CP/M

ASSEMBLY LANGUAGE PROGRAMMING

A GUIDE TO THE INTEGRATED LEARNING OF THE CP/M OPERATING
SYSTEM & ASSEMBLY LANGUAGE PROGRAMMING   KEN BARBIER

KEN BARBIER has more than thirty years of experience in electronics and computers and is self-employed as a writer and consultant. In addition to writing numerous articles on computer hardware, software, and applications, he has been involved in designing, constructing, and programming data acquisition systems for radio astronomy since 1969.

CP/M

ASSEMBLY LANGUAGE PROGRAMMING

Ken Barbier

A SPECTRUM BOOK

Prentice-Hall, Inc., Englewood Cliffs, N.J. 07632

This book is available at a special discount when ordered in large quantities. Contact Prentice-Hall, Inc., General Publishing Division, Special Sales, Englewood Cliffs, N.J. 07632.

CP/M is a trademark of Digital Research, Pacific Grove, CA 93950.
Teletype is a registered trademark of Teletype Corp., Skokie, IL.
DECwriter is a trademark of Digital Equipment Corp., Marlboro, MA 01753.
Z80 is a trademark of Zilog, Inc., Cupertino, CA 95014.

© 1983 by Prentice-Hall, Inc., Englewood Cliffs, N.J. 07632

A SPECTRUM BOOK

Printed in the United States of America

10 9 8 7 6 5 4 3 2 1

Production coordination and interior design: Inkwell
Manufacturer: Cathie Lenard

Contents

II *THE CP/M OPERATING SYSTEM*

III *8080 ASSEMBLY LANGUAGE PROGRAMMING*

IV *A LIBRARY OF USER SUBROUTINES*

V *DISK FILE ACCESS*

APPENDIXES

Preface

"I shall on all subjects have a policy to recommend."

  Ulysses S. Grant

Computer programs written in assembly language are alive and well and can be found in live-action arcade games, kitchen appliances, outer space, and the winners' circle at computer chess and Othello tournaments. Where compact object code and speed of execution are critical, assembly language has no equal.

Learning assembly language has never been easy. Higher level languages like BASIC, Fortran, and Pascal have been designed to be independent of the computer on which they are running and to communicate in words most understandable to human programmers. Assembly language, in contrast, forces the programmer to think like the machine and to become intimate with the hardware organization of the machine.

To achieve this, programmers of assembly language have to study the internal structure of their computer, learn its instruction set, and live within the constraints of its word size and mathemati-

cal capabilities. They have to learn a new set of instructions for each new computer, and they have to learn new operating procedures for the edit, assemble, and debug programs on each new computer.

In the past, before writing even the simplest program that communicates with the operator's terminal, the programmer had to know details of the hardware on which he was working—details like input/output port addresses, status word addresses, and status bit meanings. And these change from computer to computer.

The advent of the Control Program/Microcomputer (CP/M) operating system has greatly simplified the learning process for the beginning assembly language programmer. Using facilities provided by the operating system, the programmer can write routines that will communicate with input/output devices and mass storage units on any computer system running CP/M. These assembly language programs become "hardware independent" and therefore portable.

No matter what make and model computer is used, the CP/M assembly programmer will be working in a familiar environment. When tackling a new assignment, he or she can use previously created programs and subroutines that will greatly simplify the new work requirements. And the programmer will be working with system utility programs identical to those on the previous job. CP/M makes all computers look alike.

This book assumes that the reader has no previous CP/M or assembly language experience. It presents three major aspects of assembly programming under CP/M: (1) an understanding of the facilities and operation of CP/M and its utility programs; (2) an understanding of the internal organization and instruction set of the 8080 family of microprocessors; and (3) an understanding of the proper design of assembly language programs.

Whether the reader is a programmer in a higher level language, an engineer, student, hobbyist, or just someone who needs to make a computer control the real world in real time, he or she will be able to learn all the fundamentals from this book. Since the reader will be learning to use intimately integrated hardware and software facilities of the microcomputer, the presentation of topics in this book is also integrated.

From the first simple exercise in the Introduction, the reader will be concurrently learning the details of the computer hardware:

how to edit, assemble, and debug programs, and how to interface those programs to the operating system. Since this requires that the reader learn a lot of background material before beginning to write programs, every effort has been made to present this material in an informal, entertaining style. Historical references are made wherever they will help to explain a subject like binary numbers or to account for strange, archaic names applied to modern devices.

A few exercises are included that should be performed by all readers. Other exercises are suggested, and the reader is free to experiment with a CP/M based computer at any time in the learning process. It can't be damaged from the operator's console.

When the background material has been absorbed, the reader will be using the newly acquired knowledge to build up a set of library subroutines that will be useful in any future programming efforts. The editing, assembling, and testing of this library and the demonstration programs that make use of it constitute the majority of the exercises required of the reader. When these tasks are finished, the reader will be ready to begin designing and writing new and wonderful assembly language programs. Suggestions for future projects are included.

Integrated with the more rigorous topics are comments, suggestions, rules, and edicts aimed at making the reader aware that a properly constructed program requires more than just stringing together a bunch of instructions that operate correctly. The goal of these bits of advice is to instruct the programmer in the construction of readable, modifiable, portable programs. The world already has a sufficiency of the other kind.

Personal opinions like the above are mine, and I accept all responsibility for them, but I am not at all reluctant to impose them on the defenseless reader.

KEN BARBIER
Borrego Springs, California
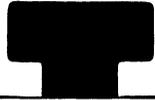
THE COMPUTER SYSTEM
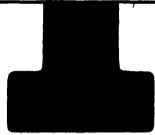
Introduction

Even if you have never written a line of assembly language pro-
gramming, sit down at the console of a CP/M based computer and
key in the following routine. Your keystrokes are those in boldface;
the other characters are displayed by CP/M. The "cr" symbol rep-
resents your pressing the RETURN key on the console. Don't
worry about what the numbers mean for now.

A> **DDTcr**
DDT VERS 1.4
-S100cr
0100 01 **0Ecr**
0101 B6 **02cr**
0102 0F **1Ecr**
0103 C3 **24cr**
0104 3D **CDcr**
0105 01 **05cr**
0106 43 **00cr**
0107 4F **C3cr**

```
0108 50 00cr
0109 59 00cr
010A 52 .cr
-G100cr
$
A>
```

Here you have used CP/M's Dynamic Debugging Tool (DDT) to key in and execute a machine language program that displayed the "$" on the console, and then returned to the CP/M operating system. There are times when keying in such a routine might be handy for testing parts of a computer. We could have sent the "$" to another peripheral device, for instance, as a quick test of its operation.

This same routine, written in assembly language, would look like this:

LISTING I-1. Assembly language version of the demonstration program.

```
                BDOS    EQU     5
                WCONF   EQU     2
                        ORG     100H
                        MVI     C,WCONF
                        MVI     E,'$'
                        CALL    BDOS
                        JMP     0
                        END
```

The use of mnemonics, like "JMP" for "jump," makes assembly language source code easier to read. Labels like "WCONF" for "Write-on-the-Console Function" make the source program more understandable.

When this source program is keyed into a disk file using CP/M's ED.COM and is assembled by CP/M's ASM.COM, it will produce the same machine language code that you typed in manually using DDT. With a short little routine like this, it might be quicker to use DDT and machine language to perform this simple function, but for any program of practical length, the editor and assembler provide the most error-free method of program generation.

Learning by doing

The purpose of this exercise is to illustrate the method by which you will be learning assembly language programming and the CP/M operating system. You will be writing, editing, assembling, and debugging programs that interface with CP/M, as this one does. This will provide you with the opportunity to see the results of your learning efforts as you go along.

As with this example, your programming efforts will begin with routines that output characters to, and read characters from, the operator's console. After mastering techniques for interfacing with input/output (I/O) devices like the console, you will be writing programs that read and write disk files. Your learning of assembly language programming will be integrated with learning the internal structure of the CP/M operating system and how to interface with it.

This book does not include any rigorous treatment of number systems, binary arithmetic, or Boolean algebra. With modern microcomputers, higher level languages are readily available for mathematical operations. You will need to become familiar with hexadecimal notation and simple logical operations, of course, and instruction in these topics is integrated with the other subjects so you won't have to struggle through a separate section devoted to number theory.

Why assemble?

Assembly language programming is not dead. The ready availability of higher level languages for microcomputers has relegated assembly programming to those application areas where it is indispensable: intimate interfacing with hardware, and for size- and time-critical operations.

Assembly language is still useful for writing programs for dedicated controllers where program size must be minimized to reduce costs. It also has applications where speed of execution is of primary importance, as in animated displays for video games or flight simulators, or in controlling high-speed machines like line printers.

And, as we will be seeing, learning assembly language programming can be both easy and enjoyable, given a friendly environment. CP/M has provided that environment. As our little exercise above has shown, it was possible to write this text and include exercises like these that can be run on any make or model of microcomputer, so long as it is running the CP/M operating system. This standard operating system has made this book possible.

In keeping with an encouraging, simple environment in which to learn, this book has been written in an informal style. This means you will have to put up with some bad puns and old jokes from time to time. In keeping with the informal style, the word *data* is herein used as a collective noun, avoiding such archaic constructs as "if those data were a zero."

Required equipment

In addition to this book and a CP/M based computer, you should have access to the manual set that accompanied the CP/M operating system. One complete copy of the CP/M system disk and a nice fresh clean empty disk should be dedicated to your exercises and experiments. You will also want a copy of the "8080/8085 Assembly Language Programming Manual," publication number 98-940, available from Intel Corporation, Literature Department, 3065 Bowers Avenue, Santa Clara, CA 95051, for $17.

The Intel manual will not be required for you to start learning the 8080 instruction set, but you should order one as soon as possible. It is to microprocessor instructions what a dictionary is to English words: a reference work to consult whenever you are not really sure you understand just what an instruction is doing.

With these tools in hand you will be ready to learn how to program in assembly language. Enjoy yourself. It is a lot of fun to make the machine obey your every wish.

Hardware Components Of the Computer System

A good name is better than precious ointment.

  Ecclesiastes

One of the greatest difficulties to be overcome by the newcomer in any technical field is getting used to all the new terms. Technical fields in particular develop jargons all their own. As if learning a whole new vocabulary were not enough of a problem, the beginner soon discovers that there are several different names for almost every item he will be learning about.

Defining terms

For instance, the word *terminal* is applied to the giant building where we are searched before boarding a plane, to the TV tube with a keyboard attached that we use to communicate with a computer, and to a little round metal loop crimped on the end of a

7

piece of wire inside that computer. Since we will try not to go flying off in all directions, and since we are not going to be looking inside computers in this book, we should have little trouble keeping track of the fact that *terminal* in this context refers to the device through which we will be communicating with our computer.

But that terminal is referred to by a number of other names as well. It will sometimes be called a console, a screen, a CRT, a VDT, a CON, and a TTY. If, on your particular computer, it happens to be composed of two parts, the terminal may be referred to as a monitor and a keyboard. And a "Monitor" is also a particular type of computer program. And so the confusion is propagated.

These examples are, of course, just the tip of the iceberg. Since the field of human endeavor that we are going to be exploring does have so many conflicting, overlapping, and duplicated terms, we will be selecting a subset of those terms in an attempt to overcome some of the confusion. As we proceed through this book, we will be defining the terms that we will be using, and we will be avoiding the use of synonyms as much as possible.

Since the rest of the computer world has not standardized on such a neat subset, it will be necessary to be aware of all the other words which refer to the same items in our set. This requires that, once we have become familiar with our own set of terms, we will have to be aware that others with whom we talk will be using different terms for the same items. Some of these different terms will be defined in appropriate places throughout this book.

The list of terms we will be using in this book has been chosen to be as close as possible to the words used in the manuals supplied with the CP/M operating system. Since the language we will be studying is the assembly language for the 8080 microprocessor, as defined by the Intel Corporation, we will also be including words compatible with the usage found in the Intel 8080/8085 Assembly Language Programming Manual.

In the section following, we will be looking at the components of a computer system, defining our terms, and building up the vocabulary to be used throughout this book. Even if you are already familiar with your computer and the CP/M operating system, it might not be a bad idea for you to read through the following section anyway, in order to get familiar with the words we will be using.

The computer operator

There are two classes of humans to be found sitting at computer terminals: computer users and computer programmers. Sometimes they are easy to tell apart. The user is the seven-year-old battling Klingons. The programmer is the long-haired, unshaven, bleary-eyed creature mumbling to himself. The distinction is made here because there are so many programs intended to be used by mere mortals (as opposed to programmers) that are so poorly designed that only a programmer can run them. One of the most important lessons you will have to learn is to always keep in mind that your programs not only have to work, they have to be usable by mere mortals.

The time to start thinking of how to make your programs more usable by nonprogrammers is right now, before one line of code has been written. The more programs you write, the more you will learn that what was obvious to you last year has now become a forgotten detail. There is nothing more frustrating than to be unable to run a program you yourself have written. It happens to all of us, too.

The CP/M operating system provides us with the ability to name programs, and call up the desired program by its name. All we fallible humans have to remember, then, is what program "NAME" will do when it does run. If that program has been properly written, once CP/M has loaded it and executed it all required operator inputs will be explicitly prompted for.

This approach may seem silly to you now. Surely you won't forget how to operate your own program! Yes, you will. It happens to all of us. So keep in mind right from the beginning that every program worth writing is worth writing properly. And the first step is to make every program usable by the proverbial "unsophisticated user."

After reading this book, you will surely become an expert programmer who can understand the inner working of any program ever written, but keep in mind that your creations should be usable by anyone, computer user or computer programmer.

Please note that we have now established definitions for "terminal," "computer operators," and the two subspecies "computer users" and "computer programmers." That didn't hurt, did it?

FIGURE 1-1. The hardware components typically included in a small microcomputer system. The smallest system that can run the CP/M operating system is sufficient for use in completing all of the exercises in this book.
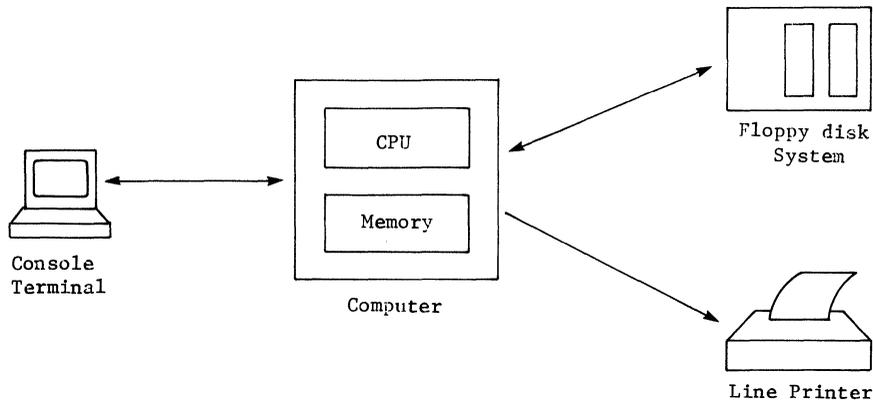
The operator's console

The very earliest computers had operator's consoles consisting of rows and rows of lights, switches, and patch cords. Programs were input in the form of patterns of patch cords in plug boards, or were keyed into switches bit by bit. When the operator was moved one step back, and could communicate with the machine from a terminal device, the term "console" went along with him. Some computers of all sizes still include a switches-and-lights type of console, but we will not consider that type of console in this book since the context in which we are working assumes the existence of a console terminal.

Back in those good old days the standard computer terminal was the ASR-33 from the Teletype Corporation. About 300,000 of these mechanical monsters have been produced, most of which are used for sending messages in the telex network.

Because the ASR-33 was inexpensive, rugged, and included a paper tape punch and reader in addition to its keyboard and printer, it became the standard of the minicomputer industry as the computer operator's console. Five years ago there were no inexpensive CRT terminals, so all little computers came with an interface suitable for the TTY. Since "ASR-33 Teletype" is a bit

FIGURE 1-2. A "switches-and-lights" type of computer front panel console. No longer seen very often, this type of console permitted the computer operator or programmer to access each bit within a data or address word. Data was switched in one bit at a time, and could be displayed one bit at a time. While this type of operation is no longer necessary, it did make it easy to visualize bit patterns within computer words.
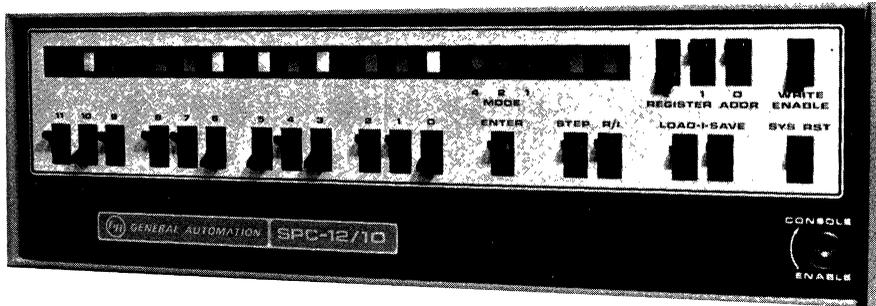
long-winded, this ubiquitous terminal is called, in short, the "TTY," as were others of its predecessors.

When the inexpensive microprocessor invaded the earth in 1975, the TTY was still the most inexpensive method for communicating between human and machine. Some early microcomputers and most minicomputers wouldn't accept any other device as their operator's console in those ancient times. And in those days before floppy disks and CP/M, the paper tape punch and reader sections of the TTY provided the only means of program and data storage and retrieval on the smallest computers.

The microprocessor itself, which provided the basis for inexpensive computers, also brought about a revolution in the terminal industry. With this smart integrated circuit (IC), it was possible to build a terminal using an electronic keyboard instead of the TTY's maze of motor, clutches, levers, and noise. The printing mechanism of the TTY was replaced by the silent screen of a TV type cathode ray tube (CRT). The slow paper tape punch and reader on the TTY have been replaced by the floppy disk, and now our CRT terminal (or just "CRT" for short) provides us with reliable, silent, and forgetful communications between human and computer. Now just what did I do forty lines back that caused all this trouble? With the TTY, we could always pick up the paper from the floor and see what we did wrong!

FIGURE 1–3. The ASR-33 from Teletype Corp. Once the standard terminal for small computers, this slow, noisy mechanical device has left a legacy in the device type designations still found in the CP/M operating system. The ASR-33 was the source of such terms as TTY, RDR, PUN, PTR, and PTP still referred to by CP/M's STAT and PIP utilities.
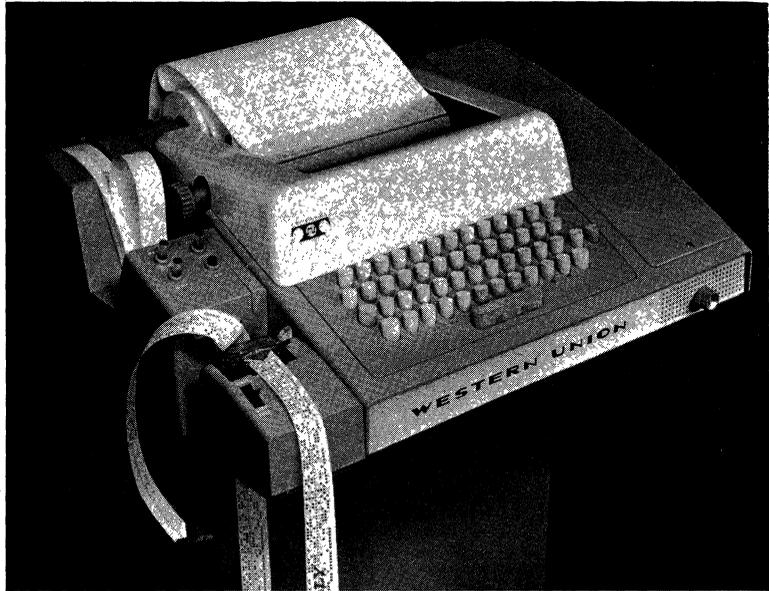
Our more modern CRT doesn't give us that opportunity, but at least now we know all about "console," "ASR-33," "TTY," and "CRT."

The computer

Our human operator, sitting at his console terminal, is communicating with a computer. Computers come in all sizes, from giant "mainframes" through "mega-mini's" and just plain old "minicomputers" down to our lowly microcomputer. At the heart of each of these machines is a section of hardware designated the central processing unit, or CPU. This designation dates from the days when the CPU was a separate rack stuffed full of printed circuit boards and heat. Another whole rack was needed to hold 16K

words of memory. Later, as integrated circuits replaced discrete devices (transistors, resistors, etc.), and became smaller and more complex, it became possible to package a complete computer in a single rack only six feet high. Now, of course, complete computers can be held in one hand. Even so, inside each computer is a CPU.

It is the CPU that processes the data. Whether the CPU is only a small portion of a single integrated circuit, or is a single integrated circuit, or is a rack full of printed circuit boards, it is still that part of the overall system that does all the manipulating of data. The data may come from some other section of the computer, and after processing be transferred to yet another, but the real work is the task of the CPU.

The 8080 and its relatives

In our CP/M based computer system, the CPU is one member of the 8080 microprocessor family, for CP/M is a program written for the original 8080. The Intel 8080 microprocessor was not the first micro, but its predecessors were so restricted in computing power that their usefulness was limited to that of smart controllers, and only a few brave souls tried to do any real computing with them.

The 8080 changed things suddenly when it became readily available in early 1975. Here was a CPU contained in a single integrated circuit package, selling for little more than $100 (in 1975; now less than $5), that executed an instruction set powerful enough to support real data processing.

It was the instruction set of the 8080, rather than the chip itself, that became an industry standard. As we will be seeing, the 8080 executes enough instructions to be both useful and easily programmable. Advocates of other microprocessors will be quick to point out the deficiencies in the 8080, proving only that you can't satisfy all the people all the time. While not perfect, the 8080 instruction set is easy to learn and easy to use. And it has become the industry standard.

The 8080 integrated circuit itself had more serious shortcomings. It was relatively slow, required three different power supply voltages, and needed a couple of extra ICs to provide clocks and system control. Retaining the 8080 instruction set, Intel later produced the 8085 microprocessor.

The 8085 greatly simplifies things for the hardware designer. Simply connect a single +5 volt DC power supply, connect either a crystal or a resistor-capacitor (RC) network between pins 1 and 2, and the 8085 is ready to run. Of course it will need memory and input/output (I/O) devices as well, but all micros require them. The real improvement provided by the 8085 is in the simplification of hardware design, and a great increase in operating speed.

While they are not produced by Intel, there are other members of the 8080 family that we will be looking at in some detail in Part III. What we should keep in mind at this point is the fact that CP/M is written using the 8080 instructions, so some member of this family must be our CPU.

Instructions in memory

No CPU can operate unless we feed it instructions to execute. These operation codes (opcodes) will be stored in memory, and the CPU will fetch them from memory one at a time, decode the operation requested, execute it, and fetch the next instruction in turn. This set of instructions placed in memory in a meaningful sequence (we hope!) constitutes a machine language program. The CPU is a machine, so the opcodes have to be in a format understandable by a machine. In the case of the 8080, that format is "bytes" consisting of 8 binary digits, or "bits," apiece.

So, you might ask, how do we get the opcodes into memory to begin with? The original technique involved keying in opcodes using one of the switches-and-lights consoles. This is slow, error prone, and a great impediment to progress. It wasn't long before all microcomputers included a small program in read-only memory (ROM) that would load our program automatically. This loader program would read some input device, like the paper tape reader on the TTY, and place the instructions it found into read-write memory, from which they could later be fetched and executed.

In computers with a dedicated purpose, such as controllers or hand-held digital games, all of the software required can be permanently placed in ROM. In a general purpose computer, we have to be able to change the program and work with varying data. Bit patterns which will be changing have to be stored in read-write memory, or "RAM." Why "RAM" for Read-Write Memory, in-

stead of "RWM?" Because that's the way it has always been. RAM is an acronym for Random Access Memory. The distinction between random access and sequential access memory (like our paper tape) was made decades ago, and we are stuck with an inexact acronym.

Just because it is not the best term we can't arbitrarily change it. We still have to be able to communicate with other computer users in terms they will understand, so RAM it is.

While we are on the subject of acronyms, two more you will be needing refer to the major classes of ROM. A PROM is a Programmable ROM. This is an IC initially fabricated with no stored program. By a process of fusing internal connections, we can "burn" a desired bit pattern (our program) into a PROM. This is fine once we are sure there are no errors in the program. Once burned, the program stored in the PROM is there to stay. A more useful device, and more expensive, is the EPROM. This Erasable PROM allows us to store a program in an IC, test it, and later erase the bit pattern and start over, if necessary.

There you have the basic components of your computer. The CPU is one member of the 8080 family of microprocessors. A loader program is stored in ROM (either a PROM or EPROM) and will read our program into RAM. To accomplish this, of course, we will need some kind of I/O device that communicates with mass storage. Read on.

Mass storage

When the characteristics of the first real digital computer were originally specified, it was decided that 4,000 words of storage would be enough to provide for any conceivable computation. Of course, each of those words was 40 bits long, and the machine was intended for calculation only.

To implement our microcomputers, we have since bitten those early long words into more manageable 8-bit bytes. When we need to calculate with similar precision, we just take a number of bytes to chew on at one time. And we have since learned to laugh at a computer with only 4K of RAM. That's just a toy!

Our 8080 family of micros can directly address 65,536 bytes of memory. Since we work with binary numbers, we think in terms of

powers of two. We will be looking into this in detail in Part III. For now, just remember that two raised to the tenth power is 1024, and using K (for the Greek "kilo") to designate thousands, 1024 bytes of storage is abbreviated "1K." Sixty-four of these increments is all the 8080 can address, and $64 \times 1024 = 65,536$. Or 64K, for short.

Main frame computers and even minicomputers do not have such a restricted addressing range, but their owners have restricted purchasing power, and high-speed main memory is expensive. The need for lower cost "mass storage" is as old as the computer itself, and this term is just as ancient.

Mass storage refers to any type of external memory: tapes, disks, drums, or even RAM when it is accessed at addresses outside the main memory address space. On our CP/M based microcomputer, we have typically two floppy disk drives for mass storage, with anywhere from 70K bytes (on 5¼" minifloppies) to a couple of million bytes (M bytes) available on each drive.

As you might guess, main memory inside our computer is addressed byte by byte using numbers from zero to 65535. Out on our mass storage device our memory locations are not so easily organized. It wouldn't be practical to try to keep track of millions of bytes of memory if each byte had its own unique address.

Disk addressing

Data on disks is recorded in a number of circular tracks, with each track broken up into a number of sectors. Each sector will have its own address, such as "track 14 sector 23," and the data stored in one sector will be not one byte but a string of data bytes. These strings are referred to as records, and each record in the original CP/M system contained 128 bytes. Double and quad density floppies and hard disk drives may use other sizes of sectors, but we will not have to concern ourselves with these details.

Neither will we have to remember that the data we want is on disk drive 2 at track 32 sector 14. It is one of the functions of our CP/M operating system to keep track of mass storage addressing details. As computer users or programmers we will be creating named files on our mass storage devices, and the operating system will handle the disk space allocation. All we, or our programs, will have to keep track of is the file name and the drive it is on.

FIGURE 1–4. Data organization on a typical eight-inch single density floppy disk. Each of 76 tracks contains 26 sectors storing 128 eight-bit bytes of data. Each sector can be identified by the disk controller by reading the identification (ID) information contained in the address field that precedes the data recorded in the sector.
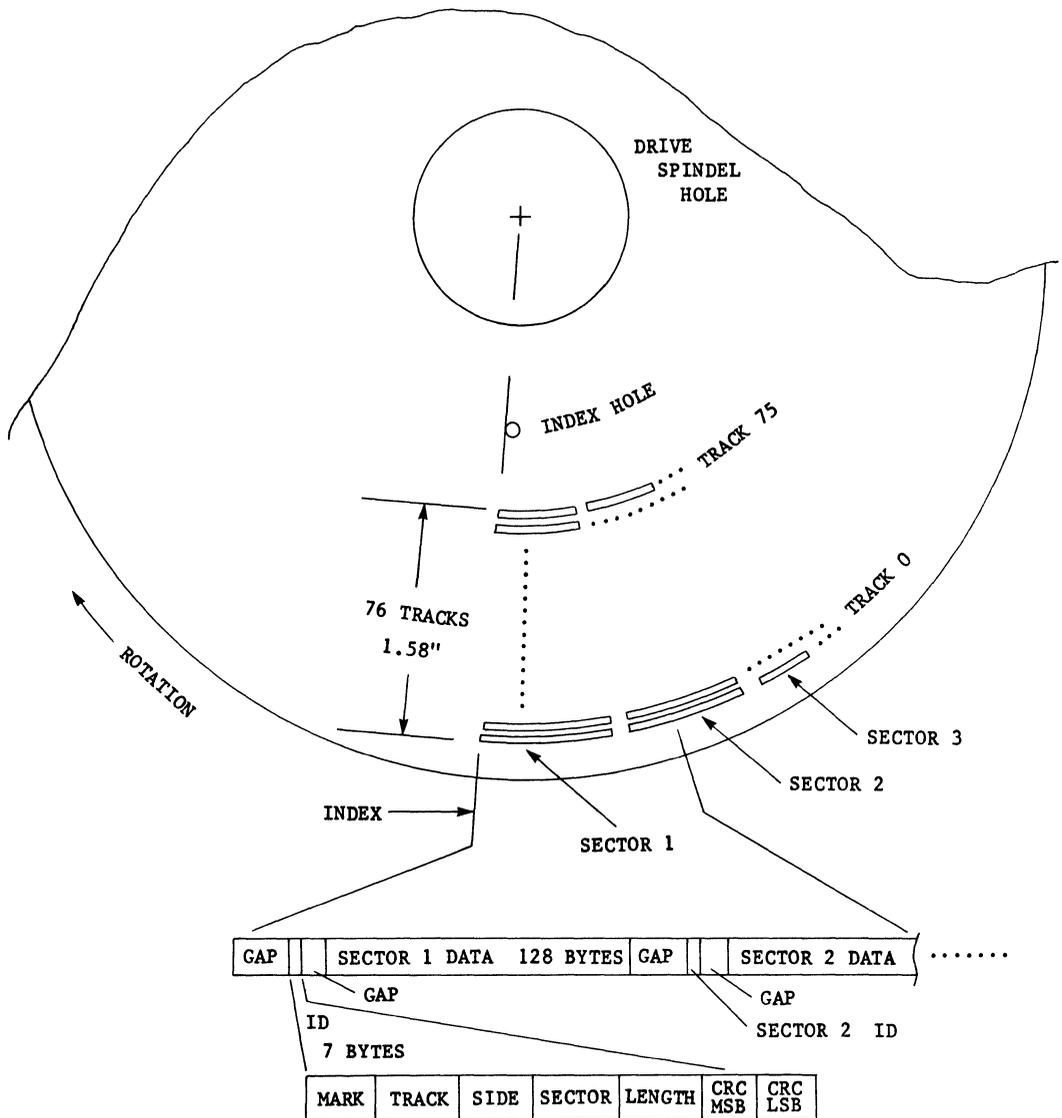
While it is nice to know some of the details of mass storage organization, thanks to CP/M we will not have to remember or work with these details. We are living in a truly enlightened age!

Hard copy

Since the objective of this book is teaching assembly language programming, and since assembly language programs are typically long and detailed, it will be virtually impossible to operate without some kind of hard copy peripheral. Back in the days of the TTY, the hard copy we needed was unavoidable. You got a printed copy of everything you typed and everything the computer output to the console. Mistakes and all! Nowadays, when you hear the boss coming, you can scroll all the error messages off the top of your CRT screen. Neat!

The TTY and its descendents, like the Decwriter and similar printing terminals, print one character at a time. As we key in our messages to the computer on a printing terminal, we will see each keystroke echoed on the printer. Output from the computer will appear to be printed a line at a time, but only because the computer can type faster than we can.

A line printer, in contrast, is built in such a way that it is incapable of printing one character and then stopping. It will receive characters and store them in a buffer until it receives a termination character, usually a carriage return (CR). When it sees this terminator, it will print the entire buffer in one pass of the print head.

Line printers matching this description print at rates of from 50 or so to about 300 characters per second. Other types of line printers use mechanisms other than a moving print head, and can use up paper at astonishing rates. Trees hate these high speed line printers.

In our typical CP/M based computer system we will assume the presence of one of the lower priced line printers. The device can in reality be the printer portion of your console terminal, if your system is so configured, but it will be considered to be a different device when we get to the discussion of device names in Chap. 3. For now, just keep in mind that references to the line printer are different than references to the printer on the console.

Some CRT terminals have a printer port built into them, and some all-in-one computer systems include a function known as screen printing. A screen printer, or a screen printing function using the line printer, allows you to save the contents of your CRT screen on a hard copy device. While this can be a handy technique for recording your mistakes for posterity, it is not a function built into CP/M, so we will not assume that our example system includes this tattle-tale.

Other peripherals

With its operator's console, CPU, memory, mass storage, and a hard copy device, our CP/M based microcomputer is complete and ready to perform. What other peripherals could we need?

If our floppy disk system conforms to one of the standard formats, we could exchange programs and data with any other computer, large or small, conforming to the same standard. This will require transporting the disk between computers. Floppy disks are ideal for this, as they can be mailed.

Too often, however, we will find a need to input data from some source that does not have the capability of writing that data onto a compatible disk. We would then require an additional input device. The old standby is the paper tape reader. In addition to the reader on the TTY, which clanks along reading 10 bytes per second, there are other types available that can read paper tape at up to several hundred bytes per second. The complement of the paper tape reader is, of course, the paper tape punch, the old standard output device. These also come in various speeds, but cannot match the speed of the fastest readers, since more mechanical action is required to punch a hole in paper than to simply detect its presence.

One reason for mentioning these two old-fashioned slow devices in the same context as our modern high speed CP/M based computer is that, even if you never see either of them, you will be encountering their names. The CP/M operating system was itself generated on a computer that expected reader and punch to be the most common input and output devices. Use of these device names as the default names for I/O devices was inherited from this machine. And we have all been sorry ever since.

Being able to read paper tape reduces our dependence on floppy disk compatibility, but it is still limiting. Perhaps the most universally compatible method of data exchange is the modem. "Modem" is a contraction of "modulator-demodulator," which still doesn't tell us much about the device. A modem is a device which enables us to communicate with other computers over a telephone line. It does this by modulating a carrier tone with our data in the form of a bit stream at the send end, and demodulating the bits from the carrier at the receive end.

With a modem connected to a computer on each end, and a telephone circuit between, we can transfer data between any two computers. Provided, of course, that the bit patterns used to represent each character in the data stream are the same in both computers. Here standardization has been achieved. We have a standardized code, ASCII, which stands for American Standard Code for Information Interchange. More on this subject later.

Other types of I/O devices are available in almost unlimited number. Those that can "look like" a modem (to the computer, anyway), and communicate in ASCII, can be hooked onto almost any computer. Most, however, have specific interfacing conventions that suit them for use on one make of computer only. But we need not worry about them now, as we are about to see.

A simple computer system

The computer system shown in Fig. 1-1 is the minimum hardware configuration required for the exercises in this book. It is assumed that the reader has access to such a system, with at least one floppy disk drive. Only the minimum 16K RAM will be required, and either some type of hard copy device or unlimited patience will be necessary.

In the discussions that follow, we will be examining the CP/M operating system in some detail, and then looking at the 8080 microprocessor as it appears to the programmer. With this background material behind us, we will start to do some simple assembly language programming. We will be learning this language by writing, editing, assembling, and debugging programs. This should provide a much superior learning environment than the traditional method of exhaustively studying a computer's instruc-

tion set and hardware configuration before beginning the first program.

In this chapter we have learned some of the terms we will be using in discussing our computer system and how to program it. We have looked at the components of a computer, and defined the minimum system required for proceeding with the mastery of assembly language programming under the CP/M operating system.

Software Components
Of the Computer
System

RE-DO FROM START
   BASIC language error message

"Hardware" refers to those parts of the computer system that make
dents in the floor when you drop them. Computerists long ago
decided that the more expensive half of the computer system, the
programming, would be called "software." Not to be confused with
softwear.

   Until the middle '70s most computer main memory was con-
structed using magnetic cores. Core memory can retain its con-
tents even with power off. It took a programming error to wipe out
all the contents of a core memory. When that happened, it was
back to the switches-and-lights console. A new copy of a loader
program then had to be reentered into the computer, and would in
turn be used to read in the operating system from some mass
storage device.

   Since keying in a loader in this manner was very time-con-
suming, this loader program was written to be as short as possible.

No error checking frills were included. To insure that the operating system was loaded properly, it was customary for this simple, short loader to first read in a smart loader, that would then load the system. The minimum loader came to be called a "bootstrap" loader, since it allowed the system to pull itself up into memory by the bootstraps.

The more inexpensive semiconductor memory rapidly replaced core memory, and the most obvious failing of this new technology is that semiconductor RAM loses its memory when the power goes down. Even a little noise on the power line, or a sag in line voltage, can wipe out the contents of semiconductor RAM. Then it is back to the console and key in the bootstrap again.

Until manufacturers began putting the bootstrap loader into ROM, that is. As EPROMS became available at reasonable prices, the rows and rows of switches and lights began disappearing from the front panels of microcomputers. Today, it is the rare exception that includes this technology on the front panel.

Since "software" referred to the programs that were getting wiped out all the time, a new term was needed to refer to the contents of ROM. The ROMs contained programs, and therefore software, but it wasn't as soft as the programs in RAM. Hence the coined word "firmware." That is, software made harder by being burned into ROM.

With these terms in mind, let's look at the soft- and firmware components of our computer.

Firmware monitor

When a microprocessor IC of the 8080 family is reset, which occurs automatically on power up and can be accomplished manually in case of disaster, it begins operation by fetching an instruction from memory location zero. As we will see later, in a CP/M system the low end of main memory address space must contain read-write memory. If the CPU wants to fetch its first instruction from location zero, and CP/M wants RAM at location zero, and our RAM forgets everything when power is off, how do we ever get our computer to start up from cold?

Microcomputer designers had to resort to a hardware trick. A bootup circuit is activated by the same reset signal that starts the

CPU. This circuit makes the RAM at location zero "disappear," and substitutes a "shadow PROM." Depending on the make and model of computer, one or more instructions are fetched from the shadow PROM and executed. At some point in this execution sequence, often immediately following the first instruction, the computer hardware is told that it is time to disconnect the shadow PROM, and reinstate RAM at location zero.

