List (LST:) | TTY: | CRT: | LPT: | UL1: |

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.typ 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.

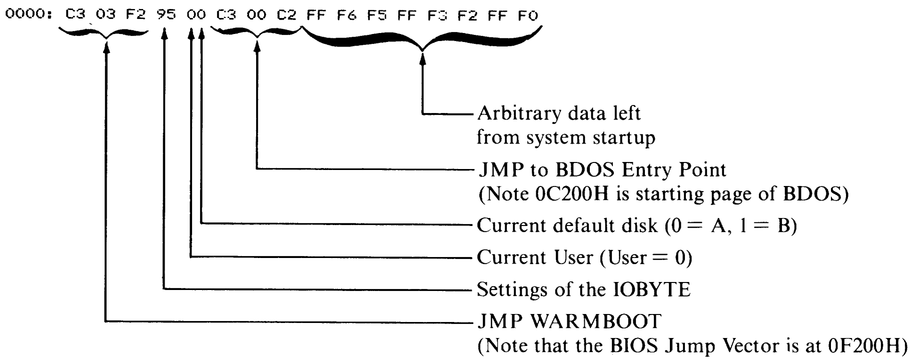
**TPA** This is the *transient program area* into which the CCP loads programs. The TPA extends up to the base of the BDOS.

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:



```
0000: 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 FCB1 and FCB2 areas will be set by the CCP as follows:

```
        Logical Disk                             Logical Disk
            |                                        |
            v                                        v
005C:  00 54 4F 46
        .  T  O  F
0060:  49 4C 45 20 20 54 59 50 00 00 00 00 00 46 52 4F
        I  L  E        T  Y  P  .  .  .  .  .  F  R  O
0070:  4D 46 49 4C 45 54 59 50 00 00 00 00 00 F2 34 F3
        M  F  I  L  E  T  Y  P  .  .  .  .  .  .  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 FCB1 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
            |
            |                                        Residue-----+
            v                                                    |
0080:  1F 54 4F 46 49 4C 45 2E 54 59 50 20 46 52 4F 4D           |
        .  T  O  F  I  L  E  .  T  Y  P     F  R  O  M           |
0090:  46 49 4C 45 2E 54 59 50 20 44 49 53 50 4C 41 59           |
        F  I  L  E  .  T  Y  P     D  I  S  P  L  A  Y           |
00A0:  00 43 52 43 4B 20 20 20 20 43 4F 4D 00 00 00 0A <---------+
        .  C  R  C  K              C  O  M  .  .  .  .
00B0:  9B 9C 00 00 00 00 00 00 00 00 00 00 00 00 00 00
        .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .
00C0:  E5 E5 E5 E5 E5 E5 E5 E5 E5 E5 E5 E5 E5 E5 E5 E5
        .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .
00D0:  E5 E5 E5 E5 E5 E5 E5 E5 E5 E5 E5 E5 E5 E5 E5 E5
        .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .
00E0:  E5 E5 E5 E5 E5 E5 E5 E5 E5 E5 E5 E5 E5 E5 E5 E5
        .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .
00F0:  E5 E5 E5 E5 E5 E5 E5 E5 E5 E5 E5 E5 E5 E5 E5 E5
        .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .
0100:  01 F9 <-----------+
                         |
                   Program Start
```

You can see that the command tail length is 01FH (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         START:    CALL    CTP         ;Test bed for CTP
0103 00                       NOP
                              ; Remainder of your program

                    ;         This subroutine breaks the command tail apart, placing
                    ;         each value in a separate string area.
                    ;
                    ;         Return parameters:
                    ;                 A = 0 - No error (Z flag set)
                    ;                 B = Count of number of parameters
                    ;                 HL -> Table of addresses
                    ;                         Each address points to a null-byte-
                    ;                         terminated parameter string.
                    ;                 If too many parameters are specified, then A = TMP
                    ;                 If a given parameter is too long, then A = PTL
                    ;                         and D points to the first character of the
                    ;                         offending parameter in the COMTAIL area.
                    ;
0080 =              COMTAIL         EQU     80H         ;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$PTL         EQU     2           ;Parameter too long error code
                    ;
                    PTABLE:                             ;Table of pointers to parameters
0104 0C01                     DW      P1          ; Parameter 1
0106 1A01                     DW      P2          ; Parameter 2
0108 2801                     DW      P3          ; Parameter 3
                                      ; <--- Add more parameter addresses here
010A 0000                     DW      0           ; Terminator
                    ;
                    ;         Parameter strings.
                    ;         The first byte is 0 so that unused parameters appear
                    ;         to be null strings.
                    ;         The last byte of each is a 0 and is used to detect
                    ;         a parameter that is too long.
010C 0001010101P1:            DB      0,1,1,1,1,1,1,1,1,1,1,1,1,0 ;Param. 1 & terminator
011A 0001010101P2:            DB      0,1,1,1,1,1,1,1,1,1,1,1,1,0 ;Param. 2 & terminator
0128 0001010101P3:            DB      0,1,1,1,1,1,1,1,1,1,1,1,1,0 ;Param. 3 & terminator
                                      ; <--- Add more parameter strings here
                    ;
                    CTP:                                      ;Main entry point <<<<<
0136 210401                   LXI     H,PTABLE            ;HL -> table of addresses
0139 0E00                     MVI     C,0                 ;Set parameter count
013B 3A8000                   LDA     COMTAIL$COUNT       ;Character count
013E B7                       ORA     A                   ;Check if any params.
013F C8                       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)