In the simplest implementation of this procedure, the first instruction that the CPU fetches from the shadow PROM is an unconditional jump to the beginning of a monitor program in ROM. This monitor ROM is usually located at the very top of the main memory address space. When the CPU decodes this jump instruction, it knows that it should fetch its next instruction from the location jumped to. The CPU will begin its next instruction fetch sequence by placing this new address on the computer's address bus.

The address bus is the set of sixteen signal lines that contain the bit pattern of the address of the next memory location to be accessed for read or write. The bootup circuitry has only to detect that the most significant bit of the address bus has been asserted. This circuit then disables the shadow PROM, and reenables RAM at the bottom of memory.

We do not want to get bogged down in hardware details in this book, but this discussion is included here as the reset and bootup procedures are pertinent to understanding the operation of our computer. All we as computer operators will be aware of is that turning power on, or hitting the reset switch, will get our machine up and running.

We do not have to *hit* the reset switch. It is sufficient to press it gently. But when a program blows up and manual reset is necessary it is customary to want to hit something, so these switches get a real workout.

The result of this reset sequence is that the CPU begins fetching instructions from our monitor PROM. "Monitor" is another old computer term that is less than enlightening in modern context. Your computer may or may not have a monitor program in the classical sense. Traditional monitor programs use the console to communicate with the operator, and provide routines that enable him to interact intimately with the computer hardware, as is necessary for diagnosing hardware failures and debugging assembly language programs.

FIGURE 2-1. A flowchart showing the sequence of operations involved in starting up a typical microcomputer. While most of these operations are transparent to the operator, assembly language programmers working closely with the hardware and software components of the computer have to be familiar with this type of bootup sequence.
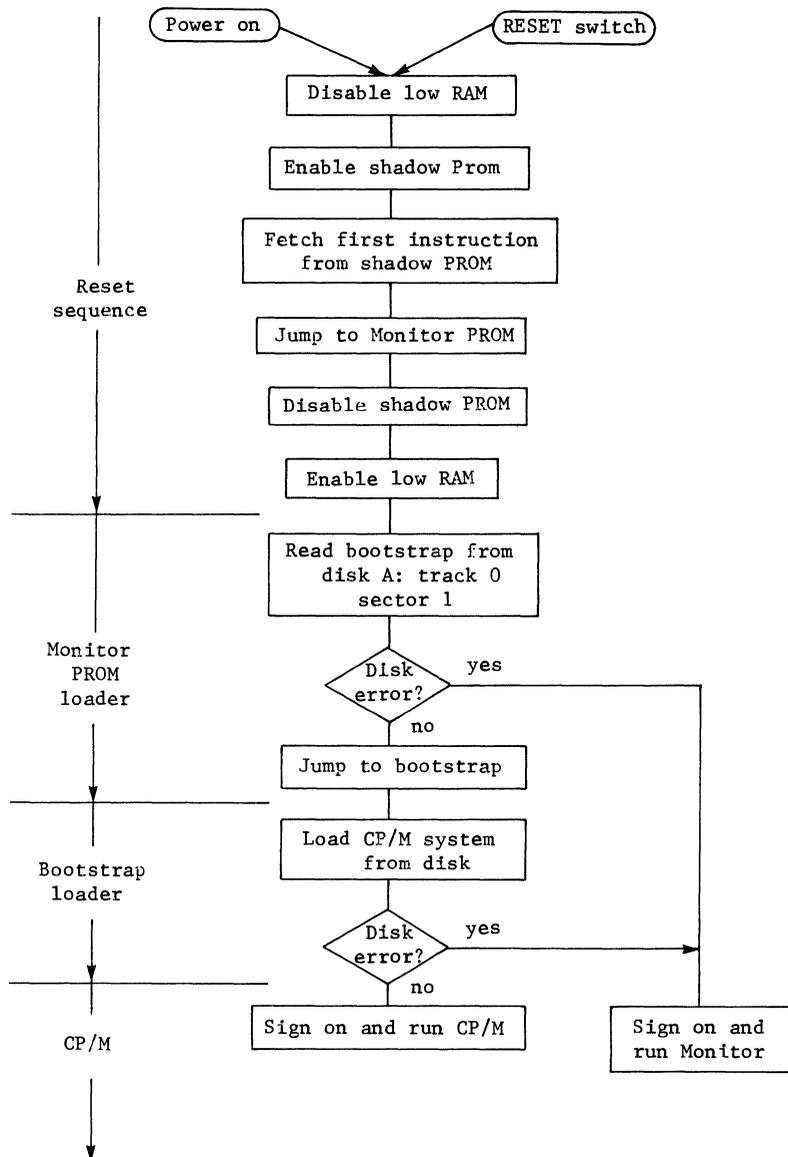
If your computer does have a complete monitor, it may come in handy in the future when we start writing assembly language programs that can, in case of programmer error, "bomb" the whole system, requiring us to hit that reset switch. The absence of a monitor in PROM will not slow us down, however, as the CP/M operating system includes DDT, the Dynamic Debugging Tool, that will provide these same functions.

Another feature provided by some monitor PROMs is a set of peripheral driver programs. These drivers are in the form of sub-routines that our programs can call, providing us with access to all of the system peripherals without having to know any details of their hardware addresses. Such drivers are often part of a software system known as an IOCS, or Input/Output Control System. Once again, an IOCS in PROM will not be necessary in a CP/M based computer, as CP/M will provide us with equivalent functions.

What is not optional in our monitor PROM is some form of loader program. Often the computer you will be using will assume that a CP/M system disk is in drive zero, and power up or reset will cause the operating system to be loaded and executed. In such a system, the functioning and even the existence of the bootup circuit and PROM become invisible. We simply push the button, and CP/M comes up running.

If we remembered to place the system disk in the drive. The right drive. Right side up.

The operating system

CP/M is, of course, the operating system in our computer. While this program was originally written on, and for, the Intel MDS-800 microprocessor development system, it has since been adapted to more computers of more different manufacturers than any other operating system. As we will be seeing in Chap. 3, this has been made possible by the ease with which CP/M can be adapted to differing hardware environments.

While other aspects of computer hardware have been stan-dardized to some degree or other, there has never been agreement on standard I/O port assignments. For instance, to transmit a char-acter from the computer to the console, a driver program must test the status of the output port to which the console is attached, to see

if it is ready to accept a character. If not, the driver must wait for a not busy signal. Once the port announces it is ready to accept a character, the driver outputs the character to the console output port.

The physical port address for console status and data will differ from one computer to another. The particular bit that is the busy bit within the byte read as the console status will differ from one machine to another. Its sense, whether one or zero for busy, will also vary from one computer to another. It is possible for a programmer to learn these hardware details for each computer he works with, and embed hardware-specific drivers in his programs. This practice has always been undesirable, as it restricts the use of a program to a particular computer. With CP/M it is not necessary.

Much more complicated operations are involved in writing to and reading from mass storage devices. As we mentioned before, CP/M can be called upon to keep track of all the details required in disk accesses, as well as operations through I/O ports.

Customizing CP/M

Since the computer user and/or programmer has been relieved of the necessity for knowing these details, the hardware specific interfacing has had to be done when the operating system was adapted to a particular hardware environment. But of course this system-to-hardware interfacing had to be done only one time. By one of us assembly language programmers, most likely.

The user-to-system conventions built into CP/M are one of the strong points of the operating system. All disk and I/O accesses are passed through a single entry point into CP/M. To implement this, function codes are passed in one register, and the data or buffer address passed in other registers. Using these conventions, it is possible to write programs that will run on any computer hardware without modification.

It is sad but true that some programmers still do not take advantage of these facilities provided by CP/M, and insist on using hardware specific addresses in their programs. As we will be seeing when we begin writing assembly language programs in Part IV, this is never necessary. Since we will refuse to repeat the errors of others, all of our programs will be completely portable.

Application programs

The firmware monitor will take some main memory address space, and the resident portion of CP/M (depending on version) will take up about 6K. There are also some special areas at the bottom of RAM that are used by the operating system. The rest of the main memory address space is available for user programs.

How much RAM is available to the user depends on how much is installed in the computer. While the 8080 family can address 64K, it is not often you find a system with the full 64K of RAM. In the programming we will be doing from here to the end of the book, the smallest possible CP/M system, residing in 16K RAM, will be sufficient.

CP/M loads and executes user programs in RAM in an area known as the "transient program area," or TPA. The TPA begins at a fixed address, and includes all available RAM not required by CP/M. In smaller systems, it may be necessary to overlay part of CP/M to gain enough user workspace. The operating system has been arranged so that this can be accomplished without interfering with the disk and I/O access portions of the operating system.

All of the non-system software (the user programs) are referred to as application programs. While we are in the process of editing, assembling, and debugging our application programs we will be using CP/M's editor (ED), assembler (ASM), loader (LOAD), and debugger (DDT). These programs are also going to be loaded into the TPA as we use them. Obviously, then, they will not reside in memory all at the same time, and only DDT will share main memory with our programs. DDT will have to be loaded along with our application programs only until the programs are fully operational.

Special memory areas

Down at the lowest addresses in our computer's RAM are locations dedicated to vectors. Vectors, in this sense, are unconditional jump instructions, like the one that got the CPU from its first instruction fetch at location zero to the monitor in PROM. The 8080 family uses eight memory locations as vectors for hardware interrupts. The Z80 and 8085 add other interrupt vectors. We do

FIGURE 2–2. A simplified memory map of a typical microcomputer running the CP/M operating system. Actual memory addresses are not shown as they will vary depending on memory available and the size and version of CP/M installed.


```
                65535 ┌──────────────┐
                      │              │ }
         PROM         │   MONITOR    │ }  1 - 2 K
                      │              │ }
                      ├ ─ ─ ─ ─ ─ ─ ─┤
                      │              │
                  ~   │   UNUSED     │  ~
                  ~   │   ADDRESS    │  ~
                      │   SPACE      │
                      ├ ─ ─ ─ ─ ─ ─ ─┤
                      │              │ }
                      │ MONITOR RAM  │ }  0 - 2 K
                      ├ ─ ─ ─ ─ ─ ─ ─┤ }
                      │              │ }
                      │    CP/M      │ }
                      │  OPERATING   │ }  6 - 8 K
                      │   SYSTEM     │ }
                      ├ ─ ─ ─ ─ ─ ─ ─┤ }
                      │              │
                      │              │
         RAM          │  TRANSIENT   │
                      │  PROGRAM     │    Total RAM
                      │    AREA      │    less 7 - 9 K
                      │   (TPA)      │
                      │              │
                      │              │
                      ├ ─ ─ ─ ─ ─ ─ ─┤
                      │              │
                      │  BUFFERS     │
                      ├ ─ ─ ─ ─ ─ ─ ─┤ }       } 256 Bytes
                      │              │ } 64
                      │  VECTORS     │ } Bytes
                  0   └──────────────┘ }
```

29
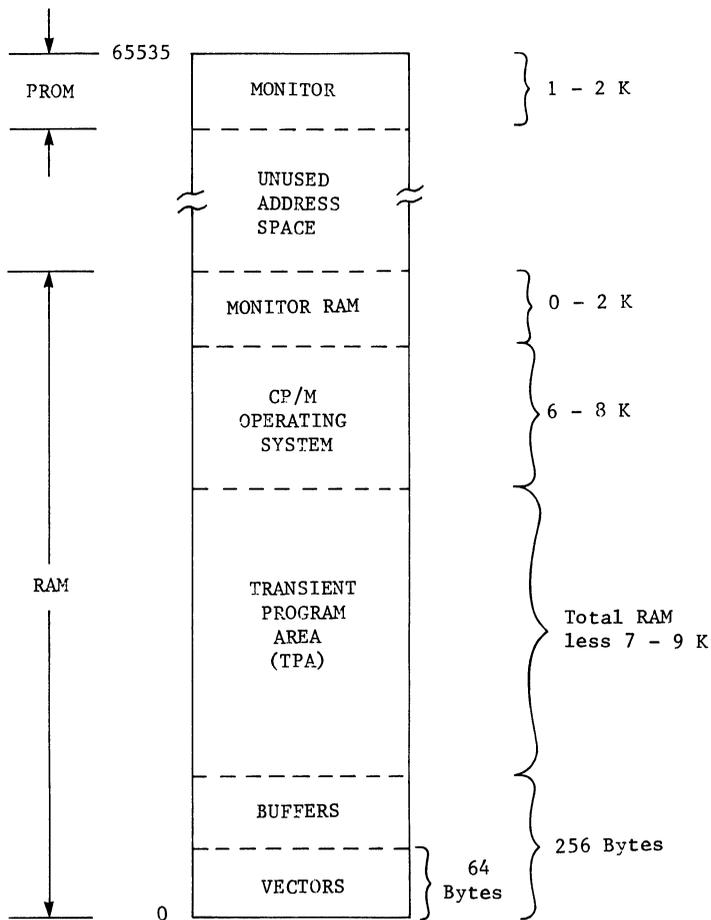
not need to be concerned with the details of these vectors at this time, so long as we keep in mind that our programs should not disturb these memory areas.

Above the space devoted to vectors, CP/M establishes buffer areas that we will be using when we interface our programs with the operating system. On our memory map in Fig. 2-2 we see that these areas all take up only 256 locations at the bottom of RAM, and the TPA begins at the next available location.

Another special area within RAM may be dedicated to monitor functions. This area will vary from computer to computer, and may not even be necessary in the machine you are using. Some monitors use only a few locations, others may grab several K of RAM for functions such as a memory mapped display image.

One of the responsibilities of the programmer who adapted CP/M to your particular computer was to insure that the operating system did not attempt to use any RAM space required by the monitor or other computer-specific functions. For this reason you will often see a machine running a 46K version of CP/M, for example, when 48K of RAM actually exists. The other 2K, it is safe to assume, was required for other functions.

In this chapter we have discussed three basic types of programs: the monitor, the operating system, and applications programs. The memory map shows how these software elements fit into memory in a CP/M based computer.

The CP/M-Based Computer

For the programming exercises in the remainder of this book, it is assumed that you have access to a minimum size microcomputer running some standard version (1.4, 2.0, 2.2) of the CP/M operating system, as in Fig. 1-1. While features of more complex systems will be discussed, only the devices shown will be required for the exercises to follow.

In this section you have acquired a vocabulary compatible with current usage in the microcomputer world. There are a myriad of other terms spoken by minicomputer users, and even more in the world of the large mainframe computers. Now it is time to reduce the size of even our minimum vocabulary, and start using logical and physical device names as defined by CP/M and its documentation.

Logical names and physical entities

Let us assume that you are sitting in front of your computer's operator's console video display terminal. Meet your "CRT:." Isn't

that easier than "computer's operator's. . .?" "CRT:" implies a physical device, in this case the tube with keyboard attached. Since this terminal . . . oops! Since this CRT: has been plugged into the appropriate port on your computer to serve as the operator's console, it has assumed the duties of logical device CON:.

Looking back at Fig. 1-1, we see only three other physical devices attached to our computer. These are the line printer, or LPT:, and two disk drives numbered 0 and 1 in the Intel MDS tradition. The creators of CP/M did not establish any three-letter-plus-colon designations for the disk drives, so we will just number the physical disk drives 0, 1, 2, etc. When we select a particular drive we are using its logical device name, A:, B:, etc. Disk drive logical names map one-to-one with physical names in our minimum system.

The same is not true for I/O devices in a CP/M system. As we see in Fig. 3-1, we have four logical I/O devices that can be accessed through the CP/M operating system. We have already discussed logical device CON:, and we know from the discussion in Chap. 1 that we will be using our LPT: to make listings of our programs. So LPT: is connected as logical device LST:, for "list device."

Similarly, our general purpose input and output devices are accessed through logical devices RDR: and PUN:. If we were using the old fashioned paper tape reader and punch, we would refer to them as PTR: and PTP:.

Selecting I/O devices

Under the heading "IOBYT Device Selectors" in Fig. 3-1 we see the schematic representation of four selector switches. If real switches were connected as shown, we could use them to switch from one I/O device to another. For instance, our RDR: could receive data from a card reader, or a paper tape reader, or the receive side (Rx) of a modem, depending on the setting of the RDR: switch.

As inherited from the MDS system, CP/M includes facilities for selecting any one of four physical devices for each logical device. Rather than the real selector switches, this is done through bit patterns stored in a one-byte memory location labeled IOBYT.

FIGURE 3–1. Logical to physical device mapping and selection in a CP/M computer. While the schematic representations of the device selectors indicate real switches, the switching is actually accomplished by selectively accessing different device driver subroutines within the operating system in response to bit patterns stored in the IOBYT.

CON:

CRT Terminal

IOBYT Device Selectors

Printing Terminal

Customized Basic Input Output System (CBIOS)

RDR:

Card Reader

Tape Reader

Rx

Modem

PUN:

Tx

Tape Punch

LST:

Line Printer

Basic Disk Operating System (BDOS)

A:

Disk Drive 0

B:

Disk Drive 1

CP/M

Logical Devices
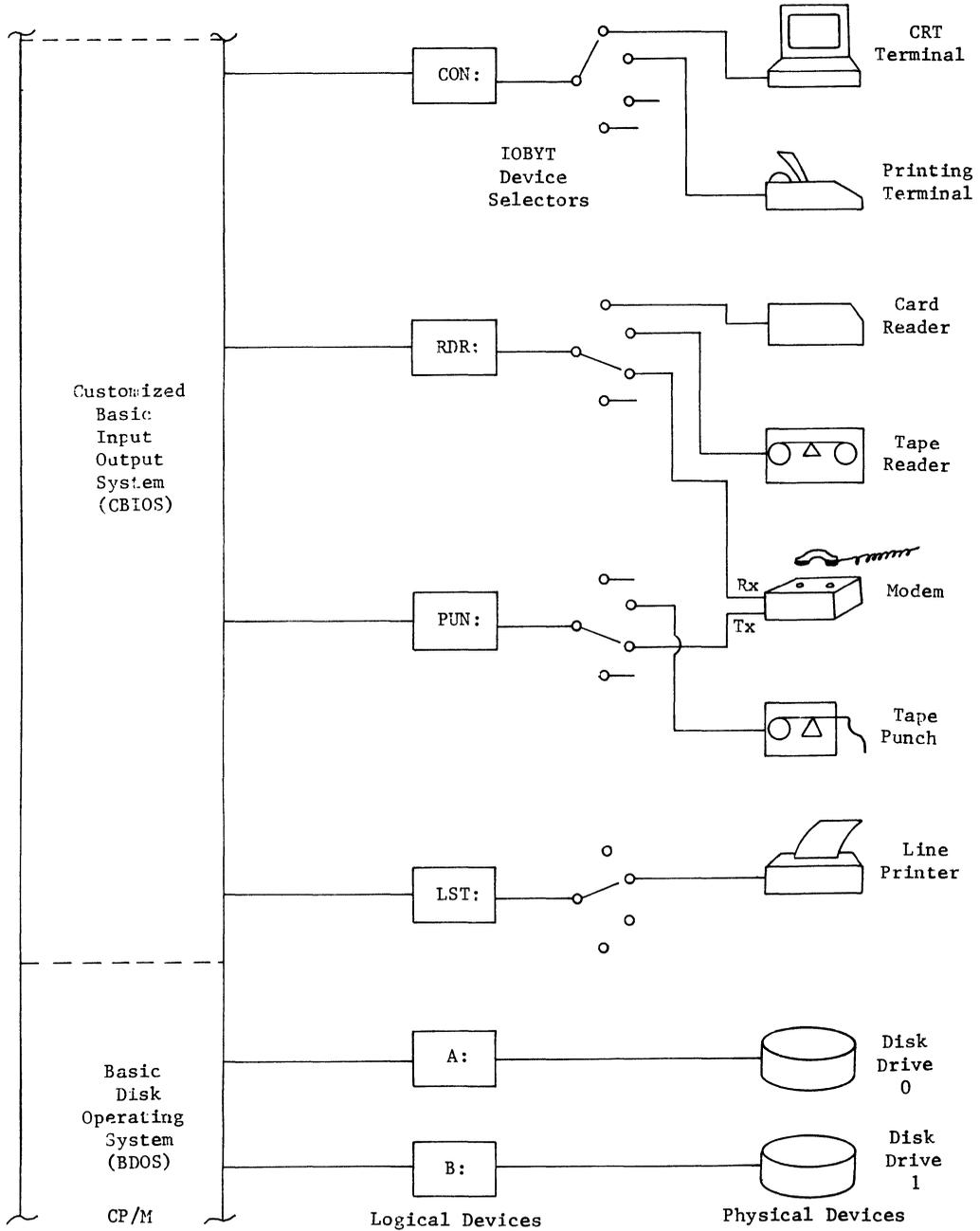
Physical Devices

Leaving the physical devices all permanently connected to the computer (if it had enough I/O ports!) and selecting them in software provides for more flexibility. Either the operator can make the selection through the CP/M CON: interface, or we could allow our programs to change the selections. Sometimes without telling us!

It is an unusual system that would have all sixteen selector inputs all tied to physical devices. Note that there is no requirement for devices to be connected to the first available input on each selector, so long as we know what device is attached for each IOBYT setting.

As you can see from this figure, a device such as the modem that includes both send and receive functions must have the proper settings in both the RDR: and PUN: sections of the IOBYT. The CON: is the only bi-directional logical device.

CP/M has facilities for changing device selections, either through the same entry vector used by our machine language routine in the Introduction, or by CON: operator action. We will be discussing more of the names and their derivation in Chap. 5.

In our minimum system, however, we will never need to change the IOBYT switches, and will only be using CRT: as CON:, and LPT: as LST:.

THE CP/M
OPERATING
SYSTEM
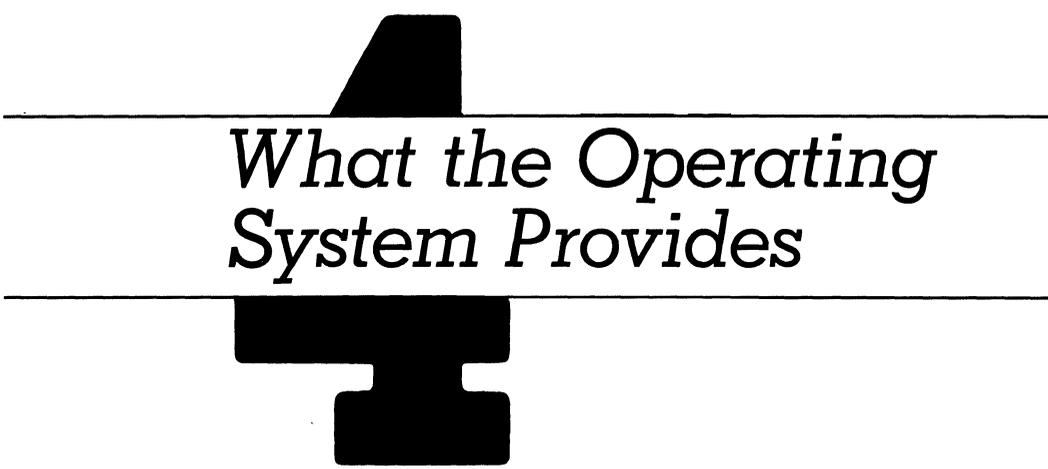
What the Operating System Provides

The services provided by the various hardware components of a computer system like that shown in Fig. 1-1 are pretty obvious by their very nature. The services provided by the various components of the software system are not so obvious, as we saw in the preceding section.

Since the bootstrap PROM program accesses the floppy disk system, as does CP/M itself, and since the computer may contain a monitor program that can access the console and possibly other peripherals, as does CP/M, it is obvious that there are more paths than one to these devices. Since we can't reach out and touch each part of the software system, or visually trace the interconnecting cables, it is not as easy to keep track of what is happening within a software system as it is in the hardware environment. Especially in a software system as complicated as the one we are working with.

In this section, we will be looking at the CP/M operating system as it appears to the operator and programmer. You will be exercising a few of the built-in and transient commands provided by CP/M, but there is no point in trying to exercise all of the

available commands in all their variations until you actually need to use them in Sects. IV and V. However, if you want, you may use a nice clean disk and experiment to your heart's content with creating, deleting, renaming, and copying disk files using the instructions which follow.

If you are sharing access to your CP/M based computer with other users, you should be careful not to use any of their disks to experiment upon. Otherwise, assembly language may not be the only new language you will learn.

Named file handling

A file in a computer is like one in a file cabinet. It can contain just about any sort of information, right or wrong, and the contents of a file can be identified by a label. In the cabinet, the labels on file folders can be of any reasonable length, and may or may not actually represent what is in the folder.

The name of a file in a CP/M system has a few constraints on it, in comparison. If the creator of the file is lazy, the name could become meaningless if it is not carefully chosen to remind him of what is in the file. Don't start off by naming all of your experimental programs "X.ASM" like some unfortunates have in the past!

CP/M allows each file name to be up to eight characters in length. A file type of three characters is appended to the name, following a period. In general form, this name/type is represented by FILENAME.TYP in this book. When the files on a disk are examined by using the DIR command, the period is not shown, and the name is always padded out to eight characters with spaces, to keep everything lined up. But you can use any number of characters up to eight in the name, and you don't have to type the spaces.

There are also some constraints on the file .TYP as well. Some types are fixed in meaning. ".ASM" is an assembly language source program. ".COM" is a command file that CP/M will load into memory and execute whenever you type in its FILENAME following the prompt ">" that says CP/M is ready to accept a command.

You can create your own file types so long as you don't conflict with the default types shown in Table 4-1. For instance, with CP/

M booted up from your system disk (see Intro.) in drive A:, place your nice clean disk in drive B: and enter the command:

SAVE 0 B:-WORK.001

followed by a carriage return (CR). This will create an empty file (zero blocks saved) with a name of -WORK and a "type" of 001, on the disk in drive B. What kind of type is this?

It really isn't any kind of type. You have used the .TYP field to number this disk as your first working disk. This empty file is like the label on the front of the file cabinet. If there are lots of

TABLE 4–1. Standard file types for CP/M disk files. The .TYP field of the file name should agree with established designations for these default types. Other designations for other types of files can be used at the computer operator's discretion.

	Standard Use *Defined by CP/M*	*.TYP*
Binary program image (Command)		.COM
Assembly language source program		.ASM
Assembler program output (hexadecimal)		.HEX
Assembler list output (print)		.PRN
Editor input—saved (backup)		.BAK
Temporary scratch file		.$$$
Submit command file		.SUB
	Other Common Usage	
BASIC language program		.BAS
BASIC compiled program (intermediate)		.INT
Fortran source		.FOR
Macro assembler source program		.MAC
Relocatable compiler output		.REL
Data file		.DAT

users on your system, you might have included your initials in the
name, like -WORK-JJ.001 for Jan Jones' first work disk. The lead-
ing dash (-) flags this filename as special; not a real information
containing file.

CP/M allows you to create just about any kind of file name and
file type, and this little exercise was to demonstrate that some
thought should go into the selection of both. And it is a good idea to
start off by naming each of your disks with this kind of empty file
that will create a directory entry to provide identification for each
disk you use.

In addition to being able to create disk files from the console,
CP/M provides your programs with the same power. Files can be
created or deleted, or accessed for reading or writing, either from
the console or from within a user's programs, using the same nam-
ing conventions in all cases.

Wildcards in file names

With your disks in the drives as above, enter the command DIR to
list the contents of the disk in drive A (a carriage return is assumed
following each command entered). "DIR" by itself lists all of the
files on the current disk. Now enter

DIR *.COM

and you will get a listing of only those directory entries with a file
type of .COM (for COMmand). The "*" is a wildcard that tells CP/
M to accept any FILENAME whatsoever. "*.COM" is equivalent
to "????????.COM" where each "?" means "accept any letter in
this character position." Any number of ?'s can be used in a file-
name in place of letters to help you search the disk directory for
sets of files with similar names.

Suppose you had been working on a program to play the game
of LIFE. In the process of updating the source program, you have
created a number of .ASM files: LIFE.ASM, LIFE-1.ASM,
LIFE-2.ASM, etc. You could find all of them by entering

DIR LIFE????.ASM

or, if you are lazy, you could have entered

DIR L*.ASM

which is a little more ambiguous, and might also have included LOAD.ASM or LOUSY.ASM if such existed on the disk.

LIFE.ASM is unambiguous. It tells CP/M that the file by that FILENAME.TYP is the only one you are interested in. You can specify files with more or less ambiguosity by including fewer letters and more ?'s, or go the whole route and use *.* and CP/M will accept any file name and any file type it finds in the disk directory.

Yes, there is a use for *.* in the real world. In the command

PIP B:=A:*.*

we have told the Peripheral Interchange Program (PIP) to copy all the files from drive A: onto the disk in drive B:. If you want, you can do just that right now, and create a copy of the system disk which is in drive A: onto your nice clean disk in drive B:.

In the course of discussing file names, file types, ambiguous and unambiguous names, and wildcards, we have seen that several CP/M utilities can be instructed to access files that have been specified using the same formats and wildcards. All of the programs accept all the wildcards and drive identifiers alike.

This is one of the most useful features of the operating system. The same formats and options for specifying files are accepted by all of the utility programs because all the programs use the same file handler routines in CP/M. And we will be seeing in Sect. V that your own programs can be just as flexible, using the same CP/M supplied file access routines. CP/M provides the same options in accessing named files for your programs as it does for itself. This is one of the features of the system that leads to painless programming.

Logical unit access

In Chap. 3 we discussed the mapping of logical to physical devices. CP/M provides both the console operator and user programs with

simplified access to logical units, and thus to the selected physical devices. From the console, the operator can specify a logical unit within a command string such as

PIP PUN:=FILENAME.TYP

and have the named disk file sent to the physical device currently attached as logical device PUN:. In this way the computer operator could send a program source file to another computer using a modem and a telephone connection.

Similarly, a program can access any of the logical devices as well, without knowing what physical device may be connected. This could allow a general purpose data communications program to be used, with the operator specifying the physical to logical assignment, for example. And finally, either the operator or the user program can change the logical device assignments.

The mechanisms for making these accesses will be discussed in Chap. 6, and again in detail in later chapters as actual applications for the techniques are programmed. While it may sound complicated to you at this point, you will be seeing that the careful design of the operating system has simplified all these file and device accesses, and the programmer is relieved of the tasks of keeping track of physical devices and the locations of data on the disks. CP/M is a nice place to work.

Line editing

Since you have been keying in command lines like those listed above, and will probably want to experiment with others as soon as you are finished reading this chapter, it is time to discuss CP/M's built-in line editing feature.

If you are keying in a command line in response to the CP/M prompt ">" and make a single-keystroke error, you can back up one character by hitting the DEL, DELETE, BS, or RUBOUT key on your terminal. These four options are shown here as different keyboards have a different designation for this key. You may have to experiment a little with your terminal; some have two of the key names listed above but only one will work properly.

When CP/M sees the delete key code, it will not always be

able to back up the cursor on the screen, depending on version. If your version does not back up and over-write the character, the deletion is shown by the repetition of the character. Since this clutters up the typed line with extra characters, you can review the command line before terminating it with the usual carriage return by entering the control code CTRL R.

A control code is entered by holding down the CTRL key while pressing and releasing the letter key specified. Then the CTRL key is released. If you have made a number of keystroke errors on a command line, and rubbed out the bad ones and re-typed the correct characters, the line as displayed on the CRT may be indecipherable. To review it before executing it, type CTRL R and it will be repeated as edited on the next line.

To give up, and abort the entire entry, type CTRL U or CTRL X. If you have made too many errors on one line, this is often the best way out. Give up! CTRL X! Retype the whole line and be sure it is right.

These are about the only line editing controls you will ever need, although there are others listed in the CP/M manuals. Don't try to learn more than you need to start with.

As with so many other good features of CP/M, these line editing controls are available to you the programmer, as well as to you the operator. Your own programs can, and always should, make use of this feature to provide a friendly environment for the computer operator. Remember, he will often be you.

And again, to avoid overloading your brain at this point, the details of how to easily include all these nice CP/M features will be given in later chapters, as the time comes for you to write programs that make use of them. Right now you might want to play around with the commands discussed this far, and exercise the line editing features. Then again, you might not want to. Do it anyway.
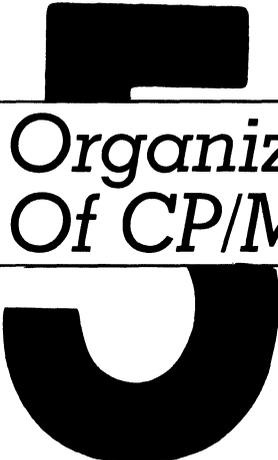
Organization
Of CP/M

We are all working together to one end,
some with knowledge and design,
and others without knowing what they do.

  Marcus Aurelius Antoninus

Before we can look at the organization of CP/M and see how it
provides all the services listed in the previous chapter, we have to
take a quick look at some items that the operating system needs.
These needs include access to programs at the most primitive
level, that communicate with the physical devices through the
computer hardware. These driver routines have to be supplied
before CP/M can run, and are at the lowest level of all the software
in your computer system.

   Eventually you will want to learn more details of this level so
that you can make additions to existing CP/M systems or adapt the
operating system to a new computer. These are a couple of the
interesting tasks that can be performed by the assembly language
programmer.

44

Disk and I/O access primitives

In the first section we took a look at how data is stored on a disk, and how a loader program somewhere in ROM is used to load the operating system into the proper place in RAM. This loader is not part of the operating system. It has to pre-exist somewhere in the computer's memory so that we can get the operating system off the disk and into memory.

This loader must be available when the power is first turned on, and also when the operator hits the RESET switch. In addition, this same loader, or a portion of it, will be used to "warm start" the operating system. A warm start assumes that the system has been running previously, so it will use the currently selected disk drive, and will make no changes in the logical to physical device mapping as defined by the contents of the IOBYT. A warm start is used at the end of transient programs to reload CP/M, or in response to the operator pressing CTRL C to abort a program in case of trouble.

Obviously, the PROM based loader must contain routines that permit accessing the disk at the most primitive level, in order to position the head of the correct disk drive to the correct track and sector where the beginning of the operating system will be found, and then load the system into RAM. These same routines will later be used by the operating system to perform the same primitive functions. One of the tasks involved in adapting CP/M to a particular computer is to connect the proper functions in CP/M to the proper disk access primitives in PROM.

The same is true for I/O device accesses. CP/M does not know the absolute addresses of all the I/O ports, so a computer specific set of I/O device driver routines is required. Since no communication with the CON: or other devices is required before the system is loaded into RAM, it is customary to include the complete I/O device access primitives within the proper area set aside in CP/M, in the Customized Basic Input/Output System (CBIOS). These driver routines will then be loaded from the disk along with CP/M. But this is not always necessary, and I/O device drivers in PROM could be used by CP/M.

In either case, a particular area within the operating system has been set aside for a number of vectors that point to the proper disk and I/O access primitives. These vectors are shown in Table 5-1.

FIGURE 5-1. The organization of the software elements within user programs, the CP/M system, and the hardware-specific support routines in a typical microcomputer. User programs access all devices through the BDOS call vector, and exit back to CP/M through the warm start upon completion. Power-up, reset, and device driver functions are provided by primitive routines stored in read-only memory (ROM) within the computer. CP/M accesses these functions through the CBIOS vectors.

```
        ┌──────────────┐         ┌──────────────┐
──────► │   JMP BOOT   │ ──────► │     BOOT     │ ──────►  TO CCP
        │              │         │              │
──────► │   JMP WBOOT  │ ──────► │     WBOOT    │ ──────►  TO CCP
        │              │         └──────────────┘
        │              │           CBIOS START
        │              │           UP ROUTINES
        │              │
──────► │   JMP CONST  │ ──────► │    CONST     │
        │              │         │              │
──────► │   JMP CONIN  │ ──────► │    CONIN     │
        │              │         │              │
──────► │   JMP CONOU  │ ──────► │    CONOU     │
        │              │         │              │
──────► │   JMP LIST   │ ──────► │    LIST      │
        │              │         │              │
──────► │   JMP PUNCH  │ ──────► │    PUNCH     │
        │              │         │              │
──────► │   JMP READER │ ──────► │    READER    │
        │              │         └──────────────┘
        │              │          CBIOS I/O
        │              │          DRIVERS
        │              │          (may be part
        │              │            of PROM)
        │              │
──────► │   JMP HOME   │ ──────► │    HOME      │
        │              │         │              │
──────► │   JMP SELDSK │ ──────► │    SELDSK    │
        │              │         │              │
──────► │   JMP SETTRK │ ──────► │    SETTRK    │
        │              │         │              │
──────► │   JMP SETSEC │ ──────► │    SETSEC    │
        │              │         │              │
──────► │   JMP SETDMA │ ──────► │    SETDMA    │
        │              │         │              │
──────► │   JMP READ   │ ──────► │    READ      │
        │              │         │              │
──────► │   JMP WRITE  │ ──────► │    WRITE     │
        └──────────────┘         └──────────────┘
           CBIOS                  DISK ACCESS       DISK DRIVES
           VECTORS                PRIMITIVES
                                  (part of
                                     PROM)
```

I/O DEVICES

47

TABLE 5–1. The Customized Basic Input/Output System (CBIOS) within CP/M includes a set of standardized interfacing vectors, that will remain in these relative locations for any version of CP/M installed in any computer. This permits all customization to be restricted to the subroutines accessed through these vectors.

ADDRS	Vector	Description/Functions
		System Load Functions
3E00H+b	BOOT	Enter after power on or RESET and after system has been loaded from disk. Display sign-on message, zero IOBYT and DRIVE, set up low RAM vectors, select current drive, go to CCP.
3E03H+b	WBOOT	Enter after CTRL C or JMP 0. Load system from disk. Set up low RAM vectors, select current drive, go to CCP.
		I/O Device Drivers
3E06H+b	CONST	Test CON: for keyboard character ready.
3E09H+b	CONIN	Wait for and read CON: keyboard character.
3E0CH+b	CONOU	Send one character to CON: display.
3E0FH+b	LIST	Send one character to LST: device.
3E12H+b	PUNCH	Send one character to PUN: device.
3E15H+b	READER	Wait for and input one RDR: character.
		Disk Access Primitives
3E18H+b	HOME	Set current drive head to track 0.
3E1BH+b	SELDSK	Select drive, store number in DRIVE.
3E1EH+b	SETTRK	Set current drive head to track specified.
3E21H+b	SETSEC	Seek current drive to sector specified.
3E24H+b	SETDMA	Set RAM buffer start address for next disk read or write.
3E27H+b	READ	Read selected disk, track, sector into RAM buffer.
3E2AH+b	WRITE	Write contents of RAM buffer into selected disk, track, sector.

b = BIAS = 400H for each 1K offset above 16K CP/M

Each vector is a three byte jump instruction, JMP BOOT, JMP WBOOT, etc. The addresses shown in the table are the absolute addresses for a 16K version of CP/M. The first vector points to routines that get things running properly after the PROM loader has bootstrapped in the CP/M system, and will be jumped to from the PROM loader.

The BOOT routines will then display the "xx K CP/M . . ." sign-on message, zero the IOBYT and DRIVE select bytes in low RAM, and set up the vectors at locations 0 and 5. BOOT then jumps to the CCP.

The next vector points to WBOOT, for warm starts. WBOOT is entered when the operator or a transient program wants the system reloaded. Within the WBOOT routines the loader in PROM will be called upon to reload the operating system. Then WBOOT will rewrite the low RAM vectors, but will leave the DRIVE and IOBYT selections as they were. WBOOT then jumps to the CCP.

CP/M itself is activated when the Console Command Processor (CCP) is entered. It is CCP that prompts with the ">" character and then waits for a command to be entered from the CON: device.

CCP, other portions of CP/M, and user programs in the TPA will all communicate with I/O devices through the next six vectors shown in the table and Fig. 5-1. These vectors point to the driver subroutines that do the decoding of IOBYT and perform the actual communications between software and physical devices.

The seven disk access vectors follow in the table. You can see from the descriptions of the routines pointed to by the vectors that everything that can be done to a disk, at least from a program, can be accomplished through these vectors and the primitive routines that they access. An operator is still required to put the right disk in the right drive, right side up. Humans can't all be replaced by machines.

All that is required to adapt CP/M to a new computer is to provide the loader in PROM and the 15 routines to be accessed through these vectors. Since this has already been done on your computer you don't need to understand the details of these programs to use CP/M. But someday you may be involved in customizing these programs to add new I/O devices or adapt CP/M to a new computer. The Digital Research CP/M manuals give all the details you will need, and include sample programs.

BDOS—The Basic Disk Operating System

In the beginning a floppy disk sector held 128 bytes of data, plus address and checksum information. Therefore the basic element of disk storage is the 128 byte record. Also in the good old days, assembly language programmers thought in terms of 256 byte "memory pages," which in nice round hexadecimal numbers hold 100H bytes. In CP/M documentation, you will run across references to 128 byte sectors and records, 256 byte pages or "blocks," and 1K byte "groups."

Double density and hard disks have physical sectors that can be multiples of 128 bytes, so we will use terms that are independent of the size of a sector on disk, to avoid confusion. All you have to remember is that

    1 record = 128 bytes
    1 block = 256 bytes = 2 records
    1 group = 8 records = 1K bytes.

Actually, you can usually let the system remember all of that. Since CP/M is structured to make things easy for the user, all the user needs to do is tell the system to read FILENAME.TYP into the TPA. BDOS will handle the details.

The first detail will be to search the directory for the named file. The name will be found in a directory entry that is the image on disk of a File Control Block (FCB). Other information contained in the FCB tells the system where to find the file on the disk, and how big it is. The minimum increment of space on the disk that can be allocated by CP/M is not one record, but is the 1K byte group. If a file contains only a single byte, it still takes up 1K bytes on the disk. If it contains 1K + 1 bytes, on up to 2048 bytes, it will take up two groups (2K) of disk space.

This seeming inefficiency is the price you pay for a system that provides all the high level niceties that CP/M does. Between the user with a file name, and the disk access primitives listed in Table 5-1, stands BDOS to take care of all the little details.

Details like remembering what files have been erased, so that their space on the disk can be reused. This is known as dynamic disk space allocation, and is what keeps you from running off the end of the disk all the time. BDOS maintains all the information

required for this in the FCBs in the directory. User programs can ask BDOS to look for files in the directory, read them into RAM, write them from RAM to disk, or erase them.

In addition, the computer operator, working through PIP or the CCP, can ask BDOS for these same services. And whether the disk file access request comes from a user program or from the user sitting at the console, BDOS will display all disk access errors on the console.

In the next chapter we will be looking at how a user program requests disk accesses from BDOS, and in Sect. V will be doing just that. There is not much else you need to know about BDOS, except that when it says

BDOS ERR ON B: BAD SECTOR

don't panic. Maybe you just forgot to put the disk in the drive. Of course it is possible that there was a real disk error in drive B. BDOS will try over and over to read or write a sector whenever it encounters a checksum error. But its patience is limited, and after a few retrys it will give up and display the message above.

The computer operator has two options at this point. If he types a carriage return on the console, the error will be ignored. If he types CTRL C, the system will be rebooted (warm start). You can ignore read errors in text files, like assembler source files, and recover the rest of the file. But don't ever ignore read errors when loading a program from a .COM file. This is the binary image of the program, and any error is usually fatal.

If you ever see the BDOS error message

BDOS ERR ON R: SELECT

or this message with any other illegal drive specified, it means your program is totally lost, and has garbaged the DRIVE select byte at location 4 in RAM. This means that it has probably garbaged lots of other locations as well. NOW IS THE TIME TO HIT THE RE-SET BUTTON. You know you always wanted to!

A BDOS error of READ ONLY means that the disk you are trying to write on is write protected by having its notch covered (5½″ disks) or uncovered (8″ disks) or that you changed the disk without letting CP/M know. If you have changed the disk, or if you

pull it out and remove (replace) the sticker on the notch, hit CTRL C and BDOS will reread the disk directories and then be able to write again.

All of these error messages are displayed on the console whether the source of the error is a program fault or human error. When running a higher level language program, like Star Wars written in BASIC, there is usually nothing you can do to recover from BDOS errors. If the error is real and permanent, indicating a defective floppy disk, recover as much data as you can from the rest of the disk and then throw the disk away. Disks are cheap. Don't risk your valuable software on a bad one. You did make a backup copy, didn't you?

CBIOS—The Customized Basic Input/Output System

CP/M is an extremely well organized operating system. It is compact and easy to adapt to a new hardware environment. After struggling through the first part of this chapter, you might not be in total agreement that it is an easy system to learn.

One reason that there seems to be a lot of jumping around from PROM to CBIOS to CCP to TPA is that the system was written to run on a minimum computer, with only 16K bytes of RAM required for version 1.4 of CP/M. To provide the user with all of the required services and still allow him 10K of user workspace in the TPA required that CP/M make maximum use of the available resources of the computer. The seemingly fragmented organization of the system, as we see in Fig. 5-1, is actually evidence of its efficient organization.

It is also evidence of the features of CP/M that make it so adaptable. Access to all disks and I/O devices can be made through the single location 5 vector pointing to the BDOS entry point. All other required vectors, the 15 CBIOS vectors, are grouped in one place, and more than enough memory space is available immediately above them for the incorporation of the customized drivers that make CP/M run on your particular computer hardware.

Table 5-2 is the memory map of version 1.4 of CP/M installed in a minimum 16K system. (The PROM is not shown, as it is part of the hardware and its address and size will vary from computer to computer.) The map shows the locations of all the vectors, RAM

variables, and the parts of the operating system. BIAS is a value that will be added to the absolute addresses shown for versions of CP/M larger than 16K. The program can be resized from 16K to 64K in 1K increments, to adapt it to any 8080 family of microcomputer

All the other adapting required has been done in your system in the CBIOS. In various articles, books, and manuals on CP/M you will find "BIOS" and "CBIOS" used interchangeably. Since some customizing is required for every computer CBIOS should be the word to use.

TABLE 5–2. The memory map for version 1.4 of CP/M. All memory usage below address 100 in hexadecimal remains the same regardless of the installed size of the operating system. The addresses for the moveable portion of CP/M are shown for a 16K version. A bias value will be added to these addresses depending on installed size.

Memory From	Address To	Contents	Function
0000H	0002H	JMP WBOOT at 3E00H+b	Warm start vector
0003H		IOBYT	I/O selector
0004H		DISK	Disk selector
0005H	0007H	JMP BDOS at 3106H+b	BDOS entry vector
0008H	0037H	not used by CP/M	Interrupt vectors
0038H	003AH	JMP DDT	DDT breakpoint
003BH	005BH	not used by CP/M	
005CH	007FH	TFCB	Transient FCB
0080H	00FFH	TBUFF	Default RAM buffer
0100H	28FFH+b	TPA	Transient programs
2900H+b	30FFH+b	CCP	Console commands
3100H+b	3DFFH+b	BDOS	Disk operations
3E00H+b	3E2CH+b	CBIOS vectors	See Table 5–1
3E00H+b	3FFFH+b	CBIOS	I/O operations

b = BIAS = 400H for each 1K offset above 16K CP/M

Inside CBIOS are all the I/O drivers shown in Fig. 5-1. Not repeated in this schematic diagram are the IOBYT selectors shown in Fig. 3-1, but each of the drivers will have to decode its part of IOBYT if this feature is enabled in your computer. This is another option with CP/M; the IOBYT does not have to be used if multiple I/O devices are not installed.

If IOBYT is implemented, it takes the format shown in Fig. 5-2. Each of the four logical devices uses a two-bit field within

FIGURE 5–2. Subfields within the eight-bit IOBYT specify one of four physical devices to be accessed by each of four logical devices in a CP/M based computer. Minimum systems with few peripheral devices do not need to implement the IOBYT. More complicated systems can use it to simplify operator selection of input/output devices. The device names shown were inherited from the Intel MDS-800 development system that spawned CP/M.

```
IOBYT     L S T   P U N   R D R   C O N
  Bit   +--+--+--+--+--+--+--+--+
         7   6   5   4   3   2   1   0

                                0   0 = TTY: ⎞
                                0   1 = CRT: ⎟ Console
                                1   0 = BAT: ⎟
                                1   1 = UC1: ⎠

                        0   0 = TTY: ⎞
                        0   1 = PTR: ⎟ Reader
                        1   0 = UR1: ⎟
                        1   1 = UR2: ⎠

                0   0 = TTY: ⎞
                0   1 = PTP: ⎟ Punch
                1   0 = UP1: ⎟
                1   1 = UP2: ⎠

        0   0 = TTY: ⎞
        0   1 = CRT: ⎟ List
        1   0 = LPT: ⎟
        1   1 = UL1: ⎠
```
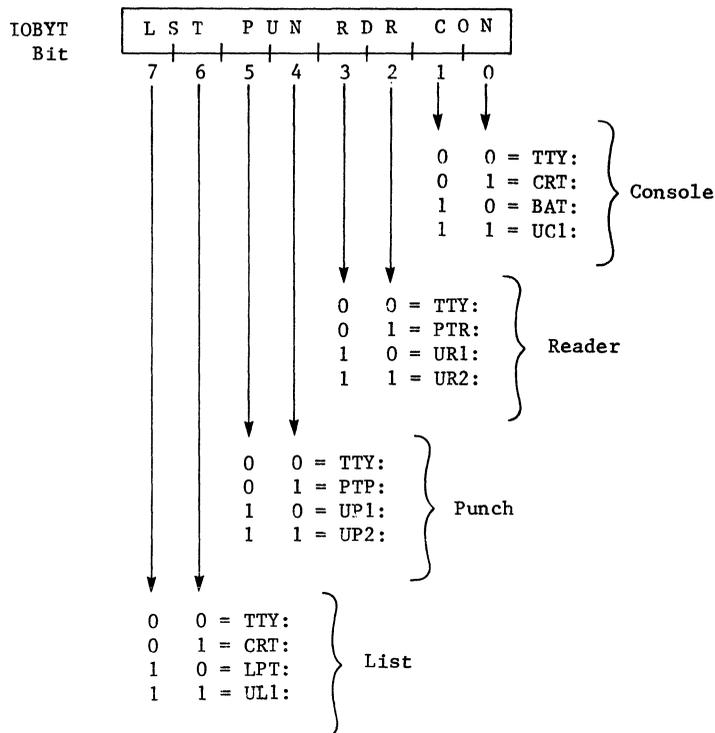
IOBYT to select one of four physical devices. Each logical device driver (CONIN, PUNCH, etc.) will have to read the current IOBYT, mask out its two bits, and decode the bits to see which device to communicate with.

All of this is straightforward, and any competent assembly language programmer can write customized drivers using IOBYT. What is not so easy to get used to is all the funny device names that go along with use of the IOBYT. With your CP/M system disk in the current drive, enter the CCP command

STAT VAL:

and you should see the following display:

CON: = TTY: CRT: BAT: UC1:
RDR: = TTY: PTR: UR1: UR2:
PUN: = TTY: PTP: UP1: UP2:
LST: = TTY: CRT: LPT: UL1:

produced by the CP/M transient utility program STAT.

Back in Chap. 3 we saw that all kinds of goodies could be hung onto our microcomputer, and selectively accessed for data transmission and reception. One handy device is the modem. If you want to select the modem to be attached as the PUN: and RDR:, how do you go about it?

In the list of devices shown by STAT VAL: we don't find any physical device name like MOD: for modem. This is another holdover from the days when CP/M was created. It was originally programmed on an Intel Microcomputer Development System, and all the logical and physical device names shown above are part of the "MDS Syndrome."

We already know that the MDS expected that old mechanical device, the ASR-33 Teletype, to be the default selection for all four logical devices. Other physical device names handed down as part of the syndrome include PTP: and PTR: for paper tape punch and reader. If a user insists on connecting more modern devices, he has to refer to them as the User Punch 1 or 2, and User Reader 1 or 2. UL1: would be the User selected List device. At least Line PrinTer and CRT: are included in the syndrome. But no MOD:.

Obviously, you would connect the modem to any available

serial I/O port, and write a CBIOS driver for PUN: and RDR: that would talk to the modem when UP1: and UR1: were assigned to the punch and reader logical devices. Or your CBIOS driver could access the modem as TTY:, if it was your number one peripheral.

In any case, a customized driver has to be incorporated in CBIOS, and the operator has to remember which funny name refers to the modem. Until, that is, you learn enough about assembly language programming to be able to customize the names of the devices in STAT. It can be done, but first you have to learn more about the system, and then how to write assembly language programs. That is a carrot dangling in front of you, in case you didn't notice.

By the way, just what is a BAT:? On the MDS it was the paper tape reader loaded up with a tape full of pre-punched console commands. In the BATch mode, the paper tape was read one command at a time, as though it was an operator issuing commands to the system. Each command was then executed in turn, then the next read from the PTR:. This allowed the operator time to go get a cup of coffee, or go to lunch, or just mess around a little. The computer could run batches of jobs unattended, getting all its commands from the tape.

CP/M doesn't use this batch mode. It has an even smarter but similar program known as SUBMIT. It reads commands from a disk file, and does all kinds of smart things. Now we have to figure out something else for BAT: to refer to. I'm sure you'll think of something, even if you don't have a mother-in-law.

CCP—The Console Command Processor

After the CP/M system has been loaded into RAM from the disk, CCP prompts the operator for a command line input by displaying on the console the currently selected drive designation followed by the "greater than" symbol: A>.

CCP expects to see a command consisting of the name of one of the resident functions, or the FILENAME of a .COM file on disk. In the latter case, the FILENAME can be preceded by a drive designator like "B:" if the .COM file is not on the current disk. If the resident function or .COM program requires options to be specified, they follow the command name on the same line, separated by spaces. Options can be other file names, ambiguous

or unambiguous, with or without drive designators. Options can also be any other information required. The line editing features discussed earlier are active while the operator is keying in the command line.

CCP executes resident commands using the options specified, and then prompts for another command line input. If the command is not one of the resident commands, CCP assumes it is the name of a .COM file on disk. In that case, CCP aids the author of that transient program by setting up a default file control block, containing the properly formatted name(s) of any files specified in the command line, and a RAM buffer containing the entire text of the command line, past the command name itself.

Suppose we have a program named COMPARE that compares the contents of two disk files and displays any differences found between them. To keep the display from scrolling off the screen faster than the operator can read it, the operator is given the option of entering a directive to tell the program to pause following the display of each miscompare. The operator invokes the program by typing

COMPARE B:TEST.ASM B:TEST.BAK PAUSE

to see all the updates made to his program TEST following the last edit session (TEST.BAK is the next to last version of TEST.ASM, automatically saved by the CP/M editor).

Here the programmer who wrote COMPARE uses the CCP generated default file control block (TFCB in Table 5-2) to find the two filenames that the program requires. Since the operator can also specify another option, COMPARE will look in the command line buffer (TBUFF in Table 5-2) to find the option.

CCP has saved the programmer a lot of effort by setting up these two storage areas before loading and executing the transient program. Another example of super service for the CP/M assembly language programmer.

Resident functions

Given unlimited storage, all of the utility routines listed in Table 5-3 could reside permanently in the system area in the computer memory. Then they could be executed instantaneously, rather

TABLE 5–3. The utility functions provided by the CP/M operating system are divided into two classes: those resident in memory at all times, and those loaded into the transient program area. These latter progams take up memory space only until their functions are completed.

Resident Commands

ERA FILENAME.TYP (afn)	ERAse file(s)
DIR	Display disk DIRectory
DIR FILENAME.TYP (afn)	Display DIRectory file(s)
REN FILENAME.TYP=FILENAME.TYP	REName a file
SAVE xx FILENAME.TYP	SAVE contents of TPA on disk
TYPE FILENAME.TYP	Display contents of a file

Transient Commands

STAT	Display STATus of current disk
STAT FILENAME.TYP (afn)	Display STATus of file(s)
STAT VAL:	Display logical/physical I/O
STAT DEV:	Display I/O assignments
ED FILENAME.TYP	EDit an ASCII file
ASM FILENAME.shp	ASseMble a program from file
LOAD FILENAME	LOAD .HEX file to .COM file
DUMP FILENAME.TYP	Display file in memory DUMP format
SUBMIT FILENAME x,y,z	SUBMIT batch processing
MOVCPM yy w	Generate re-sized system
SYSGEN	Write moved system to disk

(afn) = ambiguous file name(s) permitted
xx = size of file in 256 byte blocks
shp = disk drive for source, hex, and print files
x,y,z = optional parameters
yy = size of resulting CP/M system
w = * option

than being loaded into RAM from disk as they are required. The alternative would be for all of the utilities to be .COM files on disk, with none resident in the operating system.

Since either method would work, the division between resident and transient utilities is strictly the result of a judgment decision on the part of the designers of CP/M. Since resident functions are not loaded from disk, they execute rapidly, but take up memory space.

The resident function you will be using most often is DIR, used to display your files on disk. DIR followed by a carriage return will show all your files on the current disk. DIR followed by a drive designation will list the contents of the disk directory in that drive.

DIR can also be followed by a filename, ambiguous or unambiguous, as discussed in Chap. 4, to verify the presence of a particular file or group of files.

ERA can also be invoked with the same options that work for DIR. ERA *.BAK will clear your disk of all backup files.

ERA B:*.BAK

will do the same for all the backup files on drive B:. ERA must be used with caution, since it causes a file to be erased, and the next disk write will reuse the disk space made available by ERA. "ERA" stands for "erase," of course, and has nothing to do with political activities.

In Chap. 4 we used SAVE to create an empty file, just to put a disk name in the directory. SAVE will also create a .COM file by moving the contents of the TPA onto disk, with the name specified.

SAVE 12 TEST.COM

will create a file containing 12 memory pages, or blocks of 256 bytes each. Obviously a topic for later discussion, when you are actually writing transient programs.

REN allows you to rename a file, if you remember that in assembly language programming a curious convention has been handed down over the years. That convention is the practice of specifying everything backwards, as in MVI C,WCONF, where the C register is the destination, and WCONF is the source of a

value to be moved into C. REN wants the same reverse sequence:

REN B:GOODPROG.COM=B:TEST.COM

and is here used to put the permanent name onto a transient program that had been called TEST until it was fully debugged. The programmer had better remember to rename TEST.ASM now, too!

SAVE and REN don't get much use, normally. The final resident command, TYPE, makes up for that. Like SAVE and REN, TYPE requires an unambiguous file name as an option. It will type out the contents of the named file on the console. TYPE can also be used to list files on the line printer, by including the special control CTRL P in the command line.

CTRL P is a toggle. Enter it once and everything that is output to the console for display will be echoed to the LST: device as well. Hit CTRL P again and the LST: output will stop. Since everything displayed on the console will get printed when you use this control, you won't get nice formatted printouts like those supplied by the PIP utility. But CTRL P is a handy way to get quick program listings in conjunction with TYPE, and can be used with DIR and STAT to list the contents of your disks.

Another toggle is CTRL S. If you TYPE a source program listing, it will go scrolling past on the console so fast you won't be able to read it all. Enter CTRL S once and the display will stop. Again and it will resume flashing by. It is a shame that this control is not on a single key. That would make it easier to use this handy function.

When you become a proficient programmer you can write your own customized version of BIOS, and you can implement a similar pause control using a single key. Or better yet, you could implement a speed control to slow the display down for listing long files. Power to the programmer!

Transient utilities

Enter the command STAT *.COM and you will see why STAT is not a resident function. The display shows the statistics of all the .COM files on your disk, including STAT.COM itself. Look at

the BYTS reported for STAT and you see why it is on disk and not part of the system; it is too big.

STAT is used most often to allow you to check how much space is left on a disk. If you are about to update TEST.ASM you will want to know how big it is as well, so you can be sure that there will be space enough on the disk for the new TEST.ASM as well as TEST.BAK when you get through with your update editing session.

STAT VAL: was used earlier to show all the possible I/O devices, and STAT DEV: will show the current assignments, as programmed into the IOBYT. STAT is also used to change those assignments in systems using the IOBYT. For instance,

STAT LST:=LPT:

will set the list device part of IOBYT to binary 10 to signal CBIOS that LPT: is the physical device to assign as the logical list device. Well, you want to list on the line printer, don't you?

Since STAT can change the assignment for logical device CON: as well, and since CON: is the source of the command and the destination for the next CCP prompt, assigning another device to CON: will cause your current console device to go dead as soon as STAT makes the assignment. What to do if you try this when there is no other console device plugged into the computer? Hit the RESET switch. But remember that doing so will zero the IOBYT and reset all the device assignments to TTY:. You will regain use of the original console, but lose any other reassignments you might have made.

The other transient utilities supplied with CP/M will be discussed as they are encountered in our learning-by-doing sessions beginning in Part IV. If you must fill your head with details you won't be using yet, you can skim through their descriptions in the CP/M manuals. Since those manuals are the complete reference works on the system, everything about all of the utility programs is included in their discussions.

There is no need to try to learn everything about all of these programs at this point. You will be introduced to as much information about them as you need at each step in the learning process. But if questions should arise, remember that this book does not replace the Digital Research manuals. Turn to them if you feel the need.

User programs

Since the transient utilities are .COM files and are loaded into the TPA for execution, and since we, the users of the CP/M based computer, will be writing programs that also execute in the TPA, it is obvious that there is no real difference in form between a "CCP transient command" and a user program. The utilities supplied with CP/M are referred to as transient commands in the manuals, but as we have seen, all this means is that they are .COM files on the system disk.

Our own programs will be accessed in the same manner; by the entry of a command line into CCP following the CCP prompt. As we have seen, CCP will parse that command line, and if it contains one or two file names, will load them into the default file control block. Other entries in the command line will be saved for our program in TBUFF. Our program is then off to a flying start.

There are a number of other tasks that CP/M can perform for our user programs. In the next chapter we will be seeing how to organize our programs to make maximum use of the services that CP/M provides.

Interfacing
With CP/M

The hypothetical COMPARE program in Chap. 5 could have been written to be invoked with the simple CCP command COMPARE. The program could then prompt the operator for the names of the two files to be compared. The program would then have to parse the file names as they were input, create file control blocks for both, and then ask the operator for any of the other options permitted.

Since the operating system can do all of this for us using a single command line input with editing features, it makes no sense to have these burdens placed on user programs. The system provides a large number of labor saving facilities, and they should all be made use of in your programs. That places the burden on you to learn what is available and how to use it.

The "giant hook" at location 5

In Fig. 5-1 we see the user transient programs accessing all of the facilities of CP/M through the BDOS CALL VECTOR that the operating system wrote in memory location 5 (Table 5-2). This is

the hook on which we hang all our requests for I/O and disk access services. Since we are assembly language programmers, and know the locations and operation of all the disk and I/O drivers in CBIOS and the PROM, we could have used those drivers directly. But that would be a poor programming practice.

For one thing, each different size and version of CP/M will have the CBIOS vectors starting at a different absolute address, and the user program would either have to figure out that address, or be written to run under only one size of one version of CP/M. Not too good for program portability. Since each version of CP/M knows where its BDOS ENTRY is located in RAM, it can set up the location 5 jump instruction to point to itself. Also, if any of the PROM routines were called directly, the program would run on only one hardware configuration.

The reason that this is being stressed here is that there is a distressing number of programs being sold today that are written in such a manner that it is difficult or impossible to adapt them to a new hardware or software environment. As we are seeing, that is not necessary. Programs can be written to run under any version of CP/M on any computer. Simply hang your service requests on the giant hook.

The services available from the system vary slightly from one version of CP/M to another, although Digital Research has been careful to avoid any conflicts when updating the operating system. To avoid any yourself, you should limit your use of system functions to the subset listed in Table 6-1. These are all you will be needing for quite some time in your programming efforts.

Way back in the Introduction, we loaded register C with a function code, put our data in register E, and called BDOS at location 5. This example performed a simple task; outputting one character to the console. Much more complicated tasks are available to us, ranging from the input or output of a whole text line, on up to the reading or writing of one disk record.

All of the system service calls make use of a common set of calling conventions. We have seen the simplest in our first programming exercise in the Introduction. More complicated functions will need to use more registers for passing parameters. We will be investigating all of these in detail throughout the rest of the book, as we make use of them.

Once again, we are discussing a topic in advance of your need

TABLE 6-1. The most commonly used disk and I/O access functions provided by the CP/M operating system. All are accessed through the single BDOS entry point vector stored in memory location 5.

		I/O Device Functions
Label	*Code*	*Function*
RCONF	1	Read character from console device
WCONF	2	Write character to console device
RRDRF	3	Read character from reader device
WPUNF	4	Write character to punch device
WLSTF	5	Write character to list device
RIOBF	7	Read IOBYT from memory location 3
WIOBF	8	Write IOBYT to memory location 3
RBUFF	10	Read console edited line input
CRDYF	11	Check console for character ready

		Disk Access Functions
INITF	13	Initialize BDOS, select drive A:
DSELF	14	Log in and select drive d:
OPENF	15	Open a file for read or write
CLOSF	16	Close a file
FINDF	17	Find a file in the disk directory
NEXTF	18	Find next occurrence of a file
DELEF	19	Delete a file
READF	20	Read one disk record into memory
WRITF	21	Write one record from memory to disk
MAKEF	22	Create a disk directory entry
SDMAF	26	Set RAM buffer address for read or write

Additional functions are available but are not commonly used.

to understand its details. This is to help you understand why the given approach to each task is not always the one which seems most straightforward. You can rest assured that any approach that seems roundabout at first glance has a very good reason for existence. Accept the dictates found in this book on blind faith, and the reasons for them will be revealed later.

Acceptance on faith is necessary at times because there is so much background material to learn before you can begin writing your own programs. More follows in the next section. Through necessity, discussions of hardware topics, software topics, and programming philosophy have had to be mixed together in this book. This is to help you understand the big picture, and build your understanding block by block.

Some blocks are hard, some are soft. They are each equally necessary for you to understand.

8080 ASSEMBLY
LANGUAGE
PROGRAMMING
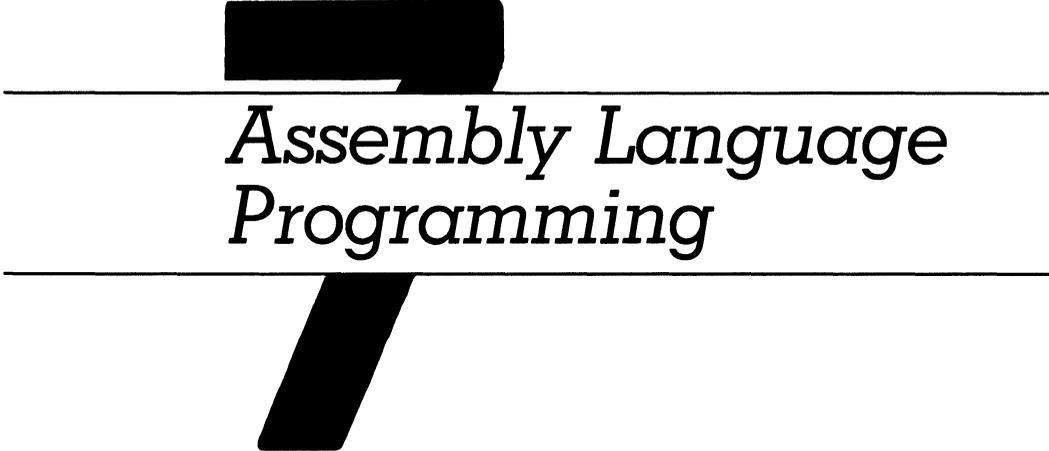
Assembly Language Programming

$$X^2 = X$$

George Boole

The binary number system uses digits that can assume only two states. These states are represented by the numbers 0 and 1. The equation $X^2 = X$ is true only if $X = 0$ or $X = 1$. On this basis George Boole developed the rules of formal logic, or Boolean algebra.

Machine language

The language that your computer understands is composed of binary digits that can assume one of two states: either of two voltage levels. These are variously referred to as logic high or low states, or logic true or false states, or voltage or current on or off. We can represent these two states by using the binary digits (bits) 0 and 1.

In the exercise in the Introduction, you keyed in a "machine

language" routine using DDT. While this use of the term machine language is common, it is not strictly correct. The actual language of the machine is composed of patterns of voltages that take one of the two possible binary states.

While you need not concern yourself with what is going on inside your computer in terms of voltages, you should have some feeling for what those voltages are accomplishing. The computer begins each instruction cycle by fetching an opcode from memory. This eight-bit pattern is placed in the instruction register. In the storage elements in this register, each bit will be at a high or a low level depending on the instruction. This set of eight voltage levels is the language the machine understands, strictly speaking.

In the exercise in the Introduction, we have taken the liberty of referring to a hexadecimal representation of those voltages as machine language. Well, we can't very well say that we want the computer to execute 0V 0V 0V 0V 5V 5V 5V 0V as our first operation. We have to use some more human-readable form of representing the contents of the instruction register following that first fetch. Since we don't want to spell out the voltage levels for each bit, we could use a shorthand, letting a zero represent a low level (zero volt) state, and a one represent the high (five volt) state. Our machine language opcode can now be represented as "00001110." And this is, of course, the binary representation of our first opcode.

If we had to enter all our programs into the computer using that old switches-and-lights type of console, this binary representation would be quite workable since each binary digit (bit) of either one or zero would correspond to one switch set either up or down. This binary word would then look like a map of our switch settings. And this is the way things were once done.

Today, writing programs in binary would soon wear out two keys on your terminal and leave the others unused. And of course it would be difficult for a human programmer to recognize more than a few binary patterns on sight. So more tractable representations of eight binary digits were developed. The first technique was octal representation. Here the binary word is broken up into three-bit groups, starting from the binary point, or right end.

The rightmost three-bit group in our first opcode is 110. In octal, if 000 is zero, then 001 is one, 010 is two, and 100 is four. You can see that 110 must be six, since it is four and two. Using only these basic patterns and their sums, we can represent numbers from zero through seven with three bits. This is a set of eight

FIGURE 7–1. An eight-bit byte containing a bit pattern expressed in octal, binary, and hexadecimal notation. The decimal equivalent of this pattern is 14.

| 0 | 1 | 6 | = 16Q |

| 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | = 00001110B |

| 0 | E | = 0EH |
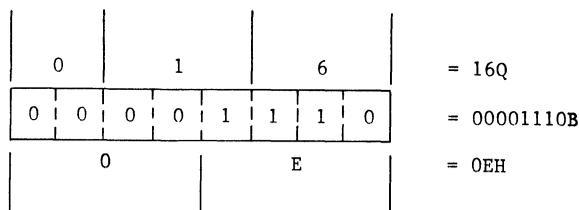
states, hence the name octal, based on the same root as octagon and octopus.

To represent the entire eight-bit opcode in octal, we take the six as the right most octal digit, next to the octal point. The next three-bit group is 001, or one, and that goes to the left of the six. This leaves only two bits for the most significant octal digit, and it is zero. So our opcode in octal is 016.

Now that we can write 00001110 as 016 in octal shorthand, we have made opcode patterns more human-readable, but have removed ourselves one more level from those voltage patterns inside the machine. But we are making things easier on the human element in the computer system.

There are those of us who once felt that it was regress and not progress when hexadecimal notation came along. It is not as easy to learn as octal, but it does make the representation of longer words a little simpler. We will be getting into hex, as it is known for short, in more detail later. For now, let's just take a quick peek at hex notation. We saw that eight bits doesn't break up evenly in octal, since we had groups of three, three, and two bits. The next step up from octal notation is hexadecimal, where we break up our eight bits into two four-bit groups. Now we need more symbols, which is where hex starts confusing us by mixing numbers with letters.

With four bits in each group, we need 16 symbols to represent each different hexadecimal digit. Hexadecimal means, of course, "six and ten." We ten-fingered humans had already invented zero through nine as symbols to use for counting, and hex notation borrows the first six letters, A through F, as additional number symbols. When we were evaluating binary bit positions to establish our octal digits, we said 000 = zero, 001 = 1, 010 = 2, and 100 = 4. Obviously then 1000 must be eight! And in hex, 1001

= 9, 1010 = A, 1011 = B, etc., on up to 1111 = F. We see here that eight (or 1000) plus two (or 0010) equals A. That makes A in hex (written 0AH) the same number as 12 in octal, and 10 in decimal. And on up to 0FH = 17Q = 15D.

Note in the above representations that if a hex number starts with a letter, we precede it with a zero (not an "oh") so everyone and every computer knows it is not an alphabetic word. We follow it with H as a tag meaning the value is in hex. Octal doesn't include letters, so leading zeros are never necessary, but our tag is "Q" instead of "oh" so it can't look like a zero. If we just write "15" without any tag we can expect everyone to assume that a decimal number is meant.

Discussions of number systems using differing bases are usually full of equations and confusions. Well, you can't have everything in a book this size. By the way, if OCT 31 = DEC 25, does that mean Halloween = Christmas?

The subject under discussion here is machine language. We have digressed a little to look at the common means for representing an eight-bit byte, which we then refer to as machine language. It is as close as we can come to the real language of the machine without using eight voltmeters and a door on the top of our CPU IC. In the rest of this book we will forget octal and be using binary and hex numbers exclusively, as is conventional in 8080 systems.

Since an eight-bit byte divides evenly into two hex digits, it is obvious that a 16-bit value divides evenly into four hex digits. We saw in Chap. 2 that a 16-bit address bus can select any memory address from 0 through 65535, so it is easy to guess that 0FFFFH = 65535. Eight bits can represent any value from 0 through 255, so 0FFH = 255. Our machine, with its 8080 family CPU, will concern itself with eight-bit opcodes and data values and 16-bit address values, neatly represented by two- and four-hex-digit numbers.

Back in our exercise in the Introduction, we keyed hex values for 10 eight-bit bytes into memory starting at hex address 100. This is our machine language program. All 10 of these bytes were not opcodes. The first byte, when decoded by the CPU, informed the CPU that it should fetch the next eight-bit value from memory and load it into register C. We will be studying registers in Chapter 9; for now just keep in mind that some opcodes are followed by eight bits of data, and yet others are followed by 16 bits of address.

The CPU will know, when it decodes the first opcode,

FIGURE 7–2. Instructions executed by the 8080 family of microprocessors can consist of one, two, or three bytes fetched from successive memory locations. When the first (opcode) byte is fetched, it is decoded by the CPU to determine if it is a stand-alone (single-byte) opcode, or if another byte of data or two more bytes of address data must also be fetched. The instruction decoder will then increment the Program Counter by the correct amount to point to the next opcode in the program.

Program
Counter

```
PC:
        ┌──────────┐ ┌──────────────┐
        │  OPCODE  │ │ NEXT OPCODE  │
        └──────────┘ └──────────────┘

PC:
        ┌──────────┐ ┌──────────┐ ┌──────────────┐
        │  OPCODE  │ │   DATA   │ │ NEXT OPCODE  │
        └──────────┘ └──────────┘ └──────────────┘

PC:
        ┌──────────┐ ┌─────────────┐ ┌─────────────┐ ┌──────────────┐
        │  OPCODE  │ │ ADDRESS LSB │ │ ADDRESS MSB │ │ NEXT OPCODE  │
        └──────────┘ └─────────────┘ └─────────────┘ └──────────────┘
```
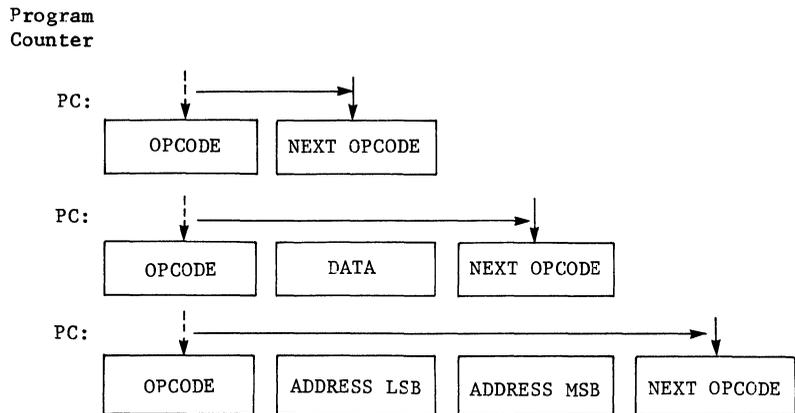
whether the next memory address will contain another opcode, or eight bits of data, or the first eight bits of a 16-bit address. Since a single instruction can consist of an opcode alone, or an opcode followed by one byte of data, or an opcode followed by two bytes of address, a machine language program will have more bytes in it than there are lines of code in the assembly language program that produced it.