```
                CTP$NEXTP:                      ;Next parameter loop
0145 E3                 XTHL                    ;HL -> Table of addresses
                                                ;Top of stack = COMTAIL ptr.
0146 5E                 MOV     E,M             ;Get LS byte of param. addr.
0147 23                 INX     H               ;Update address pointer
0148 56                 MOV     D,M             ;Get MS byte of param. addr.
                                                ;DE -> Parameter string (or is 0)
0149 7A                 MOV     A,D             ;Get copy of MS byte of addr.
014A B3                 ORA     E               ;Combine MS and LS byte
014B CA8001             JZ      CTP$TMPX        ;Too many parameters--exit
014E 23                 INX     H               ;Update pointer to next address
014F E3                 XTHL                    ;HL -> comtail
                                                ;Top of stack--update addr. ptr.
                                ;At this point, we have
                                ; HL -> next byte in command tail
                                ; DE -> first byte of next parameter string
                CTP$SKIPB:
0150 7E                 MOV     A,M             ;Get next parameter byte
0151 23                 INX     H               ;Update command tail ptr.
0152 05                 DCR     B               ;Check if characters still remain
0153 FA7301             MB      CTPX            ;No, so exit
0156 FE20               CPI     ' '             ;Check if blank
0158 CA5001             JZ      CTP$SKIPB       ;Yes, so skip blanks
015B 0C                 INR     C               ;Increment parameter counter
                CTP$NEXTC:
015C 12                 STAX    D               ;Store in parameter string
015D 13                 INX     D               ;Update parameter string ptr.
015E 1A                 LDAX    D               ;Check next byte
015F B7                 ORA     A               ;Check if terminator
0160 CA7A01             JZ      CTP$PTLX        ;Parameter too long exit
0163 AF                 XRA     A               ;Float a 00-byte at end of param.
0164 12                 STAX    D               ;Store in param. string
0165 7E                 MOV     A,M             ;Get next character from tail
0166 23                 INX     H               ;Update command tail pointer
0167 05                 DCR     B               ;Check if characters still remain
0168 FA7301             JM      CTPX            ;No, so exit
016B FE20               CPI     ' '             ;Check if parameter terminator
016D CA4501             JZ      CTP$NEXTP       ;Yes, so move to next parameter
0170 C35C01             JMP     CTP$NEXTC       ;No, so store it in param. string
                ;
                CTPX:                           ;Normal exit
0173 AF                 XRA     A               ;A = 0 & Z-flag set
                ;
                CTPCX                           ;Common exit code
0174 E1                 POP     H               ;Balance stack
0175 210401             LXI     H,PTABLE        ;Return ptr. to param. addr. table
0178 B7                 ORA     A               ;Ensure Z-flag set appropriately
0179 C9                 RET
                ;
                CTP$PTLX:                       ;Parameter too long exit
017A 3E02               MVI     A,CTP$PTL       ;Set error code
017C EB                 XCHG                    ;DE -> offending parameter
017D C37401             JMP     CTPCX           ;Common exit
                ;
                CTP$TMPX:                       ;Too many parameters exit
0180 3E01               MVI     A,CTP$TMP       ;Set error code
0182 C37401             JMP     CTPCX           ;Common exit
                ;
0185                    END     START
```

**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.

```
0007 =          TOPRAM  EQU     7       ;Most significant byte of
                ;                               BDOS entry point
0000 3A0700             LDA     TOPRAM  ;Get MS byte of BDOS entry point
0003 3D                 DCR     A       ;Back off one page
0004 2EFF               MVI     L,0FFH  ;Set LS byte of final address
0006 67                 MOV     H,A     ;HL = XXFFH
0007 F9                 SPHL            ;Set stack pointer from HL
```

**Figure 4-4.** Setting stack pointer to top of available RAM

```
                ;       Use this technique only for CP/M utility programs.
                ;       MP/M programs do not permit this.
                ;
0009 =          CONIN   EQU     09H     ;Get console input character
                                        ; (It's the 4th jump in the vector)
0002 =          BIOSPAGE EQU    2       ;Address of BIOS page
                ;
                ;       At this point you make a direct CONIN
                ;       CALL...
                ;
0000 2E09               MVI     L,CONIN ;Get LS byte of CONIN entry point
0002 CD0500             CALL    BIOS    ;Go to BIOS entry subroutine
                        ;... the rest of your program...
                ;
                ;
                BIOS:
0005 3A0200             LDA     BIOSPAGE;Get BIOS jump vector page
0008 67                 MOV     H,A     ;HL -> entry point
                                        ;(You set LS byte before coming here)
0009 E9                 PCHL            ;"Jump" to BIOS
                                        ;Your return address is already
                                        ; on the stack
```

**Figure 4-5.** Making a direct BIOS call

```
                    ;       Note: This example assumes you have not
                    ;       overwritten the CCP.
                    ;
0100                        ORG     100H    ;Start at TPA
                    START:
0100 210000                 LXI     H,0     ;Save CCP's stack pointer
0103 39                     DAD     SP      ;By adding it to 0 in HL
0104 220F01                 SHLD    CCP$STACK
0107 314101                 LXI     SP,LOCAL$STACK
                    ;
                    ;       The main body of your program is here
                    ;
                    ;       ... and when you are ready to return
                    ;               to the CCP...
010A 2A0F01                 LHLD    CCP$STACK       ;Get CCP's stack pointer
010D F9                     SPHL                    ;Restore SP
010E C9                     RET                     ;Return to the CCP
                    ;
010F                CCP$STACK:      DS      2       ;Save area for CCP SP
0111                                DS      48      ;Local stack
                    LOCAL$STACK:
0141                        END     START
```

**Figure 4-6.** Returning to CCP at program end

## 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.)