Assembly language

Beginning with the true machine language of voltage levels within the CPU IC, we have progressed up through representations of that language expressed in binary bits and on up through hexadecimal codes. After a little experience with assembly language, you will learn to read these hex codes. You will find yourself translating from "0EH" to "MVI C" subconsciously. You will understand that

MVI C means to MoVe an Immediate value into register C. And
you will know that the next machine language byte in memory will
contain that immediate value. This process of mental translation
from the hex code to the assembly language mnemonic can be
called "disassembling" the program, because it is just exactly the
opposite of what the assembler does.

This discussion so far has been oriented from the bottom up.
Let's look at the process now from the top down. We start with the
programmer's initial definition of the task: To send the character
"$" to the CON: device. We know from Chap. 6 that we can do this
most easily by using a BDOS system call. This call is made through
memory location 5, so we first define absolute memory location 5
as symbolic location BDOS:

LISTING 7–1. The assembly language
version of the demonstration program
from the Introduction.

```
BDOS    EQU     5
WCONF   EQU     2
        ORG     100H
        MVI     C,WCONF
        MVI     E,'$'
        CALL    BDOS
        JMP     0
        END
```

Defining this symbol instead of just using "5" as the instruc-
tion operand within the body of the program insures that we can
use this same routine with other operating systems that might have
a different absolute location for system calls. Simply change the 5
in the definition of the symbol and the assembler will use the new
value wherever it finds the symbol name used as an operand.
Similarly the symbol for the "write this character on the console
function" is assigned the mnemonic WCONF. This symbol could
be anything you like, within the constraints set by the writers of
the assembler. In the case of CP/M's ASM, you can use up to 16
characters. The symbol should be meaningful and help you to
remember what the function performs.

Since this program will run as a transient program, it will be
loaded into the TPA at hexadecimal address 100. So we use the
assembler pseudo-operation ORG to direct ASM to create a ma-

chine language program that will run when loaded into memory at location 100H. ORG is called a "pseudo-operation" because it does not translate into a machine language operation. It is a directive to ASM telling it where in memory this program will execute.

Now we enter the program proper. To implement a system call we load register C with the proper function code, which we have symbolized as WCONF. Next we need to put the ASCII code for "$" into the E register. Again we MVI, but this time into E. We could have looked up the hex or decimal value for "$" in the table in Appendix A, and defined it as a symbol just as we defined WCONF. But since an equivalent table is included in ASM we don't have to. We just tell the assembler to do the lookup for us by including the desired character in quotes.

The next line performs the system call, and CP/M will decode the function in register C and send the contents of register E to the CON:. Following this action CP/M will return to the calling program at the next location in the program. The ball is now back in the programmer's court, and he can't just drop it! The program is over and now must transfer CPU control somewhere, so it is usual to jump back to CP/M at the CCP entry. This can be accomplished by an unconditional jump (JMP) to location 0, wherein is stored our vector to return to CCP.

The last line in our program is another pseudo-op, END. This lets the assembler know that no more source code is to be processed. Since this program, consisting of symbols, mnemonics, pseudo-ops, and absolute values, is what we feed into ASM, it is known as the assembly language source code. The assembler will read it twice, and generate an output file listing the original source program and the machine language it has generated:

LISTING 7–2. The assembler output print (.PRN) file of the program in List. 7-1.

```
0005 =           BDOS     EQU     5
0002 =           WCONF    EQU     2
0100                      ORG     100H
0100  0E02               MVI     C,WCONF
0102  1E24               MVI     E,'$'
0104  CD0500             CALL    BDOS
0107  C30000             JMP     0
010A                     END
```

This output file consists of the machine language program in human-readable format, and the source program. The whole idea for having ASM is to allow us to write programs in a source format that humans can understand, and then have ASM generate the program in a format the machine can understand. But the two portions of this file, the machine language in hex and the source code, are neither one understandable to the computer!

This file is strictly for giving the programmer a listing of what ASM did with his source program. This listing should then be printed, so it carries a file type .PRN. When we get into Chap. 12 and start debugging programs, you will understand how valuable this listing will be.

In addition to the .PRN file, ASM will also produce another file with "object" code in it. Aha! This must be the machine language code that we can run in the computer, right? Wrong! This second file consists of a hexadecimal representation of the machine language program, along with other information. We won't go into details as to the exact format of this intermediate code at this time, but if you look carefully at it you can see the hex machine language code embedded within it:

LISTING 7–3. The assembler output hexadecimal (.HEX) file.

```
:0A0100000E021E24CD0500C300000E
:0000000000
```

Since the basis of the coding within this file is hexadecimal, this file has a file type of .HEX. We give ASM our source code in a file of type .ASM, and it produces two output files with the same file name but types .PRN and .HEX. ASM then returns to CCP.

You are probably wondering by now how you are going to get your program into the TPA and run it. Using DDT in our first exercise, it took only moments to accomplish this. Now, even with the help of ASM, we still don't have a program we can run.

The intermediate .HEX file is coded in such a way that it consists of only printable ASCII characters. This means that the program in this format can be examined on the CRT (enter TYPE NAME.HEX) and can be transferred to another computer over a modem. Since a file containing the binary machine language will

contain ASCII nonprintable characters and control codes, it could not be so easily handled.

Before we can run our program we have to convert it from a .HEX file into a .COM file. Another CP/M transient utility, LOAD, is used for this. LOAD will read the .HEX file and produce a .COM file, and return to CCP. We can then run this program by calling the .COM file by name. Simply enter the program name and CP/M will load it and run it.

This entire process is required to assemble, load, and run a program. It may seem like a lot of bother for such a short program, and it is. But it is indispensable for long programs, as we will be seeing in Part V.

In this chapter we have been looking at machine language, assembly language, and the assembly process. Now it is time to look at the 8080 and the facilities it provides that can be made use of through assembly language programming. But before we can do that, we will have to take another short digression and look at hexadecimal numbers in more detail. If these discussions on the dull subject of number systems had been segregated into a separate chapter at the beginning of the book, you probably would have skipped it. Shame on you!

Hexadecimal numbers

When the ancients first began counting, they should have started counting on their fingers. Unfortunately, they counted on their fingers *and thumbs*. As a result, we are raised on the base ten (decimal) number system, and seem to feel that it is "natural."

This is not true. Decimal numbers feel natural to us only because we grew up with them. Things in nature occur naturally in powers of two. Amoebas multiply by splitting in half, so successive generations include 1, 2, 4, 8, 16, . . . members. We have recently been looking at binary, octal, and hex numbers. These are based, obviously, on members of the series above: 2, 8, and 16. If early humankind had counted on fingers only, and used thumbs as pointers, we might have started out with hexadecimal numbers.

Hold your hands up in front of your face, with the thumbs tucked out of sight away from you, and there you have our eight-bit byte (Fig. 7-1) expressed as two four-finger hexadecimal digits!

What could be more natural than that? By curling up fingers to represent zero, and leaving them extended to represent one, we can easily duplicate the binary pattern of any eight-bit byte. This naturally separates into two four-bit groups, our two hex digits. Be careful when you express 24H this way.

Since hex numbers are so natural, you'd think you would have been using them all your life. Surprise! You have been. All we are doing here is introducing new symbols to express each increment in a base 16 number system. You have been using a base 16 device for a large part of your life.

While decimal numbers seem natural to us, the fractions we used in grammar school actually are natural. The familiar inexpensive ruler has each major division (one inch) divided into 16 smallest divisions, and these are multiplied by 2, 4, and 8 for intermediate divisions. Naturally.

FIGURE 7–3. This fictitious "hexadecimal ruler" helps the newcomer to the base-sixteen number system visualize relationships within hex numbers. In hex, eight is half of ten, and four plus C is ten. Similarly, it can be easily seen that A plus 3 is D, etc.
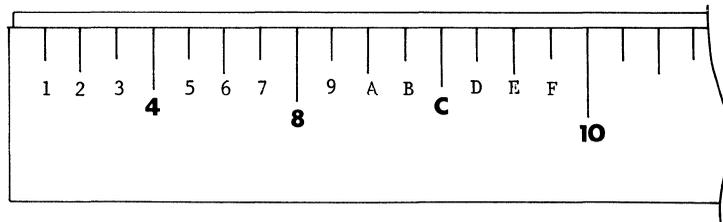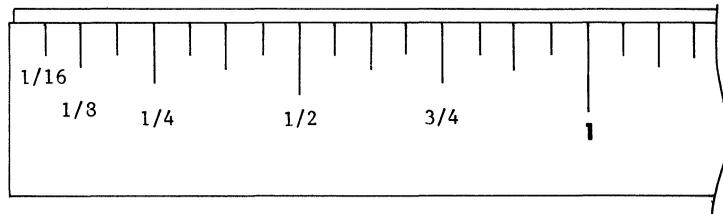
While it is stretching the truth just a bit to claim that hex numbers are the same as fractions, you can see from the illustration of the "hex ruler" that it is easy to learn the basic relationships in hexadecimal. Eight in hex is half of 10 (8H + 8H = 10H). Similarly, 4H is a quarter, and 0CH is three-quarters of 10H. Looking now at Table 7-1, we see the reason for this discussion. We use hex numbers as memory addresses, and speak of blocks of memory in terms of K bytes. Earlier we defined 1K as 1024 bytes, and mentioned that 1024 is two raised to the tenth power. Two raised to the eighth power is 256, and is the largest number that can be expressed with eight bits.

You should be able to see, now, how natural all these relationships are using hex numbers. Looking at Table 7-1, the relations between memory size in K bytes and the equivalent hex addresses form simple sequences. The only complicated numbers in the table are those silly decimals. Because they are not natural numbers.

Just as on the hexadecimal ruler where eight was half of 10, we can see from Table 7-1 that 8000H is half of our total memory address space. Since we begin mapping our memory address space

TABLE 7–1. Some common memory segment addresses expressed in different number systems: the xx K byte shorthand; decimal equivalents; and hexadecimal notation.

K Bytes	Decimal	Hexadecimal
64K	65536	10000H
48K	49152	C000H
32K	32768	8000H
16K	16384	4000H
8K	8192	2000H
4K	4096	1000H
3K	3072	C00H
2K	2048	800H
1K	1024	400H
½K	512	200H
¼K	256	100H
⅛K	128	80H

at location 0000H, our top address is FFFFH. If we counted from one up, our top address would have been 10000H, similarly to how we count from one to 10. In decimal, five is half of 10, and 50,000 is half of 100,000. In hex, 8H is half of 10H, and 8000H is half of 10000H. The other fractions work out just as nicely.

Referring back to Fig. 7-2 and Table 7-1 as we discuss memory addresses in future chapters will help you establish a feeling for hexadecimal, without the need for the usual rigorous discussions of number theory. That you probably wouldn't read anyway.

The 8080
Microprocessor
And Its Relatives

Version 2 (of anything) is the first version that works.
  Anonymous

The CP/M operating system executes on any of a number of different computers. These computers do not all have the same CPU chip inside. There are a number of microprocessors that will execute the same instruction set as the original Intel 8080, but also add new opcodes of their own. Any of these ICs can be used as the CPU in a computer running CP/M, since the operating system was itself written using only the standard 8080 instructions.

Anyone writing programs to execute within the CP/M environment should restrict his selection of instructions to those compatible with the 8080. In this chapter we will be looking at the features of the 8080 and its descendants, and will see what we must do to maintain program portability.

Characteristics of the 8080

The 8080 is an eight-bit microprocessor because that is the length of its data storage word in memory and also the size of its accumulator register. Since this CPU uses 16 bit addresses, as we have seen, it also includes some 16 bit registers and can perform some rudimentary 16 bit arithmetic operations.

We have seen that the largest number that can be expressed with only eight bits is 255, and even 16 bits will only get us up to 65,535 in decimal. Obviously there must be some way to handle numbers greater than these for any microcomputer to be a practical tool. Software is the answer, in particular the technique known as multiple precision arithmetic. Even the most expensive 32-bit minicomputers have to resort to multiple precision for some of their operations. The selection of eight bits as the basic size for this micro was made not on the basis of any required arithmetic precision, but as a compromise between IC chip complexity and the size of the instruction set.

In any computer there are only three operations you can perform. You can move data from place to place. You can operate on it mathematically. After an operation on the data you can make a decision based on the results and change the sequence of operations based on that decision. That's all you can do. Move, Add, subtract, . . . etc. Jump conditionally. Only three basic operations.

It is variations on these basic operations that requires a practical microprocessor to have more than a three-instruction set. There are a number of places you can move data to or from. There are lots of things you can do to it in addition to basic arithmetic operations. Things like logical operations, and shifts and rotates. And there are many possible tests to be performed on the results. Depending on the results of the tests, there are several variations on the basic conditional jump instruction. In the 8080 we have available 244 executable opcodes out of the 256 possible.

You may have heard that the 8080 can perform 72 basic operations, and that Brand X is better because it has 137 opcodes. The differences in evaluating how many instructions a particular computer can perform are the result of differing definitions of basic operations, and not anyone trying to mislead the purchaser. When we say that the 8080 can execute 244 different opcodes, we are counting all possible variations of each basic operation.

If you look at the numerical list of 8080 opcodes in Appendix B you will see that, of the 256 possible combinations of eight binary bits, the 8080 has implemented all but 12 of the bit patterns. This means that the designers of the chip have provided you with about all the computing power possible with an eight-bit wide opcode.

The next step up from eight bits in a binary sequence would be a 16-bit computer, which could offer 65,536 different operations. That would be overkill, and no 16-bit computer makes use of anywhere near that number of operations. So the eight-bit machines were a practical compromise, and the 8080 from Intel was the first microprocessor to offer real computing power in a single, affordable package.

Within that package exists a CPU consisting of an eight-bit wide accumulator register, some flag bits to record the results of operations on data, an arithmetic/logic unit (ALU) that performs operations on the contents of the accumulator and one other source of data, and a group of assorted registers. These other registers include six eight-bit registers that can be paired to form three 16-bit register pairs, and two dedicated 16 bit registers that are always used to point to locations in memory.

We will be examining in tedious detail in the next chapter how to use all these registers. The accumulator is obviously special. It contains one of the data bytes that will be operated on in the ALU, and generally receives the results of that operation. The two dedicated 16-bit registers are the stack pointer (SP) and the program counter (PC). The stack is a special area in memory used to store certain items, and the SP permits simplified store and retrieval operations. The program counter always points to the memory location containing the next opcode to be fetched. In our sample exercise (List 7-2), the PC would contain 100H, 102H, 104H, 107H, . . . successively.

The remaining six registers are for general purpose use. They can be used to store data, hold a count for repetitive operations, or can be linked together to form 16-bit index registers. Since our memory has a 16-bit wide address bus, we need 16 bits of address information to point to a memory location for storing data when we run out of space in our six registers. The PC points to the next opcode in our program storage area. The three index registers can point to other locations in memory for data storage. This is known as direct addressing, or absolute addressing, since the contents of

the index point directly to an absolute location in memory. By incrementing the contents of an index, we can point to successive locations in memory, as for instance locations within a lookup table.

We have already seen that following power up or RESET, the 8080 looks for its first opcode from memory location zero. We temporarily moved our bootstrap PROM to location zero to get things started at power up. The 8080 can also be forced to go to this location upon receipt of a hardware interrupt. This hardware interrupt is a signal from some device external to the CPU that says "Hey! Stop what you are doing and take care of my needs." The CPU will respond by stopping the currently executing program, saving the contents of the PC on the stack, and jumping to location zero, where it had better find an interrupt vector pointing to the routine that will service the interrupting device. You, the programmer, will have to provide that vector (a jump instruction) and that service routine, if you want this feature to operate properly.

The 8080 can recognize eight of these external interrupts. Since there are not that many unused pins available on the CPU package, some external hardware is required between the interrupting devices and the 8080. This hardware will put the correct bit patterns on the data bus to allow the CPU to know which vector to jump to when it recognizes that an interrupt has occurred. These eight vectors reside at the bottom of memory, spaced every eight locations, at locations 0, 8, 10H, 18H, . . . up through 38H.

That's the 8080. Eight bit wide accumulator and ALU. Six general purpose registers that can be linked to form three index registers. Eight interrupts available if some external hardware is provided. Absolute memory addressing. Two hundred forty-four executable opcodes. The original. The target of competition from some newer microprocessors that include the same features, execute the same instructions, but add hardware and software features not found in the good old 8080.

The Intel 8085

The major differences between the 8080 and 8085 microprocessors are in the methods used to fabricate the IC chip and the hardware improvements provided by the 8085. The 8085 is much simpler to use when designing hardware, as it uses fewer power supplies and

requires less external support. From the programmer's point of view, the 8085 has two new opcodes not found in the 8080, has a one-bit input and one-bit output port built in that can be used for serial communications, and has four hardware interrupt inputs that require interrupt vectors to be added to low memory if they are used.

All of these new hardware features are controlled by the two new instructions, which have opcodes of 20H and 30H. Since these were unused in the 8080, they are not provided for in the CP/M assembler, and we will not be discussing them in this book.

Another feature of the 8085 and the other newcomers to the 8080 family is that they will all execute instructions faster than the 8080 does. To avoid getting into hardware discussions in a book on programming, we have ignored speed of execution up to this time. The original 8080 was slow. Improved versions that could run faster were labeled 8080A, 8080A-1, etc. The same kind of speed designation is used by the manufacturers of other microprocessors in this family. To simplify things we will not refer to the various speed-selected versions of all these chips. When writing time-critical programs, you will have to know how fast your CPU can execute instructions, and that information will have to come from the computer system manuals and/or the microprocessor manufacturer's data sheets. Other than this mention of speed, we won't be getting into racing topics here.

The 8085, then, adds new hardware features and two instructions to control them. Otherwise, to the programmer, the 8085 is identical to the 8080.

The Zilog Z80

The designers of the 8080 left 12 unimplemented opcodes (the holes in the numerical listing in Appendix B) and when the 8085 was designed it made use of only two of these. The designers of the Z80 weren't so reticent; they used all of the 12 opcodes left over from the 8080. Since the Z80 uses the two new opcodes that the 8085 added, but uses them for different functions, we now have two new CPU family members that don't execute the same new instructions the same way.

Don't panic. The simplest solution to this problem will be to ignore the conflicts, and not use any instructions other than the

original 8080 set. This is recommended no matter what CPU your computer uses. So long as you stick to the 8080 instruction set your programs will run on any computer based on a standard version of CP/M. Provided you avoid the one incompatibility between the Z80 and the 8080.

One of the flag bits that records the results of ALU operations does so differently in the Z80. This very minor change resulted in a major disaster when one early version of an extremely popular program, Microsoft BASIC, blew up when loaded into a Z80 based computer. The detailed discussion later in this chapter on "establishing a common ground" will show you how to avoid any conflict with this incompatibility.

Avoiding conflicts is easier than having to accept the fact that all of the other nice features of the Z80 are not available to us when working with the CP/M assembler. Zilog made use of the 12 open opcodes to implement many times 12 new instructions. They did this by using a one-byte opcode to tell the CPU that it should fetch the next byte in memory and decode that as an entirely new instruction. In this way, a number of the previously unused opcodes are used as "windows" into a whole new instruction set.

This technique has its cost, however, because these new Z80 opcodes are effectively 16-bit instructions. Now two bytes have to be fetched from memory and decoded before the instruction can be executed. This uses more memory space and takes more time. But it does provide access to some powerful instructions that permit simple setting and testing of a single bit within a byte, or the movement of whole blocks of data with a single instruction. Again, remember that we can't make use of these goodies within the constraints of CP/M compatibility.

The Z80 improved on the hardware of the 8080 as well. It has its own new interrupt line, with yet another vector, this one at 66H. The original eight interrupt vectors can also be relocated in memory in a Z80 system. They don't have to sit at the bottom as in the 8080 and 8085. And there is a new addressing mode.

The 8080 provides absolute addressing. Jump instructions are to a definite location in memory. The Z80 adds PC relative addressing, where a jump can be specified to a position x bytes before or after the current program location. This allows programs to be written that can be relocated. Relocation means that once assembled, they can be loaded anywhere in memory regardless of the original ORG assignment, and still execute properly.

But for program portability, we have to code so that our programs can run on any CP/M computer, so all these great Z80 features will have to be ignored for now. Even though the Z80 provides more flexible interrupt vectoring, many new instructions, and relative addressing.

The National Semiconductor NSC800

Monday morning quarterbacks have it easy. They can sit back and benefit from the experiences of others. Designers of newer products can do the same, and provide a better product. The National NSC800 is obviously the result of examining the products of others and benefiting from their experience. The NSC800 borrowed its IC pinouts from the 8085, permitting those nice new built-in hardware interrupts, and then borrowed the super instruction set from the Z80. The best of both worlds, and with the added advantage that the NSC 800 is a CMOS integrated circuit. This means that it will run using a fraction of the power that any of its predecessors consumed. At a higher price, of course.

So if you are contemplating writing programs on your CP/M system that will then be executed in a battery powered portable device, you now have a 8080 compatible, CP/M compatible CMOS chip to design your programs for.

Establishing a common ground

After you have gained a little experience as an assembly language programmer, you will begin to appreciate some of the benefits to be gained by working with microprocessors other than the 8080. When designing a controller to operate at high speed in real time, the 8085 with its built-in hardware interrupts is desirable. For program-intensive applications, like writing a high level language compiler, the instruction set of the Z80 can reduce programmer effort significantly. But if you want to write a program that will run on any CP/M based computer, and will sell a million copies and make you and the IRS rich, you will have to stick with the instruction set of the good old 8080.

For your super program to be truly portable, it will have to be written so that it can be assembled with the CP/M assembler as

well. This will allow you to sell the source code at $50,000 a crack to people crazy enough to want to modify your perfect program. While assemblers for the 8085, Z80, and other micros are available that will run on a CP/M computer, their use reduces portability, and we will ignore their existence for the rest of this book.

We will restrict the instructions we use to those listed in Appendix B: the original 8080 set. Our programs will be written so that they can be assembled by CP/M's ASM, and execute in the TPA. And we will be careful about how we test for byte parity.

The 8080 includes two conditional jumps that test the parity of the contents of the accumulator that resulted from the last arithmetic or logical operation. Jump on parity even (JPE) will cause a transfer of program execution to the address specified if the contents of the accumulator ended up with an even number (0, 2, 4, 6, or 8) of one bits following an operation involving the ALU (data moves, such as MOV and MVI, don't go through the ALU in the 8080). JPO will jump if the number of one bits was odd.

In the Z80, JPE and JPO will work properly only if the preceding ALU operation was a logical operation: AND, OR, or XOR. The flag bit that only records parity in the 8080 is used also to indicate overflow following an arithmetic operation in the Z80. While this makes sense, and can be very useful, it is also in conflict with our standard, and can cause a program to execute properly on a 8080 but bomb on a Z80. It is easy to avoid any bombs when writing new programs.

All you have to do to avoid the conflict is execute a dummy logical instruction before testing byte parity. If a byte in the accumulator is logically ANDed with itself, nothing changes, but the flag bits will be set in accordance with 8080 practice, even if the CPU in use is a Z80. So if you have to test parity, execute AND A (AND accumulator with accumulator) immediately before the JPE or JPO.

If you got a little lost in that discussion, and don't even know why anyone wants to test parity, don't worry. After we discuss serial communications in later chapters, you will understand more about parity. The differences between arithmetic and logical operations will be covered later also. This discussion was included here because this is the place for discussing compatibility, before you get started writing programs that won't run on everyone's computer. Do that and just see if you ever get rich!

Register Usage
In the 8080

If our microprocessor can point to any one of 64K bytes of data in memory, why do we also want hardware registers for data storage? There are a number of reasons:

1. Speed of execution. Since the registers are part of the CPU chip, operations on their contents can be performed much faster than on the contents of memory.
2. Program portability. We know that no matter what CP/M based computer our program is executing on, we will have available the standard 8080 registers as a minimum.
3. Multiple indexes. Since some data will have to be stored in memory in most programs, having more than one memory pointer register simplifies programming tasks.

   Somo micro- and mini-computers have been built with as few as two programmer accessible hardware registers, and have become both successful and even popular. But virtually all programmers prefer to work with machines providing as many hardware

registers as possible. While most of the newer 16-bit and 32-bit micros have at least 16 registers, the 8080 provides a reasonable number of registers for most tasks. No matter what computer you are working with, you will always want more registers than it provides!

Register organization and data paths

In addition to knowing what registers are available, the assembly language programmer must know the paths that data follows between the registers, the ALU, and the outside world.

The "outside world" here refers to everything external to the CPU chip. This includes memory, and within memory in Fig. 9-1 we have shown the much ignored M register, which we will be looking at in detail below. First let's look at the internal registers.

The accumulator (A register) provides one of the two eight-bit inputs to the arithmetic/logic unit (ALU). The other input comes off the CPU's data bus. The results of the operation performed by the ALU are returned to the destination register over the same data bus. Condition codes are set depending on the action of the ALU, and these are stored in the flag register (F).

The F register includes condition bits that are tested individually by the conditional jump instructions. They tell us if two quantities are equal in size, or which one is larger, or if we have reduced a count to zero, or if the result of the last operation was a positive or negative number, or if it overflowed the accumulator. If we have performed an operation that resulted in a number too big to fit into the A register, the overflow will be recorded in the carry bit. We will be examining the actions of the bits within the flag register in detail later on, as we write programs that effect them, in keeping with our learning-by-doing philosophy.

Fig. 9-1 shows the A and F registers stuck together in one box, separated by a dashed line. The same is true of the B and C, D and E, and HL register pairs. This shows that these pairs of registers can be linked together to form 16-bit registers. The AF pair is unique in that the only paired operations possible are the stack operations PUSH and POP. Since these two registers are very special purpose, they do not participate in the 16 bit operations we will be discussing later. They are linked together only for stack

FIGURE 9-1. Register organization and data paths within the 8080 microprocessor, and as they extend to the outside world. The "M register" is actually a location in external memory. It is accessed by supplying a 16-bit memory address and then reading or writing eight bits of data.
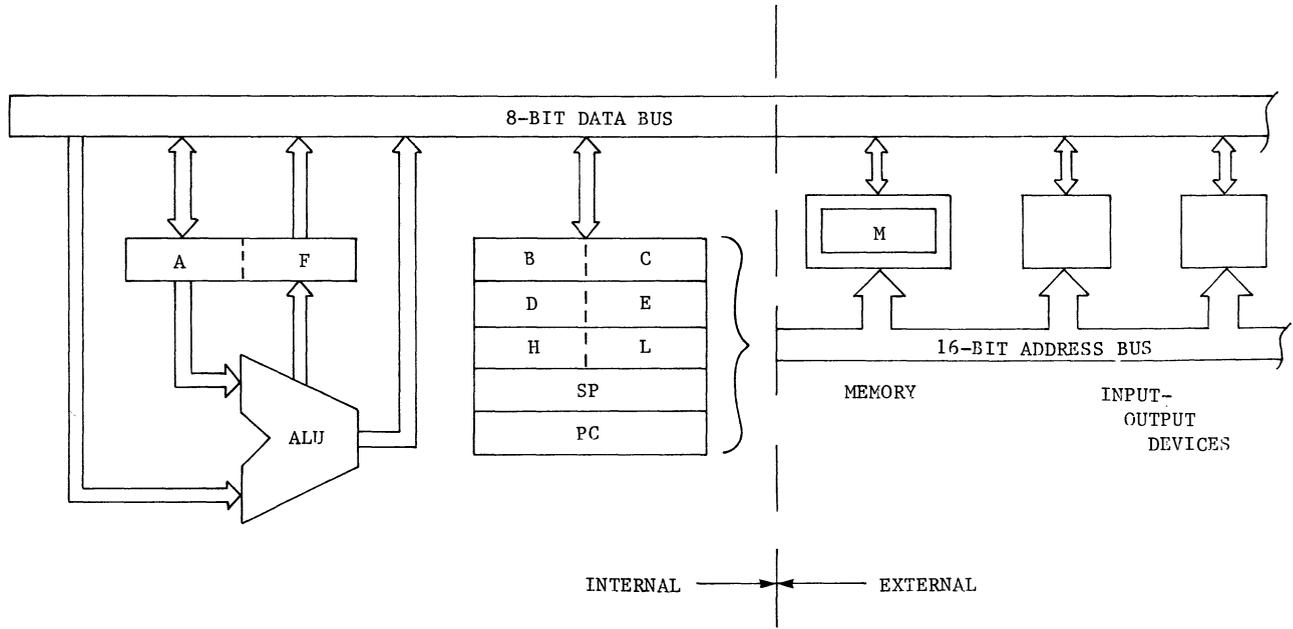
FIGURE 9–2. The flag register (F) records the results of arithmetic or logical operations performed by the ALU. Only five flag bits are implemented in the 8080 CPU. They can be tested by conditional jump, call, and return instructions.
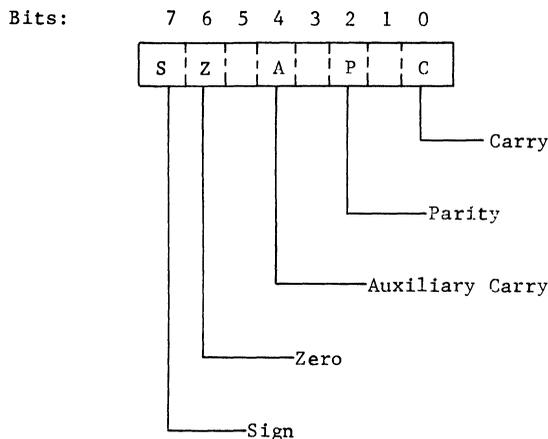
operations, and the pair is then known as the program status word (PSW).

The B, C, D, E, H and L registers are general purpose. Their contents can be the other input to the ALU, opposite the contents of A. Each of these six registers can be individually addressed for eight bit operations, and the paired registers BC, DE, and HL can be selected for 16 bit and stack operations. Data can be moved between registers and to and from memory, as well as processed through the ALU.

In operations involving paired registers, the registers shown on the left in the figure contain the most significant eight bits, and the righthand register the least significant half of the 16 bit value. These 16 bit values can then be used as pointers to memory locations, in which case the register pair is said to be an index register. When paired, the BC pair is referred to simply as the B register, DE is referred to as D, and HL as H, as in the instruction: LXI H,32767 that loads the HL pair with the value 32767. "LXI" stands for Load indeX with Immediate data. Immediate data follows the opcode immediately in the program. Index always implies a 16 bit register pair.

TABLE 9–1. Within all 8080 instructions that reference registers there are one or two three-bit register address fields. The register selections are specified by the three bits in conformance with this table.

Binary	Decimal	Register
000	0	B
001	1	C
010	2	D
011	3	E
100	4	H
101	5	L
110	6	M
111	7	A

Fig. 9-1 shows that the stack pointer SP and program counter PC can also form 16 bit addresses to select one memory location. This is all these registers can do. We know already that the PC points to the next instruction to be fetched from memory, so it must be dedicated to this purpose whenever any program is running. Which is the same as saying all the time. Stack operations will be discussed in detail a little later.

Embedded within opcodes that affect the contents of registers is a three-bit register address field. Since three bits can form eight combinations, this field can address up to eight registers for the opcode to work with. These register addresses are shown in Table 9-1.

The M register

This table shows that the binary bit patterns 000 through 101 (0 through 5) in the register address portion of an opcode select registers B, C, D, E, H, and L. The pattern 111, or seven, selects the accumulator. This leaves one pattern remaining, 110 (6), and this selects the M register.

This "register" is actually the contents of the memory location addressed by the HL index. The HL pair is sometimes referred to as *the* index register, because it is always the index for operations involving the memory register M. The other index registers BC and DE can only be used to move data between the A register and the memory location they point to.

The functional listing of opcodes in Appendix B shows that the contents of the M register can be operated on by all of the move, arithmetic, and logical operations that work on the contents of hardware registers. Since the M register can be any memory location, so long as the HL pair contains the correct address, it is easy to see that this is one of the most powerful of the 8080 general purpose registers. In spite of this it is usually the least used register.

This is caused by a human failing, one that computers are immune to, the mind set. It is too easy to think in terms of hardware registers, and ignore the M register because it is not part of the CPU chip. But a little thought will show how powerful a facility it is.

Suppose you are writing a program to fulfill the function of a calendar clock inside your computer. To simplify programming, you dedicate one eight-bit value as the count of seconds from zero through nine. Another eight bits will be used to count the tens of seconds, from ten to sixty. When the seconds reaches sixty, seconds counters are zeroed and the minutes units is incremented by one. And so on up through the years' thousands counter.

By wastefully using one eight bit value for each count you can simplify the programming effort. Simply count each time digit up, and if it reaches its limit value, zero it and increment the next higher digit. The M register is the logical choice for use in such a task, since there aren't enough hardware registers for all those digits.

When it is time to increment seconds (usually signaled by a hardware interrupt) start by loading the HL pair with the memory address of the seconds units counter, the first M register. Increment this count. Test for count = 10. If it is, zero the count, and increment the contents of HL. This makes the index point to the next M register, which contains the counts of seconds tens. Increment this M register. Test for count = 6 (6 tens = 60). If so, zero it and go on to minutes units, the next M register, and so forth.

FIGURE 9–3. A calendar clock program could be written using successive memory locations to store the time, date, and year digits. Each of these could then be accessed in sequence by incrementing the HL index register, pointing in turn to each digit within this stack of "M registers."

```
                                Clock Digit    Limit Count

                               ┌─────────────┐
                               │   YEAR T     │    10
                               ├─────────────┤
                               │   YEAR U     │    10
                               ├─────────────┤
                               │   DAY  H     │   3.66*
                               ├─────────────┤
                               │   DAY  T     │    10
              Index            ├─────────────┤
                               │   DAY  U     │    10
          ┌───────────┐        ├─────────────┤
          │  H     L  │        │   HOUR T     │   2.4**
          └───────────┘        ├─────────────┤
                               │   HOUR U     │    10
                               ├─────────────┤
                               │   MIN  T     │     6
                               ├─────────────┤
                               │   MIN  U     │    10
                               ├─────────────┤
          Index increments     │   SEC  T     │     6
                               ├─────────────┤
                               │   SEC  U     │    10
                               └─────────────┘
```
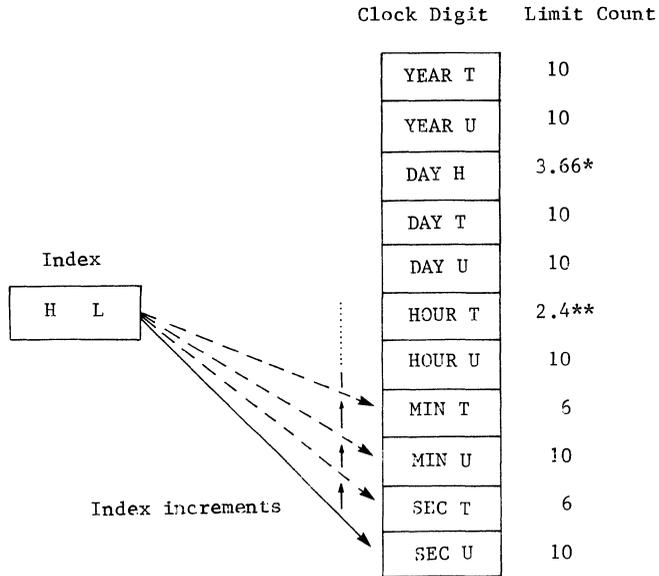
    *   Limit test must include HOUR U.

    ** Limit test must include DAY T and DAY U.

You can see from this that the M register is not one register. It is as many as we have room for in memory. Just point at the one you want by properly setting the contents of the index. Incrementing the index points to successive M registers in turn. This simplifies operations like the calendar clock program that operates on lists of values stored in the proper order in successive M registers, or memory locations. The power of the M register is that you don't have to load its contents into the accumulator before performing arithmetic or logical operations on it, as is true for some other CPUs. Just look at the functional list of operations in the Appendix,

and you can see that memory locations can be treated as registers in the 8080 family of microprocessors.

Stack operations

It takes one of us old timers to appreciate the 8080's stack. It is a recent invention, and worth its weight in gold. The stack is a memory area pointed to by the stack pointer (SP) and accessed by the stack operations. These include the register save operations PUSH and POP, the subroutine calls and returns, and the hardware and software (RST x) interrupts. There are other stack operations that are too incredibly complicated to explain here. You'll just have to wait a few chapters for XTHL and PCHL.

Suppose you are programming along and suddenly run out of registers. HL is tied up pointing to a particular place in a look up table, so you can't suddenly create another M register. All the other registers contain data you will need later, but just now you need a counter to keep track of an iterative operation. What to do?

```
              PUSH    B           ; SAVE B,C
              MVI     C,COUNT     ; LOAD COUNTER
     LOOP:    DCR     C           ; COUNT ONE
              JZ      END         ;   TIL DONE
              .       .
              .       .
              .       .
              JMP     LOOP
     END:     POP     B           ; RESTORE B,C
              .       .
              .       .
```

The dots here represent program statements within the body of the loop that required the use of a counter. We temporarily freed up the C register and used it for a counter by PUSHing the contents of the BC pair onto the stack, thus saving the values stored in them. We then loaded our count value into C (B remained unused) and entered the loop. We immediately decremented the count (just like subtracting one) and tested to see if it was reduced to zero. If C was zero, we jumped to END:, finishing the iteration. If C was not reduced to zero, we did whatever it was we wanted (the statements represented by dots) until we got to the statement JMP LOOP. This is an unconditional jump back to loca-

tion LOOP:. JZ was conditional: **J**ump if the result of the last operation was **Z**ero.

At END: we restored the original contents of BC by POPing B off the stack, which also restores the contents of SP to its original value. This is of immense importance. The stack is a handy place to stuff register data temporarily, but you must have an equal number of PUSHes and POPs.

This is important because the stack is also used to save the return address when a subroutine is called. In our exercise in the Introduction, which we also discussed in Chap. 7, we used a CP/M supplied subroutine to output a character to the console device. In List. 7-2, the CALL BDOS instruction at location 0104H automatically pushed the contents of PC onto the stack, and then jumped to location BDOS. Since the PC points to the next instruction to be executed, it contained the value 0107H. This was pushed onto the stack and the PC was then loaded with the address of the BDOS vector, in this case 0005H. The next instruction was fetched from location 5, and it was a jump to the actual location of BDOS high in RAM. All this time 0107H is sitting on top of the stack. BDOS does its thing, decoding the desired operation (WCONF) and sending the data (E) to the console. When BDOS is done, it executes a return instruction (RET). This instruction pops the contents off the top of the stack and places it into PC, so the next instruction is fetched from 0107H, and we are back in the user program.

Suppose that somewhere within BDOS the system needed to save some register contents, just as we did, or call another subroutine. Subsequent PUSHes or CALLs would operate normally, saving the required data on the stack. Each time a value is pushed onto the stack, the first thing that happens is that the SP is decremented, pointing to a lower location in memory. Then the high order byte of the register to be saved is moved into memory at the location pointed to by SP. SP is once again decremented, and now the low order byte of the register is written to memory at this address. No matter how many pushes are executed, none of the data previously written is destroyed.

Since pushes first decrement SP, then store a byte, then decrement SP, then store a byte, the stack fills up from the top down as far as absolute memory addresses are concerned. Perhaps an earlier stack oriented computer reversed the procedure. That

would account for the fact that illogical humans insist on referring to these actions as pushing data onto the top of the stack, and popping data off from the top of the stack.

It matters not that these descriptions are not technically accurate, and that the stack actually grows downward in memory, since the effect is the same in any case. The stack is a handy place to save data, the last register pair pushed on will be the first data popped off. Hence the common term for the stack, LIFO, for Last In, First Out. What is important is that we keep accurate track of stack operations, since some pops (returns) result in a transfer of data from the stack to the program counter. If the data on the top of the stack is register contents instead of a return address, because pushes and pops were not matched within the subroutine, your program will get lost!

In every programming book ever written, the action of the stack is likened to that of the spring loaded push down tray and dish dispensers in a cafeteria, where the last clean (sometimes) plate pushed on is the first grabbed off. Since this analogy has already been worked to death we won't even mention it here.

Register use by the user

By now you should have no questions about the use of the stack pointer and program counter. If you do, write each question on the border of a $20 bill (one question per bill) and mail them to the author (NOT the publisher!) of this book. You may get an answer someday, if you are lucky. Or you might just wait until Part 4, where things will become clearer as you actually execute programs and subroutines.

You should also by now realize that the A register is the most used register, since it is always one of the inputs to the ALU. The F register is not a programmer accessible register, in the sense that it can be addressed and have data moved into and out of it. Although the H and L registers can be used individually as eight-bit registers, it is more desirable to dedicate their use to that of the number one index register, which provides access to the M register(s).

That leaves us with four 8-bit registers, that can be linked to form two 16-bit indexes. If you try to always use the same registers

for the same purposes in all of your programs, you will find it much easier to keep track of their use throughout a long program. This can prove valuable when modifying an existing program or adding functions to it. You will have less trouble remembering what registers are in use and which ones have to be saved before being used within a subroutine.

For instance, if a second index register is required within a program, always use DE first. If you randomly use BC in some program segments, and DE in others, you will find it hard to remember which have contents that need to be saved temporarily within a subroutine. The worst case could be encountered when you are writing a subroutine that will be called from a great number of places within the main program. If you have always used DE before BC throughout the main program, it will be easy to remember if BC is in use at all. This could save you the time spent in pushing and popping BC when all you need is a temporary counter.

Since "counter" starts with C, why not always use the C register as your standard eight bit counter, and the BC pair when you need to count larger numbers? This seems obvious and hardly worthy of discussion now, but if you have ever had to modify an existing program, where the programmer used registers selected at random, you would understand why it is important to set down some basic guidelines for register use, and to stick with them.

The easiest guideline to remember is that the A register is the obvious candidate for passing an eight-bit value (like an ASCII character) to and from subroutines. Since A can't be paired to form a 16-bit value, and since it will have to be used for something in every subroutine we ever call, we can always assume that the first eight-bit value sent or received will be in A, and that the contents of A will always be destroyed in every subroutine we ever call. If we remember that rule, we won't be surprised by a subroutine that changes the contents of A. If we must save it, push it on the stack before calling the subroutine, and pop it off afterward.

Within your own programs, it is easiest to establish the rule that subroutines which are going to use registers other than A should save and restore their contents. You could save register contents in the main program, then call a subroutine, then restore the registers, if you knew that the subroutine is going to need to use the registers. But this places an unreasonable burden on falli-

ble human memory. It is better to write all subroutines in such a manner that, upon return to the calling program, all registers except the accumulator will be restored to their original state. Insure that you always code subroutines this way, and you won't tax your memory, or lose your data.

These are not hard and fast rules accepted by all programmers. They are guidelines that you should use as you start to design your own programs. If you just accept them blindly, for now, you will find that you are benefiting from the painful experiences of your predecessors. Why reinvent the wheel, or trip over it, just because everyone else does?

In summary, pass eight-bit values in the accumulator, and expect subroutines to destroy its contents in the process of doing their thing. Dedicate HL as your number one memory pointer. Use DE next, if you need it. Save C for an eight bit counter, and BC for bigger counting jobs. Write subroutines that will return the contents of B, C, D, E, H, and L unchanged, but can wipe out A if need be.

Follow these rules from the beginning and you will save yourself a lot of grief. Save the grief. You will need it in dealing with operating systems (yes, even CP/M!) that don't follow the same rules, and don't even establish equally logical ones for themselves. We will see how to deal with that situation in the chapter on preserving the user's environment. Meantime, we will look at how CP/M uses (and misuses) registers.

Register use by the system

We have repeatedly differentiated between the "user" and the "system." Just what is the system? It is the software environment you have to work within: the CP/M operating system and whatever firmware comes with your computer hardware.

The CP/M operating system itself was developed on an Intel Microcomputer Development System (MDS). The MDS included an extensive monitor in PROM that provided driver subroutines for a number of I/O devices, as well as extensive debugging facilities. Since this firmware was part of the computer, the writers of CP/M made their software system compatible with the tools available to them.

The most obvious tool was the MDS Monitor in PROM.

Many of its features have been adopted by CP/M, including the use of the IOBYT and those funny device designations. While all of this firmware is available on an MDS, your own computer may have had to duplicate a lot of the facilities of the MDS PROM within the space provided in the CP/M CBIOS. Obviously CP/M is a very adaptable software product. It was written so that it could be easily installed in any 8080 family microcomputer. Some of the characteristics of the MDS got integrated into CP/M in the process, so now all CP/M based computers look a little like the original MDS. We call this the MDS syndrome.

Along with the good MDS features inherited by CP/M are some not so good characteristics, like those terrible device designations and the way registers are used in the system calling conventions. We have previously defined how 8080 registers should be used. Now we have to look at how they must be used in interfacing with CP/M. Later we will see how to provide a single interface between the two worlds; a window between the logical world of our usages on one side and the MDS syndrome on the other side.

When we call BDOS to input an ASCII character from the CON: or RDR:, the character is returned to us in the A register. This is logical, as we have defined it. But when we send an ASCII character to CON:, we are forced to put the data byte in the E register, with the C register containing the byte that defines the output-to-console function. What happens to the A register?

The accumulator is the destination for all operations that input data from I/O devices connected to the CPU. Before the CON: can be read, it must have a character ready. The computer is much faster than the operator, so will spend most of its time waiting for you to press a key on the CON: keyboard. Way down there in the MDS PROM (originally) was a keyboard read routine that inputs a status byte into A, and tests it for the character ready bit. This read routine sits in a tight little loop, inputting status and testing for ready, until we finally hit a key. Then it reads the keyboard character into the A register, masks off the topmost bit, and returns to the calling program.

This is why we receive an input character in A. The IN instruction puts it there. When we send an output character, we do not put it in A because, way down there at that lowest software level, the driver first has to read a status byte into A to see if CON: is ready to accept our data byte. So if we put the byte in A, it would

have to be saved before the status could be read. To save programming effort in the driver, we are forced to use more of our registers in the calling program.

This aspect of the MDS syndrome distributes a burden throughout all high level user programs to save one register for the convenience of the driver program. Not very logical, but something we have to live with. In the next chapter we will see how to minimize this burden.

Other funnies were inherited from the MDS. When we output a byte sized value, we place the function code in C and the data in E. If we have to output a 16 bit value, we place it in the DE pair. That makes sense. Eight bits into E; 16 bits into DE.

If we input an eight bit value, it is returned in A. When we input a 16 bit value, ... Well let's see. It can't go into the AF pair, since the flag register is not general purpose. We just used the DE pair, that might be a logical place for it. Or, since we had to use the C register for our function code, so it has already been changed, it might be logical to use it for the other half of the returned value. No such luck. The MDS syndrome dictates that 16 bit values are returned with the low order byte in A and the high order byte in the B register.

OK! No more editorial comment about the logic of the MDS syndrome. We are stuck with it, and can learn to live with it, but we should know that the blame rests on other shoulders, not on those of the authors of CP/M.

Preserving the
User's Environment

Chapter 9 pointed out the desirability of creating a user environment separate from that of the operating system. Within this user environment we will use the 8080 registers as we have decided they should be used. The operating system had to use them the way its original environment dictated. We could go along with that, but experience has shown that there is a better way.

That way is to provide separation of the two environments, and establish an interface between the two worlds. An interface through which data will be passed, using subroutines that can be called from any place in any user's programs. These interface subroutines will bear the burden of maintaining the user's registers intact, so he will never have to worry about what the system has done with them.

A little extra effort in creating this interface, the window through which data will pass, will be well rewarded when you get around to writing large and complicated programs. You will always be assured that your environment is preserved intact, and this will increase the reliability and portability of your programs.

Establishing the user's stack

In the last chapter we saw that the stack is a memory area set aside as a handy place to stuff data, and a necessary mechanism for calling and returning from subroutines. Since the data we push into the stack grows downward in memory from the initial address contained in the stack pointer (SP), we must set aside a block of memory for the exclusive use of the stack, and set SP to point to the top of it.

The first thing we need to know is where some read-write (RAM) memory space is available. In a CP/M system, user (transient) programs execute within the TPA, and the TPA is in RAM. Later you will probably want to write programs that will be burned into PROM. You will have to remember to set up the initial address in the SP to point to the top of some unused RAM. Working within the TPA, all the space available to us will be in RAM, and all we will have to do is save a block for the use of the stack, and set the stack pointer initially to point to the top of that area.

How big should the stack area be? A safe size to start with is 64 bytes. You could execute 32 successive PUSHes of register contents, or nest 32 levels of subroutine CALLs, in that much space. Obviously, your actual use of the stack will be for some combination of PUSHes and CALLs. It would take a very complicated program to need all that space. We will start with this 40H stack area because it is more than enough, and we want to be sure that the space is never overrun.

To set up a stack area, way down at the end of each of our programs we will include these two lines of code:

```
        DS    64    ; STACK AREA
STAK:   DB    0     ; TOP OF STACK
```

"DS" is a mnemonic for **D**efine **S**torage, and sets aside a block of memory equal in size to the specified operand. "DB" stands for **D**efine **B**yte, and will set up a single-byte memory location with initial contents as specified by the operand. Note that one operand specifies the size of a block of memory, the other specifies the initial contents of a single location. DB and DS are pseudo-ops in that they tell the assembler how to set up memory areas, but do not produce any object code that can be executed.

We don't really care what the initial contents of the stack are, since we will have to push something in before we can pop anything meaningful out. We only included the second pseudo-operation so that we could label the address at the very top of the stack with the label STAK:. The address assigned to STAK: at assembly time will be loaded into SP at the very beginning of our program:

```
START: LXI    SP,STAK    ; SET UP STACK POINTER
```

if we remember to include this line. You will only forget it once! Very strange things happen to programs when no stack space has been allocated, and pushes wipe out part of the program, or return addresses are "stored" in PROM or nonexistent memory. You don't have to trip over the wheel if you don't want to. Just remember to LXI the SP at the start of all your programs, and to set aside stack space large enough for all the pushing and calling you will be doing.

If you are a sharp-eyed reader, you should be wondering by now why the dictatorial author didn't follow his own advice, and set up stack space and pointer in the example program in the Introduction (see List. I-1). How did the CALL BDOS work, if no stack space was allocated for the return address?

When the command "TEST" (for example) is given to the Console Command Processor (CCP) from the CON:, CCP will load the contents of a .COM file named TEST into the TPA. At this time the system is running using a stack space and pointer that have been initialized by CP/M. When the .COM file has been loaded into the TPA, CCP will begin its execution by a subroutine CALL (not a jump) to location 100H, the start of the TPA.

When your transient program has thus been activated, SP is still pointing into usable stack space within CCP. You can use this SP setting and stack space for very short programs, if you are brave. Since your program has been called by CCP, you can return to CCP without rebooting CP/M, if you are very brave.

In our example exercise, if the JMP instruction at location 0107H is replaced by a RET opcode (0C9H), the program would execute normally, but the system would not be rebooted, and the CCP prompt would reappear instantly. This technique can be used, with caution, by very short programs when you are confident you won't overrun the CCP stack space available.

How big is the available CCP stack space? The CP/M manuals are silent on the subject. Since we want our programs to be independent and transportable, don't start off on the wrong foot by relying on the CCP stack. Use it only to simplify example programs in your next book.

Saving the user's register contents

In keeping with the policy of separation of user and system, we will want to preserve the contents of all the hardware registers (except SP and A) every time CP/M is called upon for I/O services. Since we have set aside more than enough stack space, we can let BDOS use some of it whenever we call location 5. And, as we decided in Chap. 9, we will be passing data in the accumulator, so can expect that register to get wiped out in the process.

All of the other registers will be saved and restored each time we pass data through the user/system window. To accomplish this, we will write a series of I/O subroutines that start off by saving the contents of B, C, D, E, H, and L on the stack:

```
CO:     PUSH    B                       ; SAVE REGISTERS
        PUSH    D
        PUSH    H
          .       .
          .       .
```

which requires only three bytes of program space and six bytes of stack space. Now we could care less whether or not the system disturbs the contents of these registers, because upon return from BDOS we will restore the registers:

```
          .       .
          .       .
        POP     H                       ; RESTORE REGISTERS
        POP     D
        POP     B
        RET
```

before returning the our user's calling program. This example is part of a subroutine to pass one ASCII character to the Console Output function in BDOS.

Note that the POPs are a mirror image of the PUSHes. The

stack is a LIFO mechanism, so the last in (H) must be the first out. We could have also saved the contents of A, but this will seldom be necessary. We loaded A with the data to be passed to the system, so our program must know what that data was. Often, we will be done with it at this point anyway. And we can expect the operating system to return an error code in A indicating whether or not the output operation was a success. We will leave it up to the calling program to decide what to do about any errors reported.

Other subroutines with similar saves and restores will handle other device I/O, so our programs can expect to always have full access to all the working registers at all times.

Calling BDOS

Assume that some user program loaded an ASCII character into A, and called CO: expecting that character to output to the console. We know, from having accomplished exactly that function in our example program, that BDOS wants the character to be in E, and a function code (WCONF) to be in C, when BDOS is called. So after pushing all the registers onto the stack, all we need to complete subroutine CO: is:

```
          .        .
          .        .
          .        .
        MVI    C,WCONF          ; SELECT FUNCTION
        MOV    E,A              ; CHAR TO E
        CALL   BDOS
          .        .
          .        .
          .        .
```

inserted between the pushes and pops above. That three lines of assembly language programming should be pretty familiar to you by now.

Returning to CP/M

The console output subroutine just developed will be one of the library of subroutines we will be assembling in the next section. We have seen it grow out of the simple program example first presented in the Introduction. That example has now become a

usable subroutine that provides the desired interfacing between the user's register requirements and the operating system. By making use of similar subroutines for all system interfacing, we not only preserve the user's registers, but also provide a mechanism for easily adapting our programs to any other operating system, in the unlikely event that ever becomes necessary.

At the conclusion of every program, we will want to return to the operating system in an orderly manner, not by pressing the RESET switch. In the preceding chapter we saw how that could be accomplished with a RET instruction if the CCP stack pointer was maintained intact. This is never necessary, since CP/M has provided a reboot vector at location zero, and our programs can always terminate with a simple JMP 0 instruction. This is always the safe way, since then the operating system will be reloaded before it is reentered.

If there are any questions in your mind about the meaning of any of the instruction mnemonics and pseudo-ops used so far in these simple examples, you should refer to the 8080/8085 programming manual and/or Appendix B for an explanation before proceeding to the next section. You are now well on your way to becoming an assembly language programmer, and we will be moving along rapidly from now on. And about time!

IV

A LIBRARY OF
USER SUBROUTINES

Learning by Doing

All things are full of labor.
  Ecclesiastes

With all that background behind us, it is about time that we got started doing some serious assembly language programming. In this section we will be creating and assembling a series of I/O subroutines, and some test programs to exercise them.

  To make this learning process as painless as possible, the actions of the 8080 instructions will be explained as the instructions are used, just as in previous chapters, where you have already been exposed to a number of instructions. Since we will be learning by doing, let's start doing.

Getting to know ED

The CP/M manuals for ED, ASM, and DDT are complete and detailed, but you will not have to memorize all of their contents in order to get started writing, assembling, and debugging programs.

We will start this section with a short tutorial on these operations. This will not be a replacement for the CP/M manuals, but an introduction to them. You will have to learn most of their contents eventually, but you can get started with the simplified subset of commands that we will be presenting in this chapter.

Since we have become familiar with the little test program first presented in the Introduction, it is the obvious choice for your first exercise in learning ED and ASM. It constitutes a stand-alone program, that can be edited, assembled, and then executed in the TPA all by itself. As soon as we have done that we will finally abandon that program, and begin compiling the library of subroutines that you will use in all future programs as a means of separating the user from the system, as was discussed in the preceding section.

There are only two control keys you will have to learn to enter the text of the exercise using ED. These are CTRL Z and CTRL I. "CTRL" implies that the computer operator holds down the control key while typing the designated letter. In other words, press and hold the key marked CTRL or CONTROL, then press and release the designated letter key, and then release the control key. The designated letters were here shown in capitals, but the control code delivered by the keyboard will be the same even if the keyboard is not in the all caps mode.

Assembly language programs should all be typed in upper case only, if for no other reason than to make the resulting source programs more portable. All computer systems will recognize the caps only mode, since only a few years ago lower case was a rarity on inexpensive terminals like our old friend the TTY. So start by setting your keyboard to the ALL CAPS or ALPHA LOCK mode or its equivalent.

Let's get started. Fire up your CP/M based computer and insert a diskette into drive A: that includes ED.COM, ASM.COM, LOAD.COM and DDT.COM, along with lots of workspace. When CP/M prompts for a command input, enter:

ED TEST.ASMcr

where "cr" indicates your pressing the carriage return key. The editor will be loaded and will inform you that this is a new file, and then will sign on with the prompt"*" showing that it is in the command mode.

The first command you may have to enter is "-V" to turn off ED's automatic line numbering. If the "*" prompt appears indented 8 spaces from the left edge of your terminal display, your version of ED comes up with line numbering enabled. For consistency with the examples in this book, turn it off with the "minus V" command.

Now, since you want to be in the input mode, enter the command "I" and a carriage return (CR). The cursor will return to the left column of the next line on the screen. This is the only notice you will have that you are in the input mode. You can now start typing in the text of the program:

LISTING 11–1. The assembly language source code for the demonstration program in the Introduction.

```
BDOS      EQU      5
WCONF     EQU      2
          ORG      100H
          MVI      C,WCONF
          MVI      E,'$'
          CALL     BDOS
          JMP      0
          END
```

Type the four characters BDOS followed by CTRL I. If your keyboard has a TAB key, you can use it instead of CTRL I, since the action will be the same. The tab character CTRL I will cause the cursor to space over to the eighth column on the screen. Here enter EQU and another tab (since CTRL I = TAB we will refer to it simply as "tab" from now on). ED moves the cursor to column 16 following the second tab, and here you enter 5 and terminate the line with a carriage return. Now you are off and running as an assembly language programmer.

Enter the second line the same way:

WCONFtabEQUtab2cr

and to keep things lined up the way they are in Listing 11-1, enter the third line starting with the tab:

tabORGtab100Hcr

since it doesn't start with a label. And so on through the program. When all of the lines are entered, the cursor should be at the left column of the line following END. Now enter CTRL Z to exit the input mode and return to the editor command mode. Enter the command sequence:

B8Tcr

and ED will return his text pointer to the **B**eginning of the text buffer, and **T**ype eight lines: your program.

If you make a mistake while typing, and catch it as soon as you make it, you can use the DEL, RUB, or RUBOUT key on your terminal to erase the last character entered. On most terminals you will not see the cursor back up over the deleted character, but instead will see the character duplicated, indicating that ED has erased it internally.

If you don't catch the mistake until you list the whole program (B8Tcr), then you will have to skip down in the text to where the mistake is and correct it by substituting the correct word or replacing the entire line. Suppose your line 4 got entered as:

MIV C,WCONF

You can skip down from the beginning of the text buffer where the pointer is now sitting (following your command of B) and type the bad line by entering the command sequence:

3LT

which tells ED to move down 3 **L**ines and **T**ype one line. It is always a good idea to **T**ype the line before you try to change it, as then you will know you are starting with the text buffer pointer pointing at the beginning of the correct line. No, the incorrect line.

Now use the **S**ubstitute command to fix the error:

SMIVctrlzMVIctrlz0LTcr

by entering "S" for the **S**ubstitute command, a copy of the bad text (MIV), a terminator (CTRL Z), the desired text (MVI) and another terminator. Since ED allows you to string command sequences

together, we have added "0" (zero, not Oh) and LT before the carriage return. This says to ED that after substituting MVI for MIV we want him to point back at the beginning of the **L**ine and **T**ype it. Always include the **0LT** to make sure you changed what you wanted to change.

In these examples we have been using lower case letters to indicate control key keystrokes, like ctrlz and cr. The usual control keys CR and LF and RUBOUT or DELETE will produce actions, while the two-keystroke CTRL keys will be shown on your terminal as an up arrow followed by the designated letter. To make the lines more readable in this text, we have spelled out these controls. What you see on your CON: screen will differ slightly.

When the text of our example program is all correct, rewind the buffer pointer to the Beginning (Bcr) and exit ED by typing the single command E and a carriage return. The text will be written to the disk as file TEST.ASM, and CP/M will be reloaded. Enter the CCP command DIRcr and you should see two new directory entries:

```
TEST   BAK
TEST   ASM
```

created by ED. The .ASM file will contain your program, and the .BAK file will be empty, since this was a new file creation and not an update. You can use the CCP command TYPE to verify this (see Chap. 5).

You have now entered, and maybe corrected, an assembler source program. This has been done with an absolute minimum of ED commands and controls. In most of your work you will not be using very many more of the facilities offered by ED. All the complicated goodies that come with ED, like multiple find and replace command sequences, are nice to have when you need them, but there is no need to spend the effort to learn them for awhile yet. So let's get on to the assembler.

Assembling the TEST program

Using CP/M's assembler is your easiest task. Just enter:

```
ASM TESTcr
```

and let the assembler do the rest. If you have keyed in the source program correctly, you should get the following display as ASM does its thing:

LISTING 11-2. The console display resulting from assembling TEST.ASM.

```
CP/M ASSEMBLER - VER 1.4
010A
000H USE FACTOR
END OF ASSEMBLY
```

The assembly process requires two passes through the source program. Since TEST was such a tiny program, the process takes only seconds. Larger programs take much longer, as ASM has a lot to do.

The assembler reads the source code from file TEST.ASM during pass one, and builds up a symbol table. This table contains all the symbolic references and their associated byte or address values. For instance, pass one through TEST would have produced a symbol table containing entries for BDOS and WCONF. When these symbols are encountered during pass two, the assigned values (5 and 2) are fetched out of the symbol table and placed in their proper places in the object code buffer.

LISTING 11-3. TEST.PRN produced by assembling TEST.ASM.

```
0005 =          BDOS    EQU     5
0002 =          WCONF   EQU     2
0100                    ORG     100H
0100 0E02               MVI     C,WCONF
0102 1E24               MVI     E,'$'
0104 CD0500             CALL    BDOS
0107 C30000             JMP     0
010A                    END
```

The object code buffer produced by ASM is for its own internal use (we never see it directly), and would contain the program in its assembled form as shown in the second column of the .PRN file: 0E021E24. . . . The reason two passes through the source program are necessary is not obvious from this assembly run because there are no forward references for the assembler to resolve.

If you had put the equate (EQU) lines at the bottom of your program instead of at the beginning, ASM would not have known what values to assign to WCONF and BDOS the first time it encountered them. So it would have created entries in the symbol table with the proper symbol names, but would have left the corresponding values zero. When it encountered the equates at the end of pass one, it could have resolved the references and put the numbers in the symbol table next to their names, but would not be able to plug the correct values into the object buffer until it made another pass through the source code.

While ASM is smart enough to keep track of symbols and resolve forward references, it makes the program easier for humans to understand, update, or modify if you group your definitions of symbols at the beginning of your source programs. There are a number of other "do's" and "don't's" for you to learn. We will discuss them as the time arises during the next few chapters.

The command line we entered, ASM TEST, is the simplest way to assemble a program. ASM also provides option codes that can be appended to the command line. The format of these options is a little confusing, as options are entered following the source file name and a period:

ASM TEST.AZZcr

This looks like a FILENAME.TYP entry, but it isn't. It tells ASM to read the source file from drive A: (first option), and omit the creation of the .HEX file (second option) and .PRN file (third option). The first option can be any valid disk drive designation, where ASM will try to find the source file. The second can be a drive designation for the destination of the .HEX file, or Z to indicate that no output is desired. The third option can be a drive to receive the .PRN file, or Z for no output, or X to indicate that you want to see the .PRN listing displayed on the console, but you do not want it written to disk.

The options .AZZ come in handy when you are assembling a large new program that may have mistakes in it. Since no output is produced the assembly process is speeded up, and you can see assembler error messages which are always displayed on the CON:. There is no need to list these error messages here, as they are all in the CP/M ASM manual. You will get to know them soon enough.

In these exercises we are assuming that all input and output files will be on drive A:, so you will not need to be entering options during normal operations. Why complicate things when you don't have to?

Loading and running TEST

ASM produced two output files: TEST.HEX and TEST.PRN. The format of TEST.HEX is explained fully in Fig. 11-1. This format was inherited from the MDS system, so is universally known as the Intel HEX format. Since all of the characters in the file are ASCII, and since the file contains checksum error detection in each line, the file can be transmitted over any inter-computer medium.

In order to be loaded into the TPA and executed, the .HEX file has to be converted into the binary image of the program as it

FIGURE 11-1. The format of a .HEX file. This file is used to record the assembler output in a format that can be readily stored on disk and transmitted between computers. The file consists of printable ASCII characters, and a running checksum can be tested to verify error-free transmission.

```
                                    :0A0100000E021E24CD0500C300000E
Start character ─────────────────┘
Number of data bytes ───────────────┘
Start address ──────────────────────┘
Record type ───────────────────────────┘
Byte 0 ──────────────────────────────────┘
Byte 1 ──────────────────────────────────┘
Byte 2 ──────────────────────────────────┘
Byte 3 ──────────────────────────────────┘
Byte 4 ──────────────────────────────────┘
Byte 5 ──────────────────────────────────┘
Byte 6 ──────────────────────────────────┘
Byte 7 ──────────────────────────────────┘
Byte 8 ──────────────────────────────────┘
Byte 9 ──────────────────────────────────┘

Checksum ────────────────────────────────┘


End of file record ─────── :0000000000
```
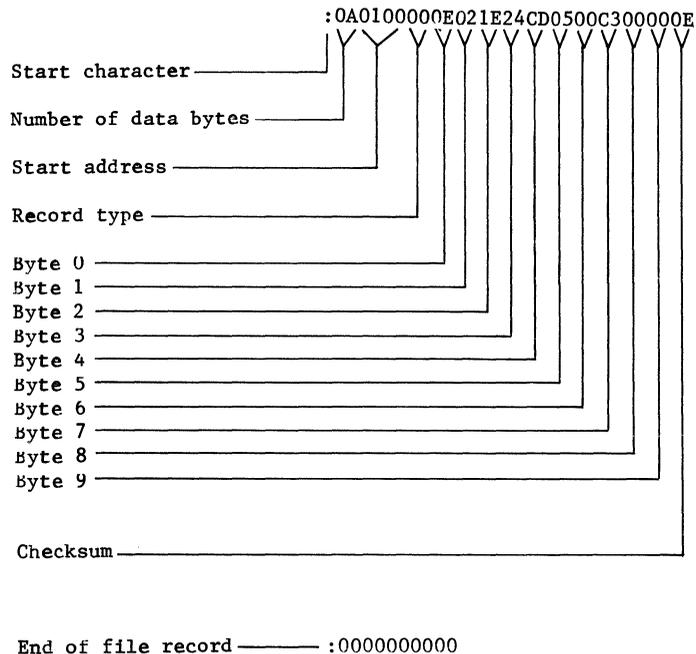
will appear in RAM. The binary image can't be transmitted from computer to computer because it consists of eight-bit bytes, and some modem interfaces want to see seven bits and a parity bit. In addition, since every possible combination of eight bits can be part of a binary file, there would be no way to signal End of Transmission (EOT).

The Intel .HEX format is ideal for data transmission, since it can be transmitted in either seven or eight bits per byte format, and is readable by both machines and humans (with proper programming incorporated in both). .HEX files can be downloaded into PROM programmers that include an ASCII interface, and are recognized by other operating systems as well as by CP/M.

It is the function of LOAD.COM to convert .HEX files into the binary memory image that can be loaded into the TPA and executed. Since it is quite possible to ORG a program at any address, LOAD.COM includes a test and will only accept .HEX files that begin at location 100H. To create a binary image of programs that begin at other addresses, you will have to use DDT. For all of the exercises in this book, you will only be loading programs into the TPA, so the simple command:

LOAD FILENAMEcr

will work nicely. Note that the file type .HEX need not be appended to the file name when LOAD is invoked, since that is the only file type meaningful to LOAD.COM.

Now you are ready to execute the TEST program. Enter:

LOAD TESTcr

and in seconds you will have a TEST.COM file on your disk. To execute the program, simply enter the CCP command TESTcr and you will see the "$" appear on your screen, followed by a return to CCP. Just like in the exercise in the Introduction. Only 11 chapters later!

Exercises

It is strongly recommended that you spend a couple of hours experimenting with the files you have now created on your disk,

before going on to the next chapter. There are a number of things you can try.

Assuming that you have keyed in TEST.ASM correctly, and that it produced .PRN and .HEX files identical to those shown in this chapter, and that it executes properly, you can now try inserting mistakes into the .ASM file. For instance, change the names of the symbols in the equate statements so that they no longer match those in the program itself. See what comes out of ASM, both the error messages and the results of having unresolved symbolic references.

You might also try to ORG the program at location 100 instead of 100H. You might as well see what this produces by way of errors, because it is a mistake you are sure to make sooner or later in your own programs.

After you have inserted some mistakes and have seen the errors and error messages they produce, restore the program to its workable configuration. Now try replacing the "JMP 0" with "RET." This will show you that CCP did CALL location 100H after it loaded the test program, and that it can be RETurned to.

Next you might want to demonstrate to yourself another interesting characteristic of stack operations. Replace "CALL BDOS" with "JMP BDOS" and run TEST. Try to figure out what happened, and how the stack contents produced the results you saw on your CON: screen.

More on ED

Before you can perform these exercises, you have to learn another ED command. When you first keyed in TEST.ASM, it was a new file. You entered it into the text buffer, reviewed it, and then wrote it out to file TEST.ASM with the command E. Before you can update the file, you have to re-enter ED with the CCP command ED TEST.ASM. Since TEST.ASM already exists on the disk, a new file will not be created. When ED first prompts with the "*" you will have to tell him to read in the existing copy of TEST.ASM. Enter the command #Acr and ED will Append all of the existing file TEST.ASM. This is the trivial case of the command Append. As long as your program file is smaller than the editor buffer you can read it all in at once, and the "#" symbol indicates that you

want the entire file appended to what is in the editor buffer. Since there is nothing in the buffer to start with, TEST.ASM will be read in and will be the only contents of the buffer.

Now ED will change the name of the original source file to TEST.BAK (for backup), and when you exit ED, the updated source program will be TEST.ASM.

After you have edited TEST.ASM and run the experiments listed above, you can experiment with the "#A" command by repeating it a couple of times. ED will append the file a couple of times, and you will have more in the editor buffer than you want. To keep from garbaging your file on disk, enter the ED command Qcr for **Q**uit, and ED will terminate without writing anything to the disk.

When you are comfortable with using ED, ASM, and LOAD, it will be time to start generating your own copies of the CP/M I/O subroutines in the following chapters. They will make your assembly language programming under CP/M much easier and more error free than if you started by just sitting down at your CRT: to bang out programs from scratch, like so many wheel designers did in the past, who had to survive without all this good fatherly advice from an old man.

Console Input/Output

If you have been writing any BASIC language programs, you probably just started at line 10 and wrote your program line by line to accomplish its desired mission. Higher level languages and assembly language programs should be structured a little differently, with the overall task broken down into blocks.

Program building blocks

Each block should perform a single function. Some dictatorial authorities go so far as to say that each block can have only one entry point and only one exit. That is a desirable goal, usually, but it is a rule that can be violated without causing the whole program to collapse. Error exits from subroutines, for example, violate the rule and are not uncommon.

The reason programs should be broken down into blocks is that it simplifies the programming effort. Each function to be implemented is written and debugged separately. When all the

blocks have been checked out, they are tied together by the main program.

In assembly language programming, each block should be a subroutine, and the main program, ideally, will merely call each subroutine in turn until the job is completed. The stack organization of the 8080 greatly simplifies this task. The lowest level blocks can perform the simplest tasks, like, for instance, outputting one ASCII character to the console. A higher level subroutine can output an entire message to the console by fetching characters out of a buffer and sending each one in turn to the lower level subroutine.

The stack makes all this possible. Each successive CALL results in another return address pushed onto the stack. The programmer doesn't need to keep track of what subroutine level he is working at. Nesting of subroutines is handled effortlessly by the stack. The main program can call a subroutine which can in turn call another subroutine at a lower level. With the stack doing the hard work, the main program could also call the lower level subroutine directly, if the task required it.

This flexibility is illustrated in Fig. 12-1. Here we see a greatly simplified schematic diagram of a program structure. The main program calls subroutines that in turn call other subroutines that handle data transfers to and from I/O devices, by using the BDOS call at location 5. These CPMIO subroutines can also be called directly by the main program as well. The first example in this figure shows CPMIO subroutine CO: being called by both the main program and at another time by an intermediate level subroutine, that had first been called by the main program. While only one level of user subroutines is shown in this diagram, the only limit to subroutine levels is the depth of the stack space created by the user.

So long as each subroutine performs a single function and uses the CPU registers in an orderly fashion, it is easy for the programmer to organize them in any order he needs to perform the desired task. These principles are illustrated by our CPMIO subroutines.

We have seen that it is desirable to separate the user's register utilization conventions from those of the CP/M operating system. It is also highly desirable for all communication with the outside world to be handled by the CP/M BDOS call at location 5. The CPMIO subroutines fulfill this dual function. Each provides

FIGURE 12-1. The interface between the user and the system. User programs and subroutines can access peripheral devices through the CPMIO subroutines which in turn call for system services using the location 5 vector. Register saving and restoring within the CPMIO subroutines preserves the user's program environment.

USER'S STANDARD REGISTER USEAGE ──────── CP/M REGISTER USEAGE

CALL ──▶   CALL ──▶   CALL ──▶
◀── RET   ◀── RET   ◀── RET

CI

CO

CRLF

GET

0005

BDOS CALL VECTOR

CBIOS

BDOS

USER MAIN PROGRAM

USER SUBROUTINES

CPMIO SUBROUTINES

DEVICES

TRANSIENT PROGRAM AREA (TPA) ──▶

THE "GIANT HOOK"

◀──── CP/M OPERATING SYSTEM
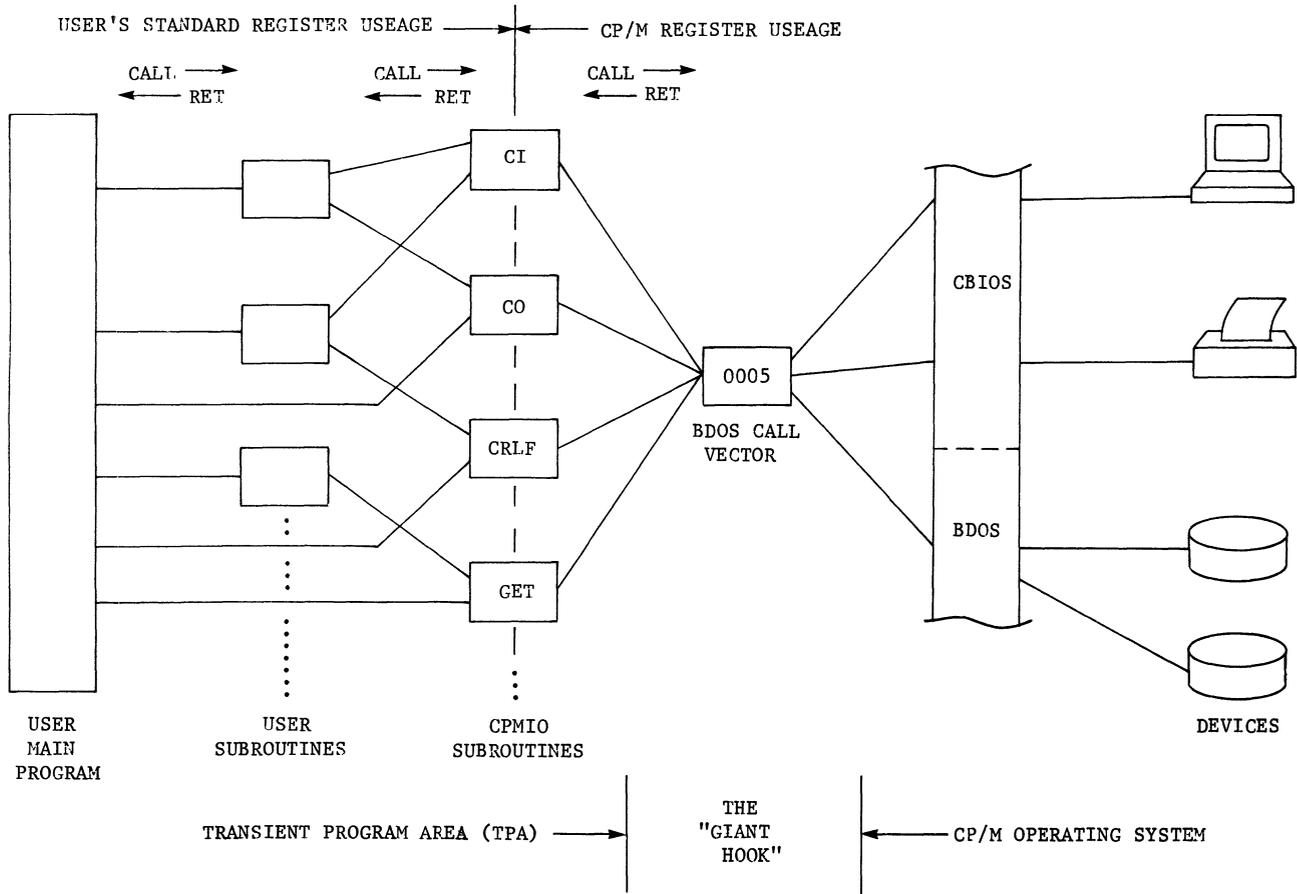
access to one I/O device at a time. When only a single byte of data is to be transferred, it is passed in the A register, and all the other working registers are preserved intact. The programmer doesn't have to remember what CP/M is doing to the register contents.

In this chapter we will be generating the code that will implement the two simplest functions: writing the contents of the accumulator to the console screen, and reading a character from the console keyboard into the accumulator. In order to test these functions, we will include a test program that will exercise these basic building block subroutines.

CI:, CO:, and a test program

List. 12-1 is the subject of discussion in this chapter. It includes the console input and output subroutines that are the first part of CPMIO, and a test program to exercise these routines. When you have entered, assembled, and loaded this software, you will be able to communicate with yourself on your CON: in a seemingly useless manner.

As you will be seeing, the communication will be useless except as a test of the subroutines. Generating the most simple-minded possible test program may produce silly results, but it does provide a means for testing each building block separately. When this has been accomplished, more blocks will be added to CPMIO, and a new test program will replace that shown in this first listing.

LISTING 12–1. Assembly language source code file CPMIO.ASM.

```
; CP/M I/O SUBROUTINES   30 JULY 82

; ASCII CHARACTERS
CR      EQU     0DH             ; CARRIAGE RETURN
LF      EQU     0AH             ; LINE FEED
CTRLZ   EQU     1AH             ; OPERATOR INTERRUPT

; CP/M BDOS FUNCTIONS
RCONF   EQU     1               ; READ CON: INTO (A)
WCONF   EQU     2               ; WRITE (A) TO CON:

; CP/M ADDRESSES
RBOOT   EQU     0               ; RE-BOOT CP/M SYSTEM
BDOS    EQU     5               ; SYSTEM CALL ENTRY
TPA     EQU     100H            ; TRANSIENT PROGRAM AREA
```

LISTING 12-1. *Continued*

```
          ORG     TPA                   ; ASSEMBLE PROGRAM FOR TPA

START:  LXI     SP,STAK               ; SET UP USER'S STACK
START1: CALL    CI                    ; INPUT A CONSOLE CHARACTER
        CPI     CTRLZ                 ; OPERATOR INTERRUPT?
        JZ      RBOOT                 ; YES, RETURN TO CP/M
        CALL    CO                    ; NO, ECHO IT AND
        JMP     START1                ;   LOOP

; CONSOLE CHARACTER INTO REGISTER A MASKED TO 7 BITS
CI:     PUSH    B                     ; SAVE REGISTERS
        PUSH    D
        PUSH    H
        MVI     C,RCONF               ; READ FUNCTION
        CALL    BDOS
        ANI     7FH                   ; MASK TO 7 BITS
        POP     H                     ; RESTORE REGISTERS
        POP     D
        POP     B
        RET

; CHARACTER IN REGISTER A OUTPUT TO CONSOLE
CO:     PUSH    B                     ; SAVE REGISTERS
        PUSH    D
        PUSH    H
        MVI     C,WCONF               ; SELECT FUNCTION
        MOV     E,A                   ; CHARACTER TO E
        CALL    BDOS                  ; OUTPUT BY CP/M
        POP     H                     ; RESTORE REGISTERS
        POP     D
        POP     B
        RET

; SET UP STACK SPACE
        DS      64                    ; 40H LOCATIONS
STAK:   DB      0                     ; TOP OF STACK

        END
```

The ten lines of code beginning CO: have all been discussed at length previously. It is the Console Output subroutine, that sends the character in the accumulator to the CON: device, through the CP/M system call, preserving all the other registers intact.

Above CO: in the listing is subroutine CI:, which will receive a single ASCII character from the console. The only differences between this subroutine and CO: are the change in the BDOS function code from WCONF to RCONF (Read CONsole Function), and the line

ANI 7FH ; MASK TO 7 BITS

The "**AN**d **I**mmediate value with accumulator" instruction is one of the logical (as opposed to arithmetical) operations performed in the 8080 ALU. It is used here to mask off the high order bit of the byte received from the console device. This masking should always be performed when receiving an ASCII character from any input device.

If you look at the list of ASCII codes in Appendix A, you will see that no more than 7 bits are ever used in a valid ASCII character. This permits the use of the eighth bit as a parity bit, to test for errors in transmission of the character. Since different data communication paths will produce different contents for this bit, all input routines should mask it off to prevent later confusion.

The masking is produced by ANDing the character with the bit pattern 01111111, or 7F in hex. Following the Boolean rules for AND:

Zero AND zero is zero
Zero AND one is zero
One AND zero is zero
One AND one is one

we can see what happens to the character "U" (55H) when it is ANDed with the mask word 7FH.

In this example, the communications path delivered the character with the parity bit (bit 7) set to one. This changed the character from 55H to 0D5H. Since we are programmers and not hardware engineers, we don't know the characteristics of the

FIGURE 12-2. Masking off the optional eighth parity bit from an ASCII character insures that the remaining seven bits represent a valid code. The parity bit may or may not be set during data transmissions, resulting in a potentially undecipherable code.

```
Bit    7  6  5  4  3  2  1  0

       1  1  0  1  0  1  0  1              0D5H

       0  1  1  1  1  1  1  1      AND     07FH

       0  1  0  1  0  1  0  1       =      055H
```

communications path, and don't really care, since we will always mask off the extra bit when receiving an ASCII character. It doesn't matter whether this bit, as received, was a zero or a one. We AND it with zero, and the result is always zero. The other bits are all ANDed with ones, so remain unchanged. You can verify this, bit by bit, using the Boolean rules above.

CI: will receive each character as it is typed on the console. To test our two subroutines, we need someone to type on the console, and a test program to do something meaningful with the typing so we know that CI: and CO: are working.

The test program begins at START:, and the first thing it does is what every program should do: establish a user's stack (see Chap. 10). The next line has a new label, START1:. This line inputs a character from the console, and the following line tests the character to see if it is the escape character CTRL Z (1AH). ComPare Immediate compares the current contents of the accumulator (our input character) with the contents of the memory location immediately following the opcode, and if they are the same, the zero flag is set. We test the zero flag with the next instruction, the conditional jump JZ, for Jump on Zero flag set. If the character received was CTRL Z, the program will jump to RBOOT, and reload the CP/M system. If the zero flag was not set, execution continues with the instruction that follows the conditional jump.

The next instruction is CALL CO, and the character we just input will be sent right back to the console device. Following this the unconditional jump will send us back to START1:, and we will input another character.

In this simple program, we could have jumped back to START: and reset the stack pointer. But that is unnecessary, and the logical limits of our loop should be the CALL CI and the CALL CO. Making all loops as short as possible may use more labels, but it speeds up execution time and makes the logic of the program easier to understand.

Notes on the listing

Comments are to be read by either humans or programmers. We use the semicolon to tell the assembler to ignore them. Comments can take the form of a whole line, with the semicolon in the first

character space, or can be appended to a program line by tabbing over from the operand field and starting with a semicolon.

There are no hard and fast rules for commenting. The program examples in this book are heavily commented to help you learn assembly language programming. In general, comments should be included to indicate what the program module accomplishes, who wrote it, when, and how it works. Any unusual or tricky techniques should be explained.

The first line in every program module should tell what the module is and should include a date. The programmer should always keep the date current. Each time you update a source program file, start by updating the date in the first line. There is nothing more frustrating than to find four different versions of a program and not know which is the latest. Dates are easy to keep track of. Version numbers, like V2.3, are too easy to get confused. They may be fine for tagging major revisions of giant operating systems like CP/M, but you probably won't be ready to write programs like that this month.

Numeric constants should be declared at the beginning of every program. Never write a program that includes lines like

OUT   123

which would cause the contents of the accumulator to be output to hardware port number 123. Sooner or later you will want to run that program on another computer, or an upgrade of your own computer, and the hardware port assignments will have changed. Trying to find 89 occurrences of numeric constants buried within a 42 page program listing is slave labor and not a suitable job for programmers, or humans either.

Of course we know better than to address hardware ports anyway. All our I/O will be through the system call at location 0005. But since that location could conceivably change too, we will define it symbolically as well.

In the beginning of our listing we establish symbols for every numeric constant we will need. These include the non-printing ASCII characters that we can't embed in the listing by surrounding them with quotes. Absolute memory addresses are also defined as needed. For this program we defined the reload CP/M vector at location 0000, the BDOS call vector at location 0005, and the start

of the TPA at location 100H. The function codes are similarly
defined. While reading through a long program, it is easier to
remember what "WCONF" means than to figure out what "2"
means.

Imagine that sometime in the near future you have written a
super program that will balance your checkbook while at the same
time watering the grass and feeding the cat. You will want to share
that program with the world (at a price, of course), and some
members of the world population may have computers that run
some operating system other than CP/M. Well, some people are
like that. At any rate, the chances are that any acceptable operating
system that runs on the 8080 family of microprocessors will have
hooks to I/O devices similar to the BDOS call. But at a different
memory location, and with different function codes. If you follow
the examples for defining symbols in this book, you will be able to
reconfigure your programs for any operating system on any 8080
family computer. Just change the numeric quantities associated
with the functions and addresses and your program is adapted to a
new environment. We weren't typing in all those extra lines for
nothing!

When you do sell your super program to 18,942 users of 38
different computers running 12 different operating systems, just
remember who started you in the right direction. Ten percent
would be a nice remembrance.

Even more ED

In Chapter 11 we learned enough ED controls and commands to
enable us to key in a simple little test program. If you are an expert
touch typist you could do the same with CPMIO.ASM using only
those simple editor facilities. But sooner or later you will have to
learn some additional ED commands, as the demands made on you
by the exercises in this book become greater.

In that first ED session, we skipped ahead a bunch of lines
using the command 3L for skip forward 3 Lines. The number of
lines can be either positive, to skip ahead, or negative, to skip
backward, as in -3LT which would move us back three lines, and
type one line. It is always a good idea to add the T to make sure you
are on the line you think you are on.

We moved all the way back to the Beginning of the editor buffer with the command B and, just as with the other commands, we can reverse this and enter -B to move all the way in the opposite direction. In this case the opposite of beginning is ending, and -B will move us past the last line of text to the end of the buffer, which is the logical place to start adding stuff.

Since CPMIO has more lines than will fit on the screen of your CON:, you will have to skip back and forth a lot to review the program. When you think it is complete, you can view successive screenfuls of text by going back to the beginning and "typing" 22 lines at a time: B22T. Then 22L22T will display the next screenful. The 22 is easy to type, and works fine for a 24 line screen. Just make sure you type 22L and not 22K each time.

A dangerous editor command is nK for Kill n lines. You will have to use this command later when you start adding more subroutines to CPMIO and will want to replace the first test program with another. To do the replacement, you will move the buffer pointer to the beginning of the line labeled START1: and enter 5K to kill the five lines starting at the buffer pointer position. Then the I command will put you in the insert mode so you can type in a new test program.

When you are ready for that exercise, you will want to use the F for Find command. After loading in CPMIO (#A) you could enter FSTART1cr and the editor would find the first occurrence of the string START1 and leave the pointer at the next character following the string. That is not where you want to be! Try FSTART1ctrlz0LT instead. Now ED will find the desired string (terminated by CTRL Z), rewind the pointer to the beginning of the line, and type the line so you can make sure it is the right one. Remember, ED will do what you say, not always what you want.

For example, suppose you wanted to get from the beginning of the text to the subroutine CO. If you had entered the command FCOctrlz0LT you would see the line:

RCONF EQU   1   ; READ CON: INTO (A)

because the first occurrence of the string "CO" appears in that line, as does the second. To keep from getting lost in words containing CO, you could have specified FCO:ctrlz0LT and you would probably get to where you wanted to be. If you hadn't written any

labels like OUTCO: in the meantime. Now you should be able to appreciate the desirability of always appending the ctrlz0LT to all your Find commands.

Finding your way around large programs is made easier by appending the colon to all program labels. Some assemblers require the colon, so you should always use it to make your programs portable. The CP/M assembler is happy with or without the colon. Use it to make ED's Find command just as happy.

The combination of labels with colons and the Find command is a safer way to skip around in a program listing than the use of numbered lines. This is why we disabled line numbering with the

TABLE 12–1. While the CP/M editor ED also provides a number of more powerful text editing features, the subset of commands and controls listed here will be enough to permit entering, correcting, merging, and splitting up assembler source files.

Keystroke(s)	Resulting Action
#A	Append complete file onto editor buffer
I	Enter insert mode
CTRL Z	Exit insert mode
B	Point to beginning of editor buffer
-B	Point to end of editor buffer
V	Turn on automatic line numbering
-V	Turn off automatic line numbering
E	Write editor buffer to disk and exit
Q	Exit ED without writing buffer to disk
±nL	Move back (−) or ahead (+) n lines
nT	Type n lines on console
nK	Kill n lines following current pointer
Fstring	Find the first occurrence of "string" following current pointer position
Sold ↑ Znew ↑ Z	Substitute "new" string for "old"
CTRL I	Tab forward to next tab stop (each 8 characters)
RUBOUT	Erase and echo last character

"-V" command in the last chapter. Line numbers are dangerous—they change with every insertion and deletion. Find your way around by labels. They don't change on you.

When you find the text you want to modify, you can kill lines and insert, or substitute one string for another. There are lots of other ED commands, but those discussed in these two chapters and listed in Table 12-1 are all you will need for most of your assembly language work. These form a completely usable subset of what is available, and a set of commands that can do everything you will ever want an editor to do. So get busy and key in CPMIO, so we can assemble and test it and get on to the more fun kind of programs.

Testing CPMIO

When you have your CPMIO.ASM file edited so that it is identical to List. 12-1, assemble it and load it and make sure that you get the same addresses specified as in List. 12-2. If your program is the same as the original, it should have start address, end address, and size (016D) identical to the example in this book. If not, compare your CPMIO.PRN file with List. 12-3. Here you can compare memory addresses and the assembled code, and by spotting differences between what you got and what is in the book you should be able to find your errors. Finally, your .HEX file should agree with List. 12-4. If all agrees, run the test program.

LISTING 12–2. The console display resulting from assembling and loading CPMIO.

```
ASM CPMIO
CP/M ASSEMBLER - VER 1.4
016D
000H USE FACTOR
END OF ASSEMBLY
LOAD CPMIO
FIRST ADDRESS 0100
LAST  ADDRESS 016C
BYTES READ    002D
RECORDS WRITTEN 01
```

LISTING 12–3. CPMIO.PRN

```
                      ; CP/M I/O SUBROUTINES   30 JULY 82

                      ; ASCII CHARACTERS
000D =                CR      EQU     0DH             ; CARRIAGE RETURN
000A =                LF      EQU     0AH             ; LINE FEED
001A =                CTRLZ   EQU     1AH             ; OPERATOR INTERRUPT

                      ; CP/M BDOS FUNCTIONS
0001 =                RCONF   EQU     1               ; READ CON: INTO (A)
0002 =                WCONF   EQU     2               ; WRITE (A) TO CON:

                      ; CP/M ADDRESSES
0000 =                RBOOT   EQU     0               ; RE-BOOT CP/M SYSTEM
0005 =                BDOS    EQU     5               ; SYSTEM CALL ENTRY
0100 =                TPA     EQU     100H            ; TRANSIENT PROGRAM AREA

0100                          ORG     TPA             ; ASSEMBLE PROGRAM FOR TPA

0100 316C01           START:  LXI     SP,STAK         ; SET UP USER'S STACK
0103 CD1101           START1: CALL    CI              ; INPUT A CONSOLE CHARACTER
0106 FE1A                     CPI     CTRLZ           ; OPERATOR INTERRUPT?
0108 CA0000                   JZ      RBOOT           ; YES, RETURN TO CP/M
010B CD1F01                   CALL    CO              ; NO, ECHO IT AND
010E C30301                   JMP     START1          ;   LOOP

                      ; CONSOLE CHARACTER INTO REGISTER A MASKED TO 7 BITS
0111 C5               CI:     PUSH    B               ; SAVE REGISTERS
0112 D5                       PUSH    D
0113 E5                       PUSH    H
0114 0E01                     MVI     C,RCONF         ; READ FUNCTION
0116 CD0500                    CALL    BDOS
0119 E67F                     ANI     7FH             ; MASK TO 7 BITS
011B E1                       POP     H               ; RESTORE REGISTERS
011C D1                       POP     D
011D C1                       POP     B
011E C9                       RET

                      ; CHARACTER IN REGISTER A OUTPUT TO CONSOLE
011F C5               CO:     PUSH    B               ; SAVE REGISTERS
0120 D5                       PUSH    D
0121 E5                       PUSH    H
0122 0E02                     MVI     C,WCONF         ; SELECT FUNCTION
0124 5F                       MOV     E,A             ; CHARACTER TO E
0125 CD0500                   CALL    BDOS            ; OUTPUT BY CP/M
0128 E1                       POP     H               ; RESTORE REGISTERS
0129 D1                       POP     D
012A C1                       POP     B
012B C9                       RET

                      ; SET UP STACK SPACE
012C                          DS      64              ; 40H LOCATIONS
016C 00               STAK:   DB      0               ; TOP OF STACK

016D                          END
```

LISTING 12–4. CPMIO.HEX

```
:10010000316C01CD1101FE1ACA0000CD1F01C303DD
:10011000001C5D5E50E01CD0500E67FE1D1C1C9C518
:0C012000D5E50E025FCD0500E1D1C1C99C
:01016C000092
:0000000000
```

Not much will happen after you enter CPMIOcr to load and execute the test program. It is waiting for you to type something on the CON:. Each character typed will be tested to see if it is CTRL Z, and if not it will be echoed on the screen. While the results of your typing are not very meaningful, at least you can be assured that CI: and CO: are working, and that is the purpose of the test.

When you get the test program running, you might try pressing a few control keys on your keyboard to see what effect they have. Try CTRL M, CTRL J, CTRL L, and CTRL K, for example. CTRL Z will of course exit the test program and reboot CP/M. And put an end to the stuttering.

CI: and CO: input and output one character at a time. With these two subroutines all checked out, we can move on to buffered I/O, and input and output a whole line at a time. The character-at-a-time subroutines will be used by the line-at-a-time subroutines, so make sure they are working before going on to the next chapter.

There you will be learning about debugging programs by setting traps, so that the computer can't get away from you in case you make a tiny mistake in your program. Remember, a computer will always do what you said, not what you meant.

Buffered Input/Output

The CPMIO.ASM file created in the last chapter includes only single-character-at-a-time input/output subroutines. While these subroutines could be called by a main program to get one character from the console keyboard, or display one character on the screen, there will be very few times in your programs where such limited I/O accesses will be practical. You will almost always want to display a whole message, or input a complete line of text from the console.

Your goal for this chapter is to enter, assemble, and test subroutines that will transfer a line at a time between programs and the computer operator. To provide separation of those lines on the screen, we also throw in a subroutine to output a carriage return and line feed to the console. These three new routines will be added to the CPMIO.ASM file that you have already created.

Saving old files

Since the first installment of CPMIO.ASM has been completely debugged already, a copy of it should be saved before you proceed

to update it. It is always possible for a power line glitch or dust on a disk to cause a total disaster to befall the version you are working on, so a backup copy should be saved of each program upgrade, as soon as it has been fully debugged.

The .BAK files created by ED are good only as temporary backup during editing sessions. You will periodically erase all of them (ERA *.BAK) to save disk space. A completely separate back-up file should be created, preferably on another disk altogether, with a FILENAME.TYP that can't be confused with ED's .BAK files.

Assuming you are working on drive A: and have a mostly empty disk in drive B: for saving backup files, enter:

PIP B:CPMIO.C12=CPMIO.ASM

and a copy of CPMIO will be created on the disk in drive B that is instantly identifiable as the version that resulted from the exercises in Chap. 12. We have used the file type field to create a distinctive tag for this backup file. If you have to use it as an assembler source file, it will have to be RENamed as a .ASM file first. Or better yet, copied back to a working disk in drive A as a .ASM file, preserving the .C12 file as permanent backup:

PIP A:CPMIO.ASM=B:CPMIO.C12

You should periodically use this technique to make copies of all your work files on a disk dedicated to this purpose. Store it away from your working disks, and you will never be caught without a means to recover last week's valuable output.

Library files

With your backup file stored away, you are ready to upgrade CPMIO to include the new line-oriented subroutines. This will entail adding text to the existing source file, and updating the test program to exercise the new routines. Since most of this effort consists of adding to the original file, it will speed things up if you enter the new text into library (.LIB) files rather than directly into the original source file. This permits you to work with smaller files while keying in and proofreading the new text. When they have

been entered and proofread, the new files can be merged into the original file, and the whole thing assembled in one piece.

ED facilitates this procedure by recognizing .LIB files as text to be inserted into the editor buffer at the current buffer pointer position in response to the "R" for Read command. The format of this command is RFILENAME and a carriage return. ED will search the disk directory for FILENAME.LIB, and if found the file will be inserted into the editor buffer in RAM. The original .ASM file on the disk will not be changed at this time, and the .LIB file will still remain on the disk. The next "E" command will cause the merged files to be saved on the disk.

So, now it is time to exercise this technique. Use ED to create and edit a file with the new subroutines shown in List. 13-1. Name this file CH13.LIB.

When you have entered the text of these three subroutines, edit another .LIB file containing the test program (List. 13-2) to exercise them. Since the test program will be inserted into CPMIO.ASM at a different location than the new subroutines, it should be in its own library file.

Also, you might want to retain a library of .LIB files so that only the routines required for a particular program need be merged into the program source file. None of the test programs will be used after the subroutines are debugged, so their files can be ERAsed when your debugging is done. The subroutines developed in this book are nicely grouped into related sets, so their files are a logical set to retain as a library.

The test program will exercise all of the routines developed in this chapter, so name it TESTC13.LIB. When you have keyed it in also, we will discuss the merging process, and explain how the subroutines and the test program work.

When you have CPMIO.ASM, CH13.LIB, and TESTC13.LIB all ready to go on the disk in drive A, enter:

ED CPMIO.ASM

and respond to the ED prompt with #A to read in the "main" program. Start off right by updating the date, so that a glance at the top line will let anyone know that this is the latest version of CPMIO.

Next you want to delete the old test program. The first line of

LISTING 13–1. Library file CH13.LIB.

```
; CARRIAGE RETURN, LINE FEED TO CONSOLE
CCRLF:  MVI     A,CR
        CALL    CO
        MVI     A,LF
        JMP     CO

; MESSAGE POINTED TO BY HL OUT TO CONSOLE
COMSG:  MOV     A,M             ; GET A CHARACTER
        ORA     A               ; ZERO IS THE TERMINATOR
        RZ                      ; RETURN ON ZERO
        CALL    CO              ; ELSE OUTPUT THE CHARACTER
        INX     H               ; POINT TO THE NEXT ONE
        JMP     COMSG           ; AND CONTINUE

; INPUT CONSOLE MESSAGE INTO BUFFER
CIMSG:  PUSH    B               ; SAVE REGISTERS
        PUSH    D
        PUSH    H
        LXI     H,INBUF+1       ; ZERO CHARACTER COUNTER
        MVI     M,0
        DCX     H               ; SET MAXIMUM LINE LENGTH
        MVI     M,80
        XCHG                    ; INBUF POINTER TO DE REGISTERS
        MVI     C,RBUFF         ; SET UP PEAD BUFFER FUNCTION
        CALL    BDOS            ; INPUT A LINE
        LXI     H,INBUF+1       ; GET CHARACTER COUNTER
        MOV     E,M             ; INTO LSB OF DE REGISTER PAIR
        MVI     D,0             ; ZERO MSB
        DAD     D               ; ADD LENGTH TO START
        INX     H               ; PLUS ONE POINTS TO END
        MVI     M,0             ; INSERT TERMINATOR AT END
        POP     H               ; RESTORE ALL REGISTERS
        POP     D
        POP     B
        RET

INBUF:  DS      83              ; LINE INPUT BUFFER
```

LISTING 13–2. Library file TESTC13.LIB.

```
START1: CALL    CCRLF           ; START A NEW LINE
        LXI     H,SINON         ; WITH SIGN-ON MESSAGE
        CALL    COMSG
START2: CALL    CIMSG           ; GET A LINE OF INPUT
        CALL    CCRLF
        LXI     H,INBUF+2       ; POINT TO ITS TEXT
        CALL    COMSG           ; ECHO THE WHOLE LINE
        CALL    CCRLF           ; AND CR, LF
        JMP     START2          ; THEN DO ANOTHER

SINON:  DB      'SIGN-ON MESSAGE',CR,LF,0
```

each of the test programs you will be using sets up the stack point-er, and will not change from one program to another. To get rid of the rest of the test program, find it with a command of:

FSTART1:ctrlzOLT

and verify that you are on the right line. Follow this with a com-mand of 5Kcr to kill the old five line program. This leaves the editor buffer pointer at the beginning of the line just below START: and this is where you want to insert TESTC13.LIB.

Simply enter RTESTC13 and a carriage return and you should hear your disk drive begin the search for the library file. ED will not let you know if the search was successful, only if the file was not found or was not read properly. When the merge is complete you will just get another "*" prompt on the screen.

Skip down in the text to the line below the RET at the end of the old subroutine CO:. A 0LT at this point should show you that the buffer pointer is at the beginning of the comment line "; SET UP STACK SPACE." This is where you want to enter the com-mand RCH13 to merge in the new subroutines.

This text could have been merged in anywhere following the test program, but you should put it exactly where shown in the overall listing of the merged file (as assembled), at the end of this chapter (List. 13-4). That way your object code will be identical to that in this listing, and any errors in your program can be easily found.

Now rewind the text buffer pointer to the beginning and examine the entire file, comparing it with List. 13-4, which of course includes the assembler output in the left hand columns that won't appear in your source listing. The first discrepancy you will notice is the absence (in your version) of the definition of a value for RBUFF. This is a one line insertion you should now make, defin-ing the code for another BDOS function.

If you have faithfully copied the programs as listed in this book, your files will have strategically placed blank lines between blocks of related code. Studies on programmer productivity in debugging unfamiliar programs show that blank lines in logical locations greatly speeds up the debugging process by breaking up listings into logical blocks. You should adapt this practice from the start. You might find, however, that the file merging process has

resulted in missing or misplaced blank lines. Now is the time to correct this.

With your source program all correct, you should be able to assemble it error free, producing the same CPMIO.PRN file as in List. 13-4. Before you try running this update, you should know something about how the subroutines work, and what the test program is doing.

CCRLF: starts a new line

By now you should have no trouble figuring out what this one does. It simply uses subroutine CO: to start a new line on the console by outputting the code for carriage return followed by the code for line feed to the CON: device. There is one byte-saving technique included in this subroutine that you haven't seen before, though.

A subroutine usually ends with a RETurn instruction. This opcode takes the calling program reentry address off the top of the stack and puts it into the program counter, effecting a return to the calling program. CCRLF: could have ended with CALL CO followed by RET, but the JMP CO accomplishes the same thing, executes faster, and saves one byte of object code.

The program that CALLed CCRLF: left its return address on the stack. The first CALL CO in CCRLF: pushed the address of the next instruction onto the stack, and CO: returns to CCRLF: at MVI A,LF by popping that address off the stack. The original calling program return address is now back on the top of the stack. If CCRLF: ends with a jump to CO:, the main program calling address will still be on the top of the stack and will be the one popped off by the RET at the end of CO:. The main program will then be reentered after CO: outputs the line feed. And this is exactly what we want.

Even more program space and execution time could be saved by locating the code for CCRLF: immediately above that of CO: in the source program, and leaving off the JMP CO altogether. Then CCRLF: would call CO: to output the CR, and then load the accumulator with the LF and just "fall through" into CO:, which would finally return to the original calling program.

We have not used this trick here, however, because we will be doing a lot of editing and merging in the future, and if any other

routine got inserted between the end of CCRLF: and the start of CO: in the process, the fall-through would be to the wrong place! Since we have lots of memory in our CP/M computer anyway, we won't take advantage of this technique here, but you should know about it as you will run across it in other programs that you might be studying for inspiration and ideas in the future. Studying other programmers' codes is, by the way, another good way for the beginner to learn. Finish this book first, though.

COMSG: displays a line of text

Up to now, all of our programs have relied on CO: to save and restore the user's registers while writing to the console. Up to now, only the A register has been used by the calling programs, so all that saving didn't accomplish much. To implement the display of a line of text, however, we will now need to use an index register to point into successive locations in the text buffer in memory, and it will have to be saved during the execution of CO: and all the action that takes place within CP/M as a result of the BDOS call. In this subroutine, we use "the" index register, the H and L pair.

In the test program file (List. 13-2) we created a text buffer containing a sign-on message. All of your programs should start with a sign-on, so the user knows that the right program got loaded into the TPA. The example message doesn't convey much information. It is assumed that all of yours will. To output this message to the console, we point to the first character with the index register. In the test program, this is accomplished by the LXI H, SINON instruction. With the index initialized, a CALL to COMSG: will cause the text to be displayed on the CON:.

We know that the HL register pair points to the "M register" in memory. In this case, no arithmetical or logical operations are to be performed; the M register is merely the source of one ASCII character. MOV A,M moves that character from the buffer into the accumulator, where it is tested to see if it is the buffer terminator character. Any character that you know in advance would never be used as a character in a text line can serve as the terminator, but our choice is a zero byte. Since 00H is not a printable ASCII character, it will never exist in any text. Also, all zeros in a byte is

easy to test for, with one of the flag register bits set aside to store the test result.

When we MOVed the byte from the buffer to the A register, the MOV instruction had no effect on the flag register (see Appendix B). To test for zero, we execute a dummy instruction. This time we use the logical OR instruction, and OR the contents of the A register with itself.

The Boolean rules for OR are:

1. Zero OR zero is zero
2. Zero OR one is one
3. One OR zero is one
4. One OR one is one

In other words, for any bit position in the byte, if either operand contains a one bit, the result contains a one bit. Only zero, zero results in a zero. When the accumulator contents is ORed with itself, nothing changes, but the flag register records the result of the logical operation.

If the fetched character was the zero terminator, the conditional return instruction, Return on Zero, ends COMSG: and reenters the calling program. Otherwise, the character is output to the console by the call to CO:. The following instruction increments the index to point to the next message buffer character, and the jump back to COMSG: restarts the process.

Here we have a subroutine that ends its operation with a conditional return instruction, and the return is embedded within the code so is not as obvious as the RETurns at the end of CO: and CI:. This is another reason for the insertion of a blank line following each logical block of code in a source program. The blank lines before and after this subroutine define it as an entity, which would not otherwise be so obvious with the embedded return instruction.

The message text that is operated on by COMSG: was stored in a buffer with a start address labeled SINON:. The text itself, in the form of ASCII characters stored in memory, was formed by the assembler in response to the pseudo operation DB, for "**D**efine **B**yte(s)." An operand consisting of text surrounded by single quotes tells the assembler that we want the ASCII code equiv-

alents for the characters to be stored at the current program counter location. Other byte values have been added onto this text list, separated by commas. CR and LF are symbols whose values were defined by the EQUate instructions at the beginning of the program. The assembler will insert the defined values into the buffer. The zero is an absolute value, but could have been symbolically defined as were CR and LF.

Including CR and LF within the text buffer is the equivalent of following CALL COMSG in the test program with another statement, CALL CCRLF, but saves one byte over the CALL, as well as one line in the source program. As is typical in computer programming, there is often more than one way to accomplish a desired action.

Note that the assembled .PRN listing shows only the first few bytes of the message text in the object code column. You can see that all of the text is actually in the object code buffer, however, by looking at the next address in the listing. It is more than a few bytes greater than the start address of the text. ASM simplifies the .PRN listing.

CIMSG: gets a line from the operator

The console message output subroutine transferred a text line unchanged from buffer to console display. Since human computer operators are more prone to error than are computers, transferring a line in the other direction should include provisions for letting the operator change his mind in the middle of a line. CP/M provides a ready made line-input-with-editing facility that saves the user the trouble of writing the equivalent function into each user program.

In Chap. 4 we discussed these line editing features, as they appear to the computer operator typing command lines into CCP. Characters struck in error can be deleted with the DEL or RUBOUT key. CTRL U or CTRL X aborts a whole line and lets you start over. CTRL C aborts the currently running program and reboots CP/M. CTRL R redisplays the line with rubouts cleaned up. It would take a lot of code in users' programs to provide the same features. CP/M provides them automatically through RBUFF, the **R**ead **BUF**fered input **F**unction call to BDOS.

There is some user program overhead required before this
function can be taken advantage of. An input buffer must be prop-
erly formatted in advance, and pointed to by the wrong index,
before calling BDOS with the function code in the C register (the
"right" index would be HL, not DE). The maximum number of
input characters permitted must be specified by a value that is
prestored in the buffer. The line as input after editing will not have
our handy zero byte terminator at the end, but instead a count of
the net characters input will be stored in the buffer along with the
text.

The advantages of using this function far outweigh the cost of
the few lines of code required to implement the function call. And,
in keeping with our philosophy of preserving a friendly user en-
vironment, all of these overhead requirements are provided within
subroutine CIMSG:. The user program simply calls CIMSG:, and
upon return finds a one-line message from the operator stored in
INBUF:, terminated by a zero byte.

Like CI: and CO:, CIMSG: is a lowest-level user subroutine
because it includes a BDOS call. It therefore includes the register
save and restore instructions required for preserving the user's
environment. After saving registers, the subroutine sets up the
first two locations in the input buffer with a maximum character
count (80 is standard on CRT terminals) and an initial character
count of zero. Note that we store the second value first, at IN-
BUF+1, then decrement the index and store the first value. This
saves a few bytes of code, since we have to pass the start address of
INBUF to BDOS, and after initializing the first two locations in
reverse order, the HL pair points to that start address.

Since BDOS wants address values passed in the DE pair, but
our MVI M, instructions used the HL pair, the single-byte register
exchange instruction (XCHG) is used to swap the address value
from HL to DE. This opcode also puts the old values originally in
DE into HL. The function code is then placed in C, and BDOS is
called.

If the computer operator enters CTRL C as the first character
typed following the BDOS call, the calling program will be aborted
and CP/M rebooted. Otherwise characters typed are stored in IN-
BUF, beginning at the third byte, up to the maximum count or an
operator typed carriage return. The line editing features are in
operation as the operator types, and only the line as edited will

remain in INBUF. This is the input line as it would be displayed by an operator entry of CTRL R immediately preceding the CR.

When BDOS returns to CIMSG: with the edited line in IN-BUF, the second INBUF location (INBUF+1) contains the count of the characters to be found in the buffer. The first location still contains the unchanged maximum character count. The text actually starts at INBUF+2, then.

To make this buffer compatible with our text-terminated-by-zero convention, CIMSG: takes the count value from where BDOS left it in INBUF+1, and places it into the E register. Zeroing the D register produces a 16-bit value that, when added to the address of INBUF+2, produces an index pointing to the end of the text in the buffer (start + length = end). Here we store a zero byte terminator. Once again saving a couple of instructions, the program does this by adding the count to INBUF+1, since this address is already in HL, and then incrementing the index to point to the next location. DAD D is a **D**ouble byte (16 bit) **AD**d **DE** register to HL.

Note that once we have terminated the text in the buffer with a zero byte, there is no more need for a character counter. The same is true for an output message buffer. This zero terminator convention is easier to work with than techniques requiring character counters.

Testing the subroutines

Our new test program starts out by displaying the sign-on message. The index is initialized to point to the start of the message buffer, COMSG: is called, and the message is displayed followed by the CR and LF that were included in its buffer.

At START2: we enter an endless loop. CIMSG: is called to input an operator message. Type in any text you want, terminated by a carriage return. This will cause a return to the CALL CCRLF instruction following START2:, which will begin a new line on the screen.

The test program then initializes the index to point to the beginning character of the input text; the third location in the input buffer. COMSG: is then called, and the line typed in is displayed a second time on the console. Another new line is started, and the operator can type some more.

Debugging with DDT

Since the object code that was generated by the CP/M assembler for CPMIO while this book was being prepared is shown in List. 13-4, and you can compare your own assembler output against it, there is little chance that any errors in your programs can't be found by comparison. This will not always be the case, obviously, and it is time to get familiar with the most commonly used features of DDT, the CP/M Dynamic Debugging Tool.

The first step in debugging any new software is to get DDT and the new program loaded into the computer memory, and then remove all the disks from the drives. Since a new assembly language program can have errors in it that could cause the program counter to get totally lost, it is always possible for such an error to cause a jump into disk write routines in CP/M or in your monitor PROM. So, to begin debugging, enter:

```
DDT CPMIO.COM
```

and you will see:

```
DDT VERS x.x
NEXT PC
0280 0100
-
```

indicating that DDT has been read into memory, and has loaded a program that starts at 100H and extends up to 27FH. Now, remove all the disks from the drives before proceeding!

You might have noticed that the highest address in CPMIO is the location of the STAK: at 211H, but DDT thinks the program ends just below 280H. This is because each disk record is 128 bytes, or 80 bytes in hexadecimal, and CPMIO extends into the third record on the disk. DDT can't know that the program ends at 213H, even though ASM and LOAD knew it, and so assumes the top of the program is the end of the last record allocated to it, at 27FH.

Since DDT is a .COM file on your disk, as is your test program CPMIO.COM, it is obvious that both can't occupy the transient program area at the same time. DDT is loaded into the TPA from the disk, but then relocates itself up in memory, overlaying

147

CCP, and freeing the TPA workspace. When relocated, DDT changes the address in the BDOS call vector at location 5, and establishes its own breakpoint vector at location 38H. DDT then loads your program into the TPA, and signs on with the "-" prompt.

DDT changes the BDOS call vector to trap any calls for I/O access that your program might make, in case you have requested DDT to trace each step in execution. As you will see shortly, tracing takes up a lot of CPU execution time, so DDT turns tracing off during the times your program uses CP/M facilities, which are already debugged.

DDT also establishes a vector to itself at location 38H, which is one of the 8080 family interrupt vectors. DDT uses the software interrupt, RST 7 (opcode = 0FFH) for breakpointing your program during test. When you set breakpoints by specifying an address at which you want program execution to stop, DDT plugs 0FFH into your program at that address. When encountered, this opcode forces a software interrupt, jumping to DDT through the RST 7 vector. DDT can then save and display the contents of the CPU registers, so that you can see what went wrong in your program.

Since DDT uses one of the interrupt vectors, it is obvious that your hardware and your programs cannot use the same vector. If DDT is executing properly on your system, you can assume that other programs in your system don't use this vector.

Breakpointing and tracing are the most powerful and easiest to use capabilities of DDT, and along with memory dump and register examination are the only DDT commands you will be needing at your current level of expertise. Later you will want to study the DDT manual to get familiar with all of the other things this program can do for you.

Refer to the test program in List. 13-4. To use DDT to test execution of this program in a step-by-step fashion, you would first enter the command:

G100,103

followed by a carriage return. This tells DDT that you want to **Go** to location **100**H, and execute your program up to (but not including) the instruction at location **103**H. DDT will plug that RST 7

opcode into 103H, and jump to 100H. Your program will load the immediate value 0211H into the stack pointer, and then fetch the opcode from location 103H. Instead of what you originally wrote, this opcode will be the restart opcode that DDT inserted into your program. DDT will be entered through its breakpoint vector, and will inform you that it has done so by displaying:

*103

and prompting for another command. Here enter:

XScr

and DDT will respond to your request to eXamine register **S** (the stack pointer) with:

S=0211

confirming that you did indeed load the stack pointer with the desired top-of-stack address. You can then test the execution of the next step also by entering a command of:

G100,106cr

and you should see the CR and LF action on your console before the breakpoint returns you to DDT. Note that in debugging a main program in this manner, all previously tested parts should be reexecuted at each breakpointed step. Don't test the next step with a G103,106 because then your stack pointer may not be properly initialized.

At each breakpoint step you can call for the display of any or all register contents, as well as the contents of memory. A faster way to test a program is by using the trace command. During traced execution, each instruction executed will be displayed along with all of the register contents in hexadecimal.

Refer to List. 13-3 and 13-4 to follow the steps involved in using DDT's trace function to check out subroutine CIMSG:. The first listing shows the console display during the session. The operator inputs are underlined.

LISTING 13–3. Console displays during the debugging of subroutine CIMSG: using DDT.

```
DDT CPMIO.COM

DDT VERS 1.4
NEXT   PC
0280 0100
-XP

P=0100 15F

-T1A

C0Z0M0E0I0 A=00 B=0000 D=0000 H=0000 S=0100 P=015F PUSH B
C0Z0M0E0I0 A=00 B=0000 D=0000 H=0000 S=00FE P=0160 PUSH D
C0Z0M0E0I0 A=00 B=0000 D=0000 H=0000 S=00FC P=0161 PUSH H
C0Z0M0E0I0 A=00 B=0000 D=0000 H=0000 S=00FA P=0162 LXI  H,017F
C0Z0M0E0I0 A=00 B=0000 D=0000 H=017F S=00FA P=0165 MVI  M,00
C0Z0M0E0I0 A=00 B=0000 D=0000 H=017F S=00FA P=0167 DCX  H
C0Z0M0E0I0 A=00 B=0000 D=0000 H=017E S=00FA P=0168 MVI  M,50
C0Z0M0E0I0 A=00 B=0000 D=0000 H=017E S=00FA P=016A XCHG
C0Z0M0E0I0 A=00 B=0000 D=017E H=0000 S=00FA P=016B MVI  C,0A
C0Z0M0E0I0 A=00 B=000A D=017E H=0000 S=00FA P=016D CALL 0005
C0Z0M0E0I0 A=00 B=000A D=017E H=0000 S=00F8 P=0005 JMP  9100
C0Z0M0E0I0 A=00 B=000A D=017E H=0000 S=00F8 P=9100 JMP  97A2
C0Z0M0E0I0 A=00 B=000A D=017E H=0000 S=00F8 P=97A2 XTHL
C0Z0M0E0I0 A=00 B=000A D=017E H=0170 S=00F8 P=97A3 SHLD A044
C0Z0M0E0I0 A=00 B=000A D=017E H=0170 S=00F8 P=97A6 XTHL
C0Z0M0E0I0 A=00 B=000A D=017E H=0000 S=00F8 P=97A7 JMP  A106

C0Z1M0E1I0 A=00 B=000D D=0000 H=0000 S=00FA P=0170 LXI  H,017F
C0Z1M0E1I0 A=00 B=000D D=0000 H=017F S=00FA P=0173 MOV  E,M
C0Z1M0E1I0 A=00 B=000D D=0008 H=017F S=00FA P=0174 MVI  D,00
C0Z1M0E1I0 A=00 B=000D D=0008 H=017F S=00FA P=0176 DAD  D
C0Z1M0E1I0 A=00 B=000D D=0008 H=0187 S=00FA P=0177 INX  H
C0Z1M0E1I0 A=00 B=000D D=0008 H=0188 S=00FA P=0178 MVI  M,00
C0Z1M0E1I0 A=00 B=000D D=0008 H=0188 S=00FA P=017A POP  H
C0Z1M0E1I0 A=00 B=000D D=0008 H=0000 S=00FC P=017B POP  D
C0Z1M0E1I0 A=00 B=000D D=0000 H=0000 S=00FE P=017C POP  B
C0Z1M0E1I0 A=00 B=0000 D=0000 H=0000 S=0100 P=017D RET  *1331
-D17E,188

017E 50 08 P.
0180 48 4F 57 20 4E 4F 57 3F 00 HOW NOW?.
```

When DDT is asked to load in CPMIO.COM, the program assumes that debugging will begin at location 100H, which is usually the case. DDT therefore initially establishes a program counter (PC) content of 100H. To guard against any programmer oversights, DDT also establishes a default stack pointer setting of 100H.

In this example we want to check out only CIMSG:, so the first operator action is to request the examination of the program counter, abbreviated "P." The "X" calls up the register eXamine routine, and DDT shows the current content is 100H. The operator overrides this by entering the start address of the subroutine, 15FH. All addresses and data values are in hexadecimal.

When DDT accepts this change and reprompts, the operator keys in a command to Trace **1A** program steps. Hexadecimal 1A is 26 in decimal, and this number of steps gets us through the subroutine and stops at the return. Note that in tracing subroutines, disaster might result if you were to trace past the return, since DDT did not execute any part of a calling program. Find the exact number of trace steps you want to display by sneaking up on the return a few steps at a time.

The display that results from the trace command shows you each program step as executed, and displays all the register contents that result from the execution. This display includes the flag register, and we start off with the **C**arry flag, **Z**ero flag, **M**inus flag, . . . etc. all initialized to zero. Note that DDT does not use the Intel standard flag bit designations (Fig. 9-2), so Table 13-1 lists both sets of abbreviations and the meanings of the flag bits.

The first subroutine step pushes the contents of the B and C registers onto the stack, decrementing the stack pointer two times in the process. Since the execution of PUSH starts by decrementing SP before the contents of C is moved to memory at the location pointed to by SP, the first location written to will be 0FFH, and not 100H. Otherwise the first opcode in our program would be overwritten.

Notice that each of the first three PUSHes decrements the SP by two, and it ends up at 0FAH after the register saves. The contents of the registers themselves do not change until the first load index instruction (LXI) places an address into the HL pair. You can easily follow the action of each step in the subroutine as registers are initialized prior to the BDOS call, "CALL 0005" as traced.

The Digital Research DDT manual states that tracing is turned off during BDOS execution steps, but you can see from this example that the turn-off didn't happen until after six instructions were executed. Ignore these steps, they are obviously not part of our program.

Tracing halts where the blank line shows in the listing. This is

TABLE 13-1. The flag (F) register functions within microprocessors do not change, but the terms used to refer to them are not always the same. Here the differences between Intel conventions and those followed by CP/M's Dynamic Debugging Tool are shown, along with the function of each F register bit.

Flag Bit	Intel Name	CP/M DDT Name	Function
7	S (Sign)	M (Minus)	Duplicates accumulator bit 7 after ALU operations.
6	Z (Zero)	Z (Zero)	Set = 1 if accumulator results in all bits zero.
5		Unused in 8080	
4	A (Auxiliary carry)	I (Interdigit carry)	Carry out of bit 3 for packed decimal arithmetic.
3		Unused in 8080	
2	P (Parity)	E (Even parity)	Set = 1 if accumulator results in even number of one bits.
1		Unused in 8080	
0	C (Carry)	C (Carry)	Set = 1 if accumulator result is greater than 255 (0FF in hexadecimal)

because we have called BDOS for the input of an operator message, and you, the operator, now have to type a line terminated by a carriage return. In this example, the operator typed "HOW NOW?" and CR.

The CR signals the end of input. The CIMSG: subroutine is returned to and the index is set to point to 017FH, wherein lies the count of characters in the input buffer. This count is moved into the E register, D is zeroed, and DE is added to HL. One increment points HL past the text in the buffer, a zero is written to memory to terminate the text, and the registers are restored to their initial contents. The RET restores the stack pointer to its original value, an action that all subroutines must accomplish.

Don't forget that PUSHes and POPs must be balanced within a subroutine.

Finally, the **D**isplay memory command is used to dump the contents of INBUF, and both the hexadecimal codes for the ASCII characters and the characters themselves are displayed.

This example should show you the power of debugging by tracing, but before you start thinking that it is the only way to fly, remember that this subroutine did not include any long loops. Sometimes it is easier to trap a program at the end of a loop, and save tracing for more straightforward execution.

This combination of DDT's debugging tools is another example of the power, flexibility, and friendly environment provided by the CP/M operating system and its utility programs. Well, DDT is supposed to kill bugs, isn't it?

Exercises

The Chap. 13 test program is a little more meaningful than the one in Chap. 12, and produces a nicer display, at least. In addition to exercising the new subroutines to demonstrate proper operation, it gives the operator a chance to experiment with the line editing features provided by CP/M. This line editing should be included in all user programs that prompt for operator input, to allow the operator the chance to correct errors or abort the program and return to CP/M without having to resort to the RESET switch.

You can also experiment with control characters embedded within a text line. On most terminals CTRL L or CTRL Z will produce a cleared screen. This action is filtered out during input of a text line and the control character is displayed by the up-arrow-plus-letter convention. But when the control character is echoed by COMSG: the screen should go blank. Type in:

HI ctrlz THERE cr

and see what happens. If nothing happens, try CTRL L in place of CTRL Z.

If you are getting bored by all of this, and are starting to fall asleep, type in a message to yourself containing a bunch of CTRL Gs.

LISTING 13-4. CPMIO.PRN for Chap. 13.

```
                         ; CP/M I/O SUBROUTINES    16 AUG 82

                         ; ASCII CHARACTERS
000D =                   CR       EQU      0DH              ; CARRIAGE RETURN
000A =                   LF       EQU      0AH              ; LINE FEED
001A =                   CTRLZ    EQU      1AH              ; OPERATOR INTERRUPT

                         ; CP/M BDOS FUNCTIONS
0001 =                   RCONF    EQU      1                ; READ CON: INTO (A)
0002 =                   WCONF    EQU      2                ; WRITE (A) TO CON:
000A =                   RBUFF    EQU      10               ; READ A CONSOLE LINE

                         ; CP/M ADDRESSES
0000 =                   RBOOT    EQU      0                ; RE-BOOT CP/M SYSTEM
0005 =                   BDOS     EQU      5                ; SYSTEM CALL ENTRY
0100 =                   TPA      EQU      100H             ; TRANSIENT PROGRAM AREA

0100                              ORG      TPA              ; ASSEMBLE PROGRAM FOR TPA

0100 311102             START:    LXI      SP,STAK          ; SET UP USER'S STACK
0103 CD4B01             START1:   CALL     CCRLF            ; START A NEW LINE
0106 211E01                       LXI      H,SINON          ; WITH SIGN-ON MESSAGE
0109 CD5501                       CALL     COMSG
010C CD5F01             START2:   CALL     CIMSG            ; GET A LINE OF INPUT
010F CD4B01                       CALL     CCRLF
0112 218001                       LXI      H,INBUF+2        ; POINT TO ITS TEXT
0115 CD5501                       CALL     COMSG            ; ECHO THE WHOLE LINE
0118 CD4B01                       CALL     CCRLF            ; AND CR, LF
011B C30C01                       JMP      START2           ; THEN DO ANOTHER

011E 5349474E2DSINON:   DB       'SIGN-ON MESSAGE',CR,LF,0

                         ; CONSOLE CHARACTER INTO REGISTER A MASKED TO 7 BITS
0130 C5                 CI:       PUSH     B                ; SAVE REGISTERS
0131 D5                           PUSH     D
0132 E5                           PUSH     H
0133 0E01                         MVI      C,RCONF          ; READ FUNCTION
0135 CD0500                       CALL     BDOS
0138 E67F                         ANI      7FH              ; MASK TO 7 BITS
013A E1                           POP      H                ; RESTORE REGISTERS
013B D1                           POP      D
013C C1                           POP      B
013D C9                           RET

                         ; CHARACTER IN REGISTER A OUTPUT TO CONSOLE
013E C5                 CO:       PUSH     B                ; SAVE REGISTERS
013F D5                           PUSH     D
0140 E5                           PUSH     H
0141 0E02                         MVI      C,WCONF          ; SELECT FUNCTION
0143 5F                           MOV      E,A              ; CHARACTER TO E
0144 CD0500                       CALL     BDOS             ; OUTPUT BY CP/M
0147 E1                           POP      H                ; RESTORE REGISTERS
0148 D1                           POP      D
0149 C1                           POP      B
014A C9                           RET
```

LISTING 13-4. *Continued*

```
                    ; CARRIAGE RETURN, LINE FEED TO CONSOLE
014B 3E0D           CCRLF:  MVI     A,CR
014D CD3E01                 CALL    CO
0150 3E0A                   MVI     A,LF
0152 C33E01                 JMP     CO

                    ; MESSAGE POINTED TO BY HL OUT TO CONSOLE
0155 7E             COMSG:  MOV     A,M             ; GET A CHARACTER
0156 B7                     ORA     A               ; ZERO IS THE TERMINATOR
0157 C8                     RZ                      ; RETURN ON ZERO
0158 CD3E01                 CALL    CO              ; ELSE OUTPUT THE CHARACTER
015B 23                     INX     H               ; POINT TO THE NEXT ONE
015C C35501                 JMP     COMSG           ; AND CONTINUE

                    ; INPUT CONSOLE MESSAGE INTO BUFFER
015F C5             CIMSG:  PUSH    B               ; SAVE REGISTERS
0160 D5                     PUSH    D
0161 E5                     PUSH    H
0162 217F01                 LXI     H,INBUF+1       ; ZERO CHARACTER COUNTER
0165 3600                   MVI     M,0
0167 2B                     DCX     H               ; SET MAXIMUM LINE LENGTH
0168 3650                   MVI     M,80
016A EB                     XCHG                    ; INBUF POINTER TO DE REGISTER
016B 0E0A                   MVI     C,RBUFF         ; SET UP READ BUFFER FUNCTION
016D CD0500                 CALL    BDOS            ; INPUT A LINE
0170 217F01                 LXI     H,INBUF+1       ; GET CHARACTER COUNTER
0173 5E                     MOV     E,M             ; INTO LSB OF DE REGISTER PAIR
0174 1600                   MVI     D,0             ; ZERO MSB
0176 19                     DAD     D               ; ADD LENGTH TO START
0177 23                     INX     H               ; PLUS ONE POINTS TO END
0178 3600                   MVI     M,0             ; INSERT TERMINATOR AT END
017A E1                     POP     H               ; RESTORE ALL REGISTERS
017B D1                     POP     D
017C C1                     POP     B
017D C9                     RET

017E                INBUF:  DS      83              ; LINE INPUT BUFFER

                    ; SET UP STACK SPACE
01D1                        DS      64              ; 40H LOCATIONS
0211 00             STAK:   DB      0               ; TOP OF STACK

0212                        END
```

14
Tricky Techniques

Some early very small minicomputers had such a restricted instruction set that it was difficult to find *one* way to accomplish some operations. The Intel 8080 microprocessor has such a relatively rich instruction set that there are, as we have seen, often many ways to accomplish a desired result.

A programmer, under pressure to produce, will usually select the most straightforward coding, even if it is not the solution that is most compact or executes fastest. But when speed or program size are important, it is necessary for you to know how to get things done using a minimum of bytes of code and cycles of CPU execution time.

Since the 8080 is stack-oriented, and most earlier machines were not, older programmers often do not know some of the tricks that are available thanks to the stack. One was mentioned in Chap. 13: letting one subroutine fall through into another. There are lots more, and we will be using some of them to implement three new subroutines. You will find these additions to CPMIO to be great labor saving devices. They should be. They are emulating state-

ments found in higher level languages. That had better make your programming easier!

TWOCR:, a one-line subroutine

Some console messages, especially error messages and warnings, should be displayed with blank lines above and below to set them off from surrounding text. Obviously, you could call CCRLF: twice in a row before and after displaying such messages. But it is easier to add one line of code to CPMIO to implement a double line feed.

Immediately above CCRLF: in your CPMIO.ASM file, add the line:

```
TWOCR: CALL   CCRLF
```

and then your programs can call TWOCR: instead of CCRLF: when you want to double space text on the console. Since this addition is called as a subroutine but has no return of its own, it will execute a call to CCRLF:, producing a carriage return and line feed. Then the return from CCRLF: will be back to the instruction following the call. That instruction is the entry point to CCRLF: so you will get another CR and LF before the original calling program is returned to.

SPMSG: displays in-line messages

In order to display our sign-on message (Chap. 13) we set aside a message text buffer, SINON:, and had to load an index register with its start address before calling our console output message subroutine, COMSG:. There is a better way to display messages, allowing the programmer to include message texts within the flow of programs, instead of setting up separate text buffers.

In the middle of a BASIC language program, you can say:

```
PRINT 'Message text for the console'
```

and the text within the quotes will be displayed for the operator. The text thus follows the instruction PRINT and precedes the next

instruction (or program statement, as it is known in BASIC). The text appears at the point in the program where it will be displayed, rather than in a remote buffer area.

Using SPMSG: you can include a similar function in assembly language programs, without having to set up message text buffers, and without using even one index register! Simply write:

```
CALL   SPMSG
DB     'Message text for the console', 0
```

and SPMSG: will output your text and return to the instruction following the zero byte terminator. This allows you to place console messages within the mainstream of your program, improving program readability and reducing program size and register usage.

Later, we will be looking in detail at how the stack operations make this possible. First let's look at the test program for this chapter and see how our new subroutines are used.

LISTING 14-1. TESTC14.LIB

```
START2: CALL    TWOCR               ; DOUBLE SPACE LINES
        CALL    SPMSG               ; PROMPT FOR TEST
        DB      'TESTING FOR YES OR NO FROM CONSOLE',CR,LF,0
        CALL    GETYN
        JNZ     START3              ; GOT A "NO"
        CALL    SPMSG               ; GOT A "YES"
        DB      LF,'YOUR ANSWER WAS "YES!"',0
        JMP     START2
START3: CALL    SPMSG
        DB      LF,'YOU SAID "NO!"',0
        JMP     START2
```

This time you will not replace all of the old test program within CPMIO.ASM. Leave the first four lines to retain the sign-on message display. Delete the next six lines, START2: through the JMP START2, and then merge in TESTC14.LIB, which you will create from the program lines in List. 14-1.

This new test program first checks out TWOCR:. It then calls our stack-oriented message routine to display the text in the line immediately following. Once again, this text must be terminated by a zero, so that SPMSG: knows when to quit and return to the next opcode.

That opcode calls our final console I/O subroutine, GETYN:.

This routine will prompt the console operator for a yes or no decision, and will return to the calling program with that decision recorded in the zero flag.

GETYN: interrogates the operator

When called, this subroutine will display the short prompt:

(Y/N)?:

and wait for the operator's response. Line editing is in effect for the operator, and so an immediate CTRL C will cause a program abort and a return to CP/M. The only other valid responses are either upper- or lower-case "Y" or "N" as the first character in the line. GETYN: uses the buffered console input subroutine for operator responses, but only examines the first character on the line returned.

A valid response therefore could be "yep" or "YEAH!" or just "y" by itself, with a CR signaling that the operator is ready to have the response accepted. If the console operator answers incorrectly, with neither "y" nor "n" nor CTRL C as the first character on the line, he will be reprompted. GETYN: will return to the calling program with the zero flag set if the answer is yes, or the zero flag not set if the answer is no.

Reading through the test program in List. 14–1 will show you how all of these subroutines are used. GETYN: is called, the response tested by the conditional **J**ump on **N**ot **Z**ero, and one of two messages is displayed showing what the program thinks the operator meant.

In real programs, the single line CALL GETYN followed by a conditional jump on zero or not zero will effect a program response to the operator's wishes. Every time a yes or no answer is needed, the programmer calls SPMSG: to ask the question, and GETYN: to receive the answer.

Now it is time for you to key in the two .LIB files in List. 14–1 and 14–2, and merge them into CPMIO.ASM. To be consistent with the listings in this book, you should merge the subroutines in right before INBUF: in your existing program. Be careful with those nested quotes in the test program!

LISTING 14-2. CH14.LIB

```
; MESSAGE POINTED TO BY STACK OUT TO CONSOLE
SPMSG:  XTHL                        ; GET "RETURN ADDRESS" TO HL
        XRA     A                   ; CLEAR FLAGS AND ACCUMULATOR
        ADD     M                   ; GET ONE MESSAGE CHARACTER
        INX     H                   ; POINT TO NEXT
        XTHL                        ; RESTORE STACK FOR
        RZ                          ; RETURN IF DONE
        CALL    CO                  ; ELSE DISPLAY CHARACTER
        JMP     SPMSG               ; AND DO ANOTHER

; GET YES OR NO FROM CONSOLE
GETYN:  CALL    SPMSG               ; PROMPT FOR INPUT
        DB      ' (Y/N)?: ',0
        CALL    CIMSG               ; GET INPUT LINE
        CALL    CCRLF               ; ECHO CARRIAGE RETURN
        LDA     INBUF+2             ; FIRST CHARACTER ONLY
        ANI     01011111B           ; CONVERT LOWER CASE TO UPPER
        CPI     'Y'                 ; RETURN WITH ZERO = YES
        RZ
        CPI     'N'                 ; NON-ZERO = NO
        JNZ     GETYN               ; ELSE TRY AGAIN
        CPI     0                   ; RESET ZERO FLAG
        RET                         ; AND ALL DONE
```

With the new test program lines and the new subroutines merged into CPMIO.ASM, you should be able to assemble, load and run the test. Your CPMIO.PRN file should match Listings 14-3 and 14-4. Only the portions of CPMIO that have been changed are included in these listings. Did you remember to patch in the one-line TWOCR: subroutine?

LISTING 14-3. A partial listing of CPMIO, showing the new code resulting from including TESTC14.LIB

```
0100 319602     START:  LXI     SP,STAK             ; SET UP USER'S STACK
0103 CDA101     START1: CALL    CCRLF               ; START A NEW LINE
0106 217101             LXI     H,SINON             ; WITH SIGN-ON MESSAGE
0109 CDAB01             CALL    COMSG
010C CD9E01     START2: CALL    TWOCR               ; DOUBLE SPACE LINES
010F CDD401             CALL    SPMSG               ; PROMPT FOR TEST
0112 5445535449         DB      'TESTING FOR YES OR NO FROM CONSOLE',CR,LF,0
0137 CDE001             CALL    GETYN
013A C25B01             JNZ     START3              ; GOT A "NO"
013D CDD401             CALL    SPMSG               ; GOT A "YES"
0140 0A594F5552         DB      LF,'YOUR ANSWER WAS "YES!"',0
0158 C30C01             JMP     START2
015B CDD401     START3: CALL    SPMSG
015E 0A594F5520         DB      LF,'YOU SAID "NO!"',0
016E C30C01             JMP     START2

0171 5349474E2DSINON:   DB      'SIGN-ON MESSAGE',CR,LF,0
```

```
                        ; MESSAGE POINTED TO BY STACK OUT TO CONSOLE
01D4 E3         SPMSG:  XTHL                    ; GET "RETURN ADDRESS" TO hL
01D5 AF                 XRA     A               ; CLEAR FLAGS AND ACCUMULATOR
01D6 86                 ADD     M               ; GET ONE MESSAGE CHARACTER
01D7 23                 INX     H               ; POINT TO NEXT
01D8 E3                 XTHL                    ; RESTORE STACK FOR
01D9 C8                 RZ                      ; RETURN IF DONE
01DA CD9101             CALL    CO              ; ELSE DISPLAY CHARACTER
01DD C3D401             JMP     SPMSG           ; AND DO ANOTHER

                        ; GET YES OR NO FROM CONSOLE
01E0 CDD401     GETYN:  CALL    SPMSG           ; PROMPT FOR INPUT
01E3 2028592F4E         DB      ' (Y/N)?: ',0
01ED CDB501             CALL    CIMSG           ; GET INPUT LINE
01F0 CDA101             CALL    CCRLF           ; ECHO CARRIAGE RETURN
01F3 3A0502             LDA     INBUF+2         ; FIRST CHARACTER ONLY
01F6 E65F               ANI     01011111B       ; CONVERT LOWER CASE TO UPPER
01F8 FE59               CPI     'Y'             ; RETURN WITH ZERO = YES
01FA C8                 RZ
01FB FE4E               CPI     'N'             ; NON-ZERO = NO
01FD C2E001             JNZ     GETYN           ; ELSE TRY AGAIN
0200 FE00               CPI     0               ; RESET ZERO FLAG
0202 C9                 RET                     ; AND ALL DONE

0203            INBUF:  DS      83              ; LINE INPUT BUFFER

                        ; SET UP STACK SPACE
0256                    DS      64              ; 40H LOCATIONS
0296 00         STAK:   DB      0               ; TOP OF STACK
```

How SPMSG: works

The text to be displayed by this subroutine is in a buffer, just like that for COMSG: in the previous chapter. The only difference is that this buffer is embedded within the program, immediately following the CALL SPMSG:. After the subroutine is called, we still have to fetch one character at a time, test for zero, and output the character through CO: if it is nonzero.

In using COMSG: we pointed to the text by loading the HL index with the start address of the text. Now, the start address of the text is the "opcode" of the "instruction" immediately following the CALL SPMSG:. In other words, it is the return address, or would be for a usual subroutine call.

Since return addresses are pushed onto the stack by CALL opcodes, a pointer to the beginning of the message text is sitting on

top of the stack, waiting for us to load it into an index register. POP H would do this, but would of course destroy the previous contents of the HL register pair. We said we were not going to "use" any indexes. That is not literally true. The subroutine will not *change* any index register contents. Effectively the same thing, as far as the programmer is concerned.

This is accomplished by the use of the super powerful 8080 instruction XTHL. This mnemonic stands for e**X**change **T**op of stack with **HL** register. The "return address" from the stack is moved into the HL pair, at the same time that the contents of the HL pair is moved to the top of the stack. Obviously the 8080 microprocessor contains some temporary holding registers that are invisible to the programmer but that permit this bidirectional simultaneous swap.

So with one instruction that is only one byte long we have both loaded the index with the start address of the message and at the same time saved the previous contents of the index. Now we have to fetch the first character from memory. Just to teach you a new instruction or two, we do this differently this time.

The logical "exclusive or" operation can be abbreviated XOR, and it executes by comparing each bit of the contents of the accumulator with each bit of a second operand. The exclusive part of this OR implies that the resultant bit should be a one if either operand bit is a one (the same as OR) but not if both bits are one (the exclusive part).

1. Zero XOR zero is zero.
2. Zero XOR one is one.
3. One XOR zero is one.
4. One XOR one is zero.

You can compare this "truth table" with those for AND and OR given in previous chapters.

The 8080 mnemonic for this operation is XRA, since the A register is always one of the operands. XRA A says take the exclusive or of A with A. For any bit pair, since both bits are the same, the truth table above says the result will always be zero. XRA A zeros the accumulator. Well, that didn't accomplish much since we want the character that is to be displayed in the A register. If we

now MOV A,M to get it there, we will still have to execute some operation through the ALU to set the zero flag so we can detect the end of message terminator. Since we have zeroed A, we can add the contents of the M register to it. Zero plus any number equals the number, which doesn't accomplish anything either. Except in a computer, where the operands are added in the ALU, and the results stored in the flag register. ADD M in this case sets the zero flag if our fetched character was the terminator.

But it wasn't. We are still pointing to the first character. We add M to A, zero is not set, and now we can output the character. We can, but if we do that right away, we will lose the zero flag bit setting because of all the stuff that would be going on downstream, and we haven't tested it yet. So instead we do things in an orderly manner. First we increment the index to point to the next character. Then we reswap HL and the top of stack. Neither of these double-precision operations affect the flag bits. Now we test zero with the conditional **R**eturn on **Z**ero. If we didn't INX the index and put it back on the stack, the RZ wouldn't have the correct return address to work on. Bomb!

From this you can follow the logic of this subroutine. It fetches and displays each text character in turn, always keeping a pointer to the next character on the top of the stack. When the fetched character is the terminator, the top of the stack contains the real return address, and a return gets us back to the calling program at the instruction following the message text. Well, we wanted to do just that. How about that?

With a dozen carefully chosen bytes of code, carefully arranged in a logical sequence, we have performed a function that relieves the programmer of the tedious and error prone operations of setting up text buffers, counting their characters, and keeping track of their addresses in memory. That was the way it used to be done. And to think that some stuffy old purists consider the 8080 to be a toy. The heck with them. We won't tell them that it is not a toy, but a powerful computing machine.

How GETYN: works

Nothing much new here for you to learn. Everything has previously been covered up to the LDA opcode. LoaD A register fetches the contents of the memory location pointed to by the

address portion of this three-byte instruction. That address is, in this case, the first console input character in the input buffer. Note that we have used ASM.COM's arithmetic ability to point to IN-BUF: plus two, since the first two locations contain the maximum length and character count values that CP/M insists on. We skip over them by letting ASM compute the address INBUF+2.

When we have fetched that first character, we mask off both the eighth bit (ASCII uses only seven, remember) and also the bit that flags the difference between ASCII upper and lower case letters. Two birds killed with one stone. And a new representation learned. The mask byte is specified here in binary, so you can see the bit pattern more easily. This makes the operation more obvious than the hex equivalent mask of 5FH.

Once lower case letters have all been converted to upper case by that operation, we only have to test the upper case possibilities by comparing with the immediate ASCII values for "Y" or "N." If the "Y" compare was true, ComPare Immediate sets the zero flag. So, if the answer was yes, we return to the calling program with zero set.

If no yes, we test again for "N." If no yes and no no, we jump back to the beginning and ask the operator to try again. Wouldn't it be nice if we could reach out of our program and slap the operator's wrist? If the operator input "N," our compare sets the zero flag. We can't return with it set, because that is the signal for a yes answer. So we execute an instruction that we know will clear the zero flag. Compare with zero does just that. Now we return to the calling program.

Assuming your programs include a lot of operator prompting that can be answered yes or no, this subroutine makes the main programs a lot shorter. A simple CALL GETYN followed by the conditional jump are all the instructions the main program has to execute. We have loaded up the subroutine with as much of the burden as possible, even returning with the decision information carried in a simple-to-test flag register bit.

This sort of thing is the "why" of subroutines. It is also the mark of a well designed program. If the main program consists of a string of subroutine calls with little intervening activity, it shows that the programmer did plan ahead, and that a library of sub-routines was created to perform each of the necessary tasks. Sound familiar?

The end of I/O subroutines

With our CPMIO library all completed and checked out, we are ready to put together some subroutines that will get files from the disks and write files back to disks. Then you will be ready to do some real system programming. If you can't think of any more programs that need to be written, don't worry. There will be lots of ideas for you to work on in the final chapter. Your labors are only beginning.

DISK FILE ACCESS
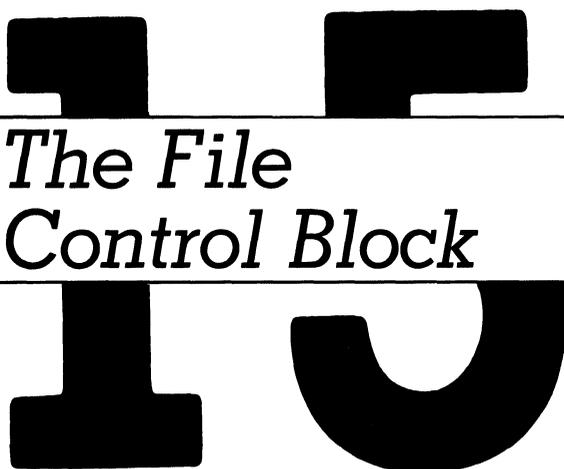
The File
Control Block

CP/M provides the basic I/O capabilities and we have expanded on them in the preceding sections to produce some handy subroutines for handling input and output using logical I/O devices. Logical devices are a mechanism for accessing the physical devices connected to our computer without having to know their hardware-specific characteristics. For program portability, we never directly address the physical I/O devices.

The same is true for disk access. While CP/M provides some handy routines for accessing I/O devices, it is, after all, a disk operating system. So we can expect CP/M to provide equally powerful disk access capabilities. And it does, by taking upon itself the drudgery of keeping track of physical disk addresses (track x sector y), and letting us "talk" to the disk through named files.

We have previously discussed the concept of named files. In this section we will be assembling and testing subroutines that access files as named by the computer operator. And, finally, we will be combining our I/O subroutines and our disk access subroutines in a complete utility program. After completing your copy

of this program, you will know what CP/M does, how it does it, and how to make use of its capabilities. You will be ready to start writing your own programs.

Getting to know the FCB

All of our I/O accesses were made through the giant hook at location 5, and our disk accesses will also use this CP/M entry point. But for access to named files, we will have to pass more information to CP/M than can be stored in all the available registers in our 8080 family CPU. Instead of passing all the required information in registers, we will use a block of RAM memory known as a File Control Block (FCB).

A FCB provides the operating system with the information it has to have to find a desired file on the disk for read operations, and the FCB also contains workspace that the system needs to perform disk write operations. For this reason, the FCB must be in RAM, even if our program always uses the same file name, which could be stored in ROM. Although a file control block can reside anywhere in RAM, and indeed several can be in use at the same time, we will start by using the default transient file control block (TFCB) that CP/M establishes for us at memory locations 005CH through 007CH.

The format of the TFCB is shown in Fig. 15–1. It consists of 33 bytes of memory: the little squares in the figure. An additional three bytes is required for certain optional operations in CP/M versions 2 and above, but we won't have to be concerned with them at this time, since we won't be using those options. Some specific locations within the TFCB that our programs will be working with are labeled in the figure.

The first memory location contains the disk drive selection byte (dr). If this byte is zero, the currently selected drive is assumed for the disk access that uses this FCB. A code byte defining the physical drive that is the currently selected drive is stored in memory location 4, so that our FCB can address a file on another disk without disturbing the current selection. A different drive can be specified in a FCB, and will be selected only temporarily for those disk operations that use that particular FCB.

Next in the block of memory comes the file name and type. The file name field is always eight bytes. Short names are padded

FIGURE 15-1. The format of the File Control Block (FCB) set up automatically by CP/M in response to a command line typed by the operator. The functions of each of the 33 eight-bit bytes within the FCB are discussed in the text.


```
       ┌TFCB                              FCBTY┐        FCBEX┐          ┌FCBS2
       │                                  │     │          │    ┌ FCBRC
       ↓                                  ↓     ↓          │    ↓
FCB+  0   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15
      ┌──────────────────────────────────────────────────────────┐
      │ dr  f1  f2  f3  f4  f5  f6  f7  f8  t1  t2  t3  ex  s1  s2  rc │
      └──────────────────────────────────────────────────────────┘

FCB+ 16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32
     ┌──────────────────────────────────────────────────────────────┐
     │ d0  d1  d2  d3  d4  d5  d6  d7  d8  d9  dA  dB  dC  dD  dE  dF  cr │
     └──────────────────────────────────────────────────────────────┘
                                                        FCBCR ─┘
```
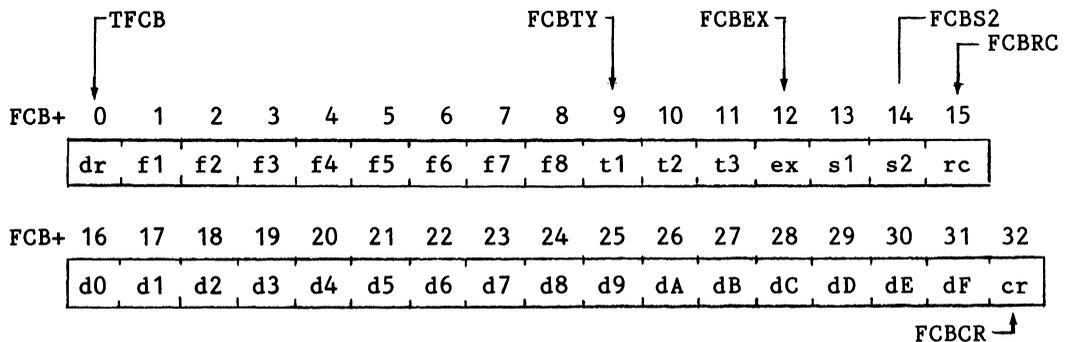
```
dr        = drive select:  0 = current drive
                           1 = drive A:
                           2 = drive B:
                           etc.

f1 to f8  = file name characters

t1 to t3  = file type characters

ex        = file extension

s1        = system use byte

s2        = system use byte

rc        = record count

d0 to dF  = disk allocation groups

cr        = current record
```

out with the ASCII code for space (20H). The three-byte file type field is also padded with spaces, if necessary. For wildcard file accesses, any of these 11 file name/file type bytes can contain the ASCII code for "?" (3FH). Wildcards can be used to find files on disks, but read or write operations should use FCBs that contain an unambiguous file name. Note that the period we use as a name/type separator is not included in the FCB.

The rest of the FCB consists of bytes of binary data. At FCB+12 is an extent byte (ex), used if the file is larger than 16K bytes. As we will be seeing, each FCB can only address 16K bytes of disk space.

After ex are two bytes reserved for internal use by CP/M,

171

followed by the record count byte (rc) at FCB+15. The next 16 bytes (labeled d0 through dF in the figure) will contain the addresses of 1K byte groups of records on the disk. These addresses are not in a nice "track x sector y" format, but are simply binary numbers from zero up to the highest 1K byte block used on the disk. The lowest numbered blocks are reserved by CP/M for the disk directory entries. User blocks typically start at block two.

The extent, record count, group addresses, and the final byte containing a current record counter are all set up and maintained by CP/M. All these FCB locations other than the drive select byte and file name/file type are operating system workspace, and we don't have to concern ourselves with them, or write into them, unless we want to confuse the system.

The operating system will even fill in the drive select byte and file name/file type portion of the TFCB for us, if we enter a command line like:

B:COPY C:FILENAME.TYP

when prompted by CCP. In response to this command line, CCP will load the program named COPY.COM from drive B: into the TPA, and set up the FCB drive select byte for C:, file name bytes to FILENAME, and file type bytes to TYP. The rest of the TFCB will be zeroed out, in anticipation of our use of the TFCB to perform disk reads or writes, or other file referencing operations. The TFCB is then ready for COPY.COM to use to access FILENAME.TYP on drive C:. Drive A: will still be maintained as the current drive, and will be returned to for the next CCP prompt (A>).

We will be using this technique a little later on in this chapter. But first we have to see how CP/M uses the rest of the file control block, and how disk directory entries are made.

How CP/M uses the FCB

In Chap. 5 we saw that the basic unit of data storage on standard floppy disks is the 128 byte record. Even if the physical sector length on your disks is not 128 bytes, CP/M will still make it appear that you are working with 128 bytes of data for each read or write operation.

When we use the FCB to access named files, we let CP/M keep track of all those little 128 byte records for us, and part of the overhead involved is the fact that the operating system will allocate disk space in groups consisting of eight records each. Each group therefore contains 1024 bytes, the minimum size (other than zero bytes) of any file on a CP/M disk.

Working with groups simplifies keeping track of where all the files are on a disk. Since the first floppy disks contained 76 tracks of 26 sectors, one disk could hold about 240 groups, after subtracting space for the operating system and disk directory. Using this 1K byte granularity, a unique address for each group on a disk can fit into a single 8-bit byte. This was the deciding factor in setting up the granularity of the disk space allocation.

The operating system handles all of the computations involved in translating a group number to the group's starting address at track x sector y. It must also keep track of the count of records within each group.

Creating a disk file

When we want to write a file onto a disk, we provide CP/M with a FCB containing the drive selection and the file name and type. As we write the first record, CP/M will find the lowest numbered disk group not already allocated to another file, and will place its 8-bit address into byte d0 in the file control block in memory. We can then write up to eight records into that group. As each record is written, CP/M updates the record counts in rc and cr in the FCB.

Each group of eight records written causes another unused 1K bytes of disk space to be allocated to the file, until we close the file, or until all 16 of the disk allocation bytes (d0 through dF) in the FCB are full. If our file is larger than 16K bytes, CP/M will automatically open an extension of the file, incrementing the contents of ex in the FCB for each 16K bytes of disk space used.

Whenever we are done writing into a file, we tell CP/M to "close" the file. At that time the first 32 bytes of the FCB are written onto the disk as a directory entry. Since the directory entry has to be on the same disk as the data, and that disk can then be put into any drive, the drive select byte (dr) in the FCB image in the disk directory is zeroed. If we later erase the file, that first byte is rewritten as 0E5H in the directory, and CP/M knows it can reuse

all of the groups allocated to this file, as well as the FCB image space in the directory.

If our writes to the file overrun that first 16K byte boundary, CP/M will write the filled FCB image into the directory, thus keeping track of the first 16 groups allocated, and then start all over with the disk allocation map in the FCB zeroed, but with the extension byte incremented. This will be repeated for each 16K byte extension for large files. When an extended file is closed, the last FCB image will be written into the disk directory. With 64 directory entries available on the standard floppy disk, there is more than enough directory space to store the multiple FCB images necessary for extended files.

We can see what all this looks like by referring to Fig. 15–2. In 15–2a, DDT has been used to dump the TFCB (memory locations 005CH through 007CH) after file COPY.PRN has been written to the disk. When DDT dumps up to 16 bytes of memory on each line, it follows this display with a display of the ASCII equivalents of all printing characters (non-printing characters are shown by periods). This makes it easier to spot our file name.

Since the TFCB doesn't start on a nice even memory address, the DDT dump isn't as nicely formatted as Fig. 15–1. We can still make out the different elements within the TFCB, however. We can see that groups 21, 22, 25, 26....3A have been allocated to this file. The group numbers are not contiguous, giving evidence that this disk has had some files erased in the past.

When COPY.PRN was closed after the write, the directory entry shown in Fig. 15–2b was created. Here one 128 byte sector from the directory track is displayed. That sector contains four directory entries. We can see that COPY.PRN consists of 72H (114 decimal) records (14,592 bytes) written into 15 groups, which could have held a total of 15360 bytes. Some disk space (six 128 byte records) is therefore wasted, as a result of our 1K byte granularity.

The other directory entries shown in Fig. 15–2b illustrate how CP/M extends files (WORD.COM) and how erased files are flagged (GET.COM). The directory entry for an erased file is a temporary thing. The entire directory entry will be overwritten by a new entry the next time CP/M is called upon to open a new file on the disk. Thus both unused disk space and erased file directory entries are reused by CP/M. This is what is meant by dynamic disk space allocation.

FIGURE 15–2. Typical file control block contents. The contents of the default TFCB have been dumped by DDT in Fig. 15–2a. A more readable dump is in Fig. 15–2b, where the contents of four FCBs within one disk directory sector are shown.

(a)

```
-D5C,7C

005C 00 43 4F 50                                     .COP
0060 59 20 20 20 20 50 52 4E 00 00 00 72 21 22 25 26 Y    PRN...r!"%&
0070 27 28 2A 2B 2C 2E 36 37 38 39 3A 00 72          '(*+,.6789:.r
-
```

(b)

```
*A: T 2 S 13

               DRIVE A - TRACK 2  SECTOR 13
0000  00 57 4F 52 44 20 20 20  20 43 4F 4D 00 00 00 80  .WORD    COM....
0010  19 1A 1B 1C 1D 1E 1F 20  29 2D 3B 3F 40 41 42 43  .......  )-;?@ABC
0020  00 57 4F 52 44 20 20 20  20 43 4F 4D 01 00 00 32  .WORD    COM...2
0030  44 45 46 47 48 49 4A 00  00 00 00 00 00 00 00 00  DEFGHIJ. ........
0040  00 43 4F 50 59 20 20 20  20 50 52 4E 00 00 00 72  .COPY    PRN...r
0050  21 22 25 26 27 28 2A 2B  2C 2E 36 37 38 39 3A 00  !"%&'(*+ ,.6789:.
0060  E5 47 45 54 20 20 20 20  20 43 4F 4D 00 00 00 05  eGET     COM....
0070  24 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  $....... ........
```

SHOFN: displays the TFCB file name

Before moving on to the disk access subroutines in the rest of this section, you are going to have to add a few more I/O subroutines to your ever-growing library. But since these are "disk-oriented" I/O subroutines, their source code should be kept in its own .LIB file. This is a good time to break up CPMIO.ASM into smaller library files, anyway, to make future additions easier. The first addition is SHOFN:, a subroutine to display the drive, file name, and file type fields from the TFCB.

Before proceeding with editing the new additions and merging them with your existing library, let's take a look at SHOFN: and see how it works. You will want to refer to the TFCB format (Fig. 15–1) as well as the source listing for SHOFN: (List. 15–1). SHOFN: begins by saving the contents of the BC and HL

LISTING 15–1. SHOFN.LIB

```
; DISPLAY FILENAME.TYP FROM TRANSIENT FCB
SHOFN:  PUSH    B               ; SAVE TEMP STORE
        PUSH    H               ; AND INDEX
        LDA     FCBTY           ; SAVE FIRST CHAR OF TYPE
        MOV     C,A             ; IN TEMPORARY STORE
        XRA     A               ; FORCE TWO TERMINATORS
        STA     FCBTY           ; FOR FILE NAME
        STA     FCBEX           ; AND FILE TYPE
        LXI     H,TFCB          ; SHOW DISK DRIVE
        MOV     A,M
        ANI     0FH             ; LIMIT TO 4 BITS
        ORI     40H             ; CONVERT TO ASCII
        CALL    CO
        MVI     A,':'           ; SHOW THE COLON
        CALL    CO
        INX     H               ; AND SHOW THE FILE NAME
        CALL    COMSG
        MOV     A,C
        LXI     H,FCBTY         ; RESTORE TYPE
        MOV     M,A
        MVI     A,'.'           ; SHOW SEPARATOR
        CALL    CO
        CALL    COMSG           ; SHOW TYPE
        POP     H
        POP     B               ; RESTORE AND
        RET                     ; RETURN
```

register pairs, the only registers other than the accumulator that it will be disturbing. The subroutine will be using the message output subroutine (COMSG:) to display the file name and type, and COMSG: wants a zero terminator at the end of the string to be written to the console. So the next step is to save the contents of FCBTY and stuff a zero there to terminate the file name string. The save is done by loading the memory contents into the A register, then moving it into the C register. A is then zeroed, and that zero stored in the same memory location. Another zero is then stuffed into FCBEX to terminate the file type string. Now we are ready to display.

The disk drive designator is fetched from TFCB, limited to the lowest four bits, and ORed with 40H to convert it to ASCII. Since this byte can take on the values 0, 1, 2,... up to 15, ORing it with 40H converts it to @, A, B,...O. If the current drive has been specified, the "at sign" is shown, and can be interpreted as "at the current drive." Otherwise the drive designator letter is displayed,

followed by a colon, both by using the single character console output subroutine CO:.

The index is then incremented to point to the first file name character, and the file name is displayed by the call to COMSG. The index is reinitialized to point to FCBTY, the original contents of this location restored, and then a period and the type field are output to the console. The registers we used are then restored, and the calling program is returned to.

Note that within this subroutine, absolute memory addresses are referenced in two ways: by the load and store instructions LDA and STA, or by setting up an index pointing to the desired memory location. In general, it is faster and uses fewer registers to use the "direct addressing" instructions LDA and STA, but these instructions are only efficient for accessing a single memory location, and can only transfer data between that location and the accumulator.

Using the index register is much more efficient when referencing a string of memory addresses. Then "indexed addressing" instructions like MOV A,M can be used. The selection of one technique or the other is up to the programmer. That is you, now that you know when to use one method and when to use the other.

Breaking up with ED

Since you have spent so much time in creating the CPMIO subroutines and the test program completed in Chap. 14, you may be a little reluctant at this point to start breaking all that up into little pieces. But there is a very good reason for doing so, and the process can be accomplished without much pain using ED.

In this section we will be writing disk access subroutines, and combining them with the previously written subroutines into a complete program. When this program is completed, you will probably be ready to start working on program ideas of your own. The subroutines developed and tested as you work your way through this book provide the base for your future efforts.

But each of your future programs will not need to include all of the subroutines. For this reason you should create a library of groups of subroutines that will be used together in most programs. The CPMIO subroutines fit together in one group, and will be used in programs that need computer operator interaction. The

disk subroutines from this chapter fit together into another group. Programs that include both disk and operator accesses would include both groups.

Similarly, other parts of complete programs are best maintained in separate .LIB files and combined with the main program after it is written. Table 15–1 lists the .LIB files that will be generated in this section. CPMIO.LIB is only one of them. It is a part of the program you put together in the last chapter.

That program will contribute the text for two .LIB files, IOEQU.LIB and CPMIO.LIB. The first will contain all of the EQUates found at the beginning of the program. These specify the ASCII characters, CP/M functions, and CP/M addresses used by CPMIO. The second file will contain the I/O subroutines alone, without the test program or the references to RAM locations found in the complete program.

TABLE 15–1. The library of user subroutines constructed by completing all of the exercises in this book.

Source Listing	File Name	Contents
(Part of CPMIO.ASM)	IOEQU.LIB	Data and address value assignments for nondisk programs.
15-2	DISKEQU.LIB	Data and address value assignments for disk programs.
18-1	COPY.LIB	COPY main program.
16-1	GET.LIB	GET: a file from disk subroutine.
17-1	PUT.LIB	PUT: a file onto disk subroutine.
15-1	SHOFN.LIB	Show file name subroutine.
16-2	DISKSU.LIB	Disk subroutines, miscellaneous.
(Part of CPMIO.ASM)	CPMIO.LIB	Input/output subroutines.
16-3	RAM.LIB	Memory area assignments.

To create these two .LIB files, start by making two copies of CPMIO.ASM:

PIP IOEQU.LIB=CPMIO.ASM
PIP CPMIO.LIB-CPMIO.ASM

and then edit them in turn, killing all of the lines you do not want to retain in the .LIB files. You may want to include title lines at the start of each file, giving their creation date.

Adding more .LIB files

IOEQU.LIB should include all of the text from "; ASCII CHAR-ACTERS" up to, but not including "ORG TPA." When you have this one complete, use PIP to make a copy of it named DISKE-QU.LIB. Now edit this file to bring it up to its final configuration, shown in List. 15–2.

LISTING 15–2. DISKEQU.LIB

```
; ASCII CHARACTERS
CR       EQU     0DH             ; CARRIAGE RETURN
LF       EQU     0AH             ; LINE FEED
CTRLZ    EQU     1AH             ; OPERATOR INTERRUPT

; CP/M BDOS FUNCTIONS
RCONF    EQU     1               ; READ CON: INTO (A)
WCONF    EQU     2               ; WRITE (A) TO CON:
RBUFF    EQU     10              ; READ A CONSOLE LINE

; CP/M DISK ACCESS FUNCTIONS
INITF    EQU     13              ; INITIALIZE BDOS FUNCTION
OPENF    EQU     15              ; OPEN FILE FUNCTION
CLOSF    EQU     16              ; CLOSE FILE FUNCTION
FINDF    EQU     17              ; FIND FILE FUNCTION
DELEF    EQU     19              ; DELETE A FILE FUNCTION
READF    EQU     20              ; READ ONE RECORD FUNCTION
WRITF    EQU     21              ; WRITE ONE RECORD FUNCTION
MAKEF    EQU     22              ; CREATE FILE FUNCTION
SDMAF    EQU     26              ; SET DMA FUNCTION

; CP/M ADDRESSES
RBOOT    EQU     0               ; RE-BOOT CP/M SYSTEM
DRIVE    EQU     4               ; CURRENT DRIVE SELECTION
BDOS     EQU     5               ; SYSTEM CALL ENTRY
MEMAX    EQU     7               ; MSB OF TOP OF MEMORY
```

LISTING 15–2. *Continued*

```
TFCB    EQU     5CH             ; TRANSIENT FILE CONTROL BLOCK
FCBTY   EQU     TFCB+9          ; FILE TYPE IN FCB
FCBEX   EQU     TFCB+12         ; FILE EXTENT IN FCB
FCBS2   EQU     TFCB+14         ; SYSTEM USE IN FCB
FCBRC   EQU     TFCB+15         ; RECORD COUNT IN FCB
FCBCR   EQU     TFCB+32         ; CURRENT RECORD IN FCB
TBUFF   EQU     80H             ; TRANSIENT BUFFER
TPA     EQU     100H            ; TRANSIENT PROGRAM AREA

; CP/M FLAGS
BDAOK   EQU     0               ; BDOS RETURN FOR ALL OK
BDER1   EQU     1               ; BDOS RETURN ONE
BDER2   EQU     2               ; BDOS RETURN TWO
BDERR   EQU     255             ; BDOS RETURN ERROR FLAG
```

This file now contains all of the addresses we will be using for disk accesses, as well as those required by the I/O subroutines. For programs with only I/O usage, the shorter EQUates file can be used.

Merging files with PIP

Now, almost, we are ready to test our latest subroutine. All we have to do is put together a simple test program (List. 15–3) and merge it with DISKEQU.LIB, SHOFN.LIB, and CPMIO.LIB. Call this file TESTC15.LIB.

LISTING 15–3. TESTC15.LIB

```
        ORG     TPA

        JMP     START           ; SKIP OVER RAM SPACE
HSAVE:  DS      2               ; TWO BYTES OF STORAGE
        DS      32              ; SOME STACK SPACE
START:  LXI     H,0             ; GET AND SAVE THE
        DAD     SP              ; BDOS STACK POINTER
        SHLD    HSAVE           ; IN TEMPORARY STORAGE
        LXI     SP,START        ; SET LOCAL STACK POINTER
        CALL    TWOCR           ; SPACE TWO LINES
        CALL    SHOFN           ; DISPLAY THE TFCB
        CALL    TWOCR           ; SPACE MORE
        LHLD    HSAVE           ; RESTORE THE BDOS STACK
        SPHL
        RET                     ; AND RETURN TO BDOS
```

Merge the .LIB files required for testing SHOFN: by entering:

PIP TESTC15.ASM=DISKEQU.LIB,TESTC15.LIB,SHOFN.LIB,CPMIO.LIB

This demonstrates another way to merge files, other than using the ED command "R." It comes in handy when the total size of the .ASM file ends up being larger than the editor buffer space available. That limit is easily reached in a 16K CP/M system.

When merging files, make sure that the main program precedes the files containing subroutines. The first executable statement that ASM sees must be the entry to the main program, and must be assembled at the TPA start address.

Testing SHOFN:

With all the pieces put together and assembled you should be ready to test SHOFN:. Since this program simply displays the TFCB set up for us by CCP, it never uses INBUF, and our editing has eliminated INBUF:. So you see some ASM error messages. This time they can be ignored. To test SHOFN: enter:

TESTC15 B:FILENAME.TYP

or some similar line,using whatever drive selection and file name you want. These will be displayed, and then a return to CP/M is made.

There are a couple of new techniques embodied in TESTC15. To be compatible with the final versions of our .LIB files, the stack space has been included in this part of the program. The TPA entry contains a jump to START: and the space between the jump and START: is used for storage and stack space.

To implement the quick return to CCP that was discussed in Chap. 9, we have to either use CCP's stack, which might be too small, or save the CCP stack pointer value that exists when our program in the TPA is called by CCP. We do the latter in this example. There is no 8080 instruction that permits storing the contents of SP directly, but we can add it to HL in one of the few 16-bit arithmetic operations included in the 8080 instruction set.

To save and restore CCP's stack pointer, we first zero HL and then add SP to HL. This puts the contents of SP into HL in a roundabout way. We then stuff this into HSAVE, and set our own stack pointer value. When our program execution is completed we restore the CCP SP by loading HL with the value saved in HSAVE, and then we can transfer it to SP with the 16-bit register to register move instruction SPHL.

This technique was thrown in here because you might find it useful in your programs in the future. Just remember that it will only work if your program never disturbs CP/M as loaded into your computer's memory. This method is the fast return to CP/M. A jump to RBOOT is the safe, but slower, return.

GET: Reads a File From the Disk

In the next three chapters we will be looking at subroutines that will read a file from disk into memory, and then write a file from memory to the disk. Finally, all the pieces we have so far assembled will be put together into a complete utility program.

When the first demonstration programs and subroutines were discussed in this book, virtually every instruction used was explained in depth. In more recent chapters the explanations have become less detailed. As you have been learning the language, you have developed a feel for what is going on, and by now you should be reaching that state of enlightenment where the program listings alone should be self-explanatory. Ultimately, you will be able to read a memory dump.

If the operation of any instruction in any of these program examples is not obvious from its context, or the comments in the listing, or the discussion in the book, there is still the "dictionary" to turn to for help. All of the information in the Intel programming manual has not been duplicated in this book. You will still be referring to that manual years after this book has been closed for the last time.

Find it fast in the directory

Let's write a program that will read a named file into memory, and then write that file out to another disk in any disk drive. To enable making lots of copies, our COPY program will let you swap the disk written to, and replace it with a new one and write another copy. Any number of times you want.

This COPY program will be handier to use than PIP if you want multiple copies, and in addition can be used on computers with only a single disk drive. And any computer can become a single-drive system, if one disk drive is temporarily sick. So COPY will be a useful utility, in addition to being a means for demonstrating all our subroutines.

COPY will assume that the computer operator has entered a command line in the format:

B:COPY C:FILENAME.TYP

so that CCP will set up the TFCB before executing COPY, as was discussed in Chap. 15. COPY can be on any disk, the target program can be on any disk, and the disk to be written on can be placed in any drive.

The first task the program must accomplish is to verify that the target file is on the disk that the operator specified. CP/M provides a BDOS function for finding a file, and the list of functions (Table 6–1) calls it FINDF, and shows that it has a function code of 17. We could write a BDOS call using this function to see if the file is on the disk. But it turns out that we can instead kill two birds with one BDOS call.

The CP/M Interface Manual lists all available functions, and discusses their operation in more depth than is shown in Table 6–1. If you refer to that manual, you will see that if a program just assumes that the file is on the disk specified, and goes ahead and tries to open the file (function 15), BDOS will return an error code showing that the file cannot be found. If the file is on the disk, it will be opened, and we can proceed to read it. Two functions in one BDOS call.

Since finding a file and then opening it for read must precede any attempt to read the first record, these functions are part of subroutine GET:, in keeping with our policy of making the main

program as simple as possible by loading up the subroutines with work. GET: starts by setting up a pointer into the buffer space in RAM that is to receive the contents of the file. It then tries to open the file, calling BDOS with DE pointing to the TFCB and C containing the OPENF code.

If BDOS returns an error code of 0FFH (BDERR in List. 15–2), we know the file is not on the disk specified, so we tell the operator that the program "CAN NOT FIND C:FILE-NAME.TYP" and we return to CP/M through ERREX: and DONE: (List. 16–1).

LISTING 16–1. GET.LIB

```
; READ A FILE FROM DISK INTO "BUFFR"

GET:      LXI     H,BUFFR        ; GET BUFFER START
          SHLD    NEXT           ;   ADDRESS FOR DMA
          LXI     D,TFCB         ; SEE IF FILE IS ON DISK
          MVI     C,OPENF        ; AND OPEN FOR READ
          CALL    BDOS
          CPI     BDERR          ; IS IT THERE?
          JNZ     GET1           ; YES, READ IT IN
          CALL    TWOCR          ; NO, SHOW ERROR
          CALL    SPMSG
          DB      'CAN NOT FIND ',0
          CALL    SHOFN          ; SHOW FILE NAME
ERREX:    CALL    TWOCR          ; ERROR EXIT TO CP/M
          JMP     DONE

GET1:     XRA     A              ; ZERO RECORD COUNTER
          STA     RECCT          ; AND READ A FILE INTO BUFFR
GET2:     LHLD    NEXT           ; SET BUFFER ADDRESS
          XCHG
          MVI     C,SDMAF
          CALL    BDOS
          LXI     D,TFCB         ; READ ONE RECORD INTO
          MVI     C,READF        ; BUFFER
          CALL    BDOS
          CPI     BDAOK          ; READ OK?
          JZ      GET3           ; YES, DO MORE
          CPI     BDER1          ; MAYBE, END OF FILE?
          JZ      GETEX          ; YES, NO PROBLEM
          CALL    REMSG          ; NO, SHOW ERROR
          JMP     ERREX          ; AND ALL DONE

GET3:     LDA     RECCT          ; COUNT THE RECORD
          INR     A
          STA     RECCT
          LHLD    NEXT           ; INCREMENT BUFFER ADDRESS
          LXI     D,128          ; BY RECORD SIZE
```

LISTING 16–1. *Continued*

```
          DAD     D
          SHLD    NEXT
          LDA     MEMAX             ; ROOM LEFT IN RAM?
          DCR     A                 ; STOP BELOW CCP
          CMP     H                 ; COMPARE MSB
          JNZ     GET2              ; CONTINUE IF NOT EQUAL
          CALL    TWOCR             ; ELSE SHOW OUT OF MEMORY
          CALL    SPMSG
          DB      'OUT OF MEMORY',0
          JMP     ERREX             ; AND GIVE UP

GETEX:    CALL    CCRLF             ; NORMAL EXIT
          CALL    CPDMA             ; RESTORE CP/M DMA
          RET
```

Read the file into BUFFR

A successful open results in a jump to GET1: and we zero a record counter in RAM before entering a loop that will fetch successive records from disk and store them in RAM until the complete file is read, or we run out of memory. The flowchart of GET: in Fig. 16–1 will help you follow the logic of this loop as we examine it in the program listing.

We initialized the contents of NEXT with the starting address of BUFFR when we first entered GET:. NEXT will always contain the memory address of the next block of RAM to be loaded with data from the disk. The contents of NEXT are passed to BDOS with the Set Direct Memory Address Function (SDMAF) so that CP/M will read the records into our buffer instead of the default transient buffer (TBUFF) at location 80H.

With our memory address passed on to CP/M, we next call BDOS with the read record function, READF. One record, 128 bytes, will be transferred from disk into memory, and one of three possible error messages will be returned to us by BDOS. The nicest BDOS message is "All is OK," so we test for it first. If BDAOK, we continue the program at GET3:.

If all was not OK, we test for an error return of one, which means that the end of the file has been reached. If so, we are all done, and exit through GETEX:, restoring the default buffer DMA address (CALL CPDMA) as we go. This is important, as BDOS

FIGURE 16–1. The flowchart of subroutine GET:. Flowchart symbols include ovals for entry points and labels; action blocks; decision diamonds; and circles showing continuations to and from programs not part of the flowchart. Flowcharts make it easy to follow the logic of a complicated program.
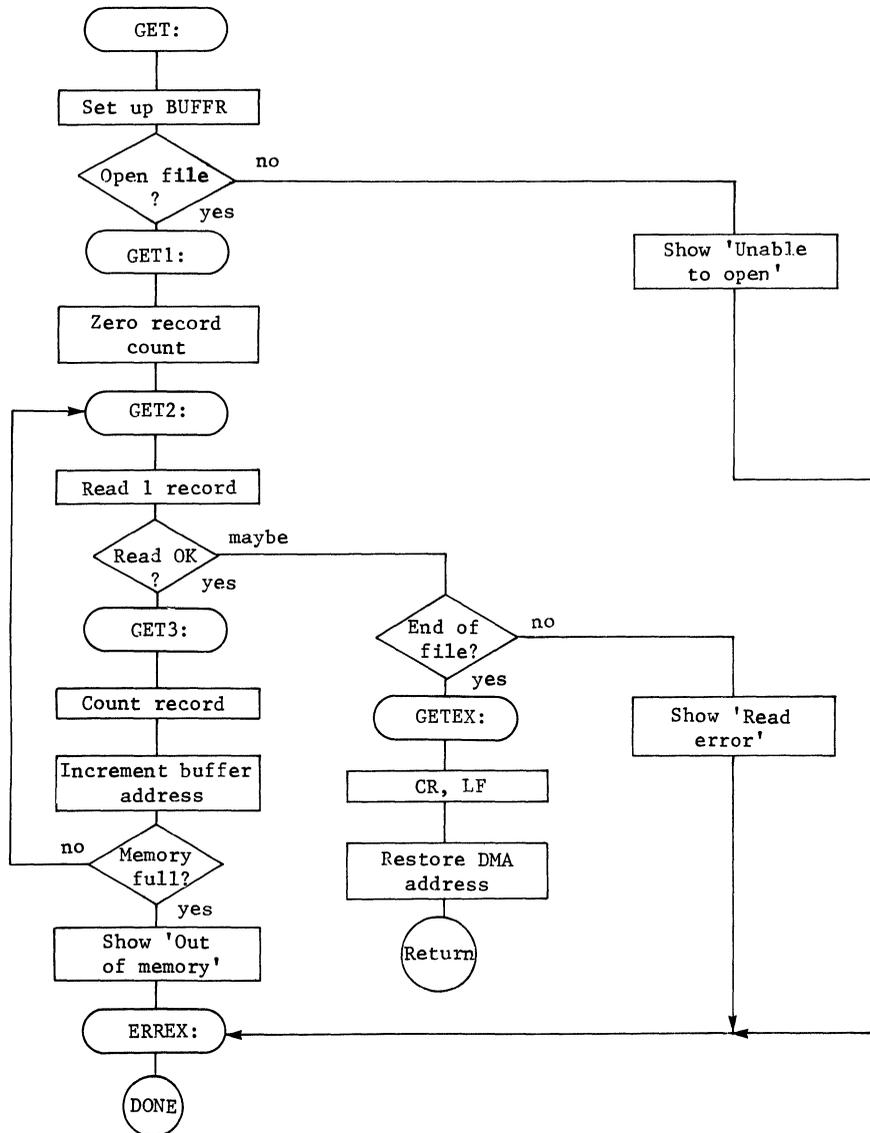
needs a buffer to write into every time it reads a disk directory, and we don't want it writing into BUFFR!

If all was not OK, and the reason was not that we were done reading, then a real error has occurred, and we have to give up trying to COPY this file. We so inform the operator by calling an error message routine (REMSG) and then exit to CP/M through ERREX:.

Back at "All OK" we jumped to GET3:, and here we count the record and increment the buffer address by 128 bytes. We then compare this new address with the highest address we dare write to, the bottom of BDOS in the CP/M system. If our file is big enough, it will overwrite CCP in memory, but we will stop short of overwriting BDOS, since we need BDOS to handle the disk accesses.

The giant hook into CP/M at memory location 5 contains a jump to the BDOS entry point. The third byte of the jump instruction, at location 7, is the most significant byte of the 16-bit memory address of the BDOS entry. If we compare this byte (MEMAX, List. 15–2) against the most significant byte of our current DMA address, we could detect that we have just overwritten the BDOS entry.

Since that is what we have to avoid, we instead compare MEMAX with the most significant byte of the DMA address after it has been decremented by one. This insures that we will always stop before we overwrite BDOS. If the compare is alright, we continue the read loop (JNZ GET2), otherwise we give up, informing the operator that we have run out of memory. That, like the read error, is an unrecoverable error. We can't use GET: to read this file, so we can't use COPY to copy this file.

Back to you, ED

The listing of GET: is the source listing for GET.LIB. It includes references to subroutines that are part of DISKSU.LIB, and memory addresses that are defined in RAM.LIB. So, while you have your computer all fired up entering GET.LIB, go ahead and type in the disk subroutines in List. 16–2 and the RAM definitions in List. 16–3. As you are entering these files you can be teaching

LISTING 16-2. DISKSU.LIB

```
; DISPLAY READ ERROR MESSAGE
REMSG:  CALL    TWOCR
        CALL    SPMSG
        DB      'PERMANENT READ ERROR',CR,LF,0
        RET

; DISPLAY WRITE ERROR MESSAGE
WEMSG:  CALL    TWOCR
        CALL    SPMSG
        DB      'PERMANENT WRITE ERROR',CR,LF,0
        RET

; DISPLAY WRITE OPEN ERROR MESSAGE
WROPN:  CALL    TWOCR
        CALL    SPMSG
        DB      'CAN NOT OPEN FOR WRITE',CR,LF,0
        RET

; RESTORE CP/M DMA ADDRESS TO THE TRANSIENT BUFFER
CPDMA:  LXI     D,TBUFF
        MVI     C,SDMAF
        CALL    BDOS
        RET

; GET A VALID DRIVE SELECT DESIGNATOR
DRSEL:  CALL    CIMSG         ; INPUT THE SELECTION
        LDA     INBUF+2       ; USE FIRST CHARACTER ONLY
        ANI     01011111B     ; CONVERT TO UPPER CASE
        SUI     '@'           ; SET A=1, B=2, ETC.
        JM      DRERR         ; CAN'T BE LESS THAN ZERO
        SUI     17            ; OR GREATER THAN 16
        JP      DRERR
        ADI     17            ; RESTORE LEGAL NUMBER
        RET                   ; AND RETURN WITH IT
DRERR:  XRA     A             ; ELSE SET ZERO FLAG
        RET                   ; AND RETURN
```

LISTING 16-3. RAM.LIB

```
; RAM VARIABLES AND BUFFERS
INBUF:  DS      83            ; LINE INPUT BUFFER
DRSAV:  DS      1             ; CURRENT DRIVE AT ENTRY
RECCT:  DS      1             ; TOTAL RECORDS READ/TO WRITE
CTSAV:  DS      1             ; SAVE LOCATION FOR COUNT
NEXT:   DS      2             ; NEXT DMA ADDRESS

; SET UP STACK SPACE
        DS      64            ; 40H LOCATIONS
STAK:   DB      0             ; TOP OF STACK

SINON:  DB      'MULTI-WRITE FILE COPY  12 SEPT 82',0

; FROM HERE THROUGH CCP IS BUFFER SPACE
BUFFR:
        END
```

yourself how the subroutines work, and what the RAM areas are used for.

    We will not generate any test programs in this or the next chapter. The final testing will be done when all of the .LIB files are complete, and are merged into COPY.ASM. Then the big debugging job will start. Something for you to look forward to.

PUT: Writes a File Onto the Disk

One of the first things an operator learns when using the CP/M operating system is that you can't change a disk in a drive if you want to write on it. Change a disk, and you have to reboot the system before that disk will be accepted by the system for both read and write (R/W) operations.

Our COPY program would not be very useful if it did not include a method for overcoming this characteristic of CP/M. By using the BDOS function INITF (function code 13), the program overcomes the read only (R/O) tag placed on a changed disk by the operating system. By overcoming the R/O status, we are able to read a file into memory one time, and then write it out many times on many disks. This is handled within subroutine PUT:.

How PUT: works

The experts all agree that assembly language code is so difficult to follow that no program or subroutine source listing should be long-

er than one page. PUT: won't quite make it on a single 8½ × 11 inch page, but it will fit on a legal size sheet. So that makes it legal, and no experts should be offended by this subroutine.

Refer to the PUT: source listing (List. 17–1) and flowchart (Fig. 17–1) for the following discussion. PUT: opens by "rewinding" the buffer pointer address in NEXT that GET: (or a previous PUT: pass) left pointing to the wrong end of the buffer. It then saves the record count (RECCT) left by GET: so that multiple writes can be made of the file. PUT: then looks at the drive select byte in the transient file control block to see that it is greater than zero.

Between the time that GET: read the file into memory and PUT: was called upon to write it, the operator was asked by the main program (Chap. 18) for a drive selection, and that selection must be A, or B, or C, etc. The current drive is assumed to be selected anytime no drive is specified in the command line. That was a valid selection for read, but this write routine insists on a real drive designator input by the operator. Otherwise it aborts the write attempt, and exits through PUTEX:.

With a valid drive number in TFCB, PUT1: enables the disk for writing by calling the initialize disk function of BDOS. This acts the same as a reboot without the return to CCP. You may notice that this action causes the current disk to be accessed momentarily, even if the write operation is going to be to another drive. Whatever CP/M is doing in response to INITF, it enables all the disks

LISTING 17–1. PUT.LIB

```
; WRITE A FILE FROM "BUFFR" TO DISK

PUT:    LXI     H,BUFFR         ; SET UP BUFFER START
        SHLD    NEXT
        LDA     RECCT           ; SAVE RECORD COUNT
        STA     CTSAV
        LDA     TFCB            ; LOG-IN SELECTED DISK
        ORA     A               ; IS IT LEGAL?
        JNZ     PUT1
        CALL    WROPN           ; NO, SHOW UNABLE TO OPEN
        JMP     PUTEX           ; AND TRY AGAIN

PUT1:   MVI     C,INITF         ; ENABLE WRITE ON ANY DISK
        CALL    BDOS
        XRA     A               ; INITIALIZE FCB
        STA     FCBCR           ; CURRENT RECORD
```

LISTING 17-1. *Continued*

```
                LXI    H,0
                SHLD   FCBEX              ; EXTENT AND S1
                SHLD   FCBS2              ; S2 AND RECORD COUNT
                LXI    D,TFCB             ; SEE IF FILE EXISTS
                MVI    C,FINDF            ; FIND FUNCTION
                CALL   BDOS
                CPI    BDERR              ; IS IT ALREADY?
                JZ     PUT2
                CALL   CCRLF              ; YES, OK TO ERASE?
                CALL   SPMSG
                DB     'OK TO ERASE ',0
                CALL   SHOFN
                CALL   GETYN
                JNZ    PUTEX              ; IF NO, TRY AGAIN
                LXI    D,TFCB             ; IF YES, ERASE IT
                MVI    C,DELEF            ; DELETE FUNCTION
                CALL   BDOS
PUT2:           LXI    D,TFCB             ; OPEN FILE FOR WRITE
                MVI    C,MAKEF            ; MAKE A FILE FUNCTION
                CALL   BDOS
                CPI    BDERR              ; GOT IT MADE?
                JNZ    PUT3
                CALL   WROPN              ; NO, SHOW UNABLE
                JMP    PUTEX              ; AND TRY AGAIN

PUT3:           LHLD   NEXT               ; WRITE BUFFER TO DISK
                XCHG                      ; (FINALLY)
                MVI    C,SDMAF            ; SET ADDRESS TO WRITE FROM
                CALL   BDOS
                LHLD   NEXT               ; THEN INCREMENT BY 128
                LXI    D,128
                DAD    D
                SHLD   NEXT
                LXI    D,TFCB
                MVI    C,WRITF            ; WRITE A RECORD FUNCTION
                CALL   BDOS
                CPI    BDAOK              ; WRITE OK?
                JZ     PUT4
                CALL   WEMSG              ; NO, WRITE ERROR MESSAGE
                JMP    PUTEX              ; AND TRY AGAIN

PUT4:           LDA    RECCT              ; COUNT THE RECORD
                DCR    A
                STA    RECCT              ; IF NOT END OF RECORDS
                JNZ    PUT3               ; THEN DO ANOTHER
                CALL   CPDMA              ; ELSE RESTORE CP/M DMA
                LXI    D,TFCB
                MVI    C,CLOSF            ; THEN CLOSE THE FILE
                CALL   BDOS
                LDA    CTSAV              ; RESTORE RECORD COUNT
                STA    RECCT              ; FOR NEXT WRITE
PUTEX:          CALL   CCRLF
                CALL   CPDMA              ; RESTORE CP/M DMA
                RET                       ; AND ALL DONE
```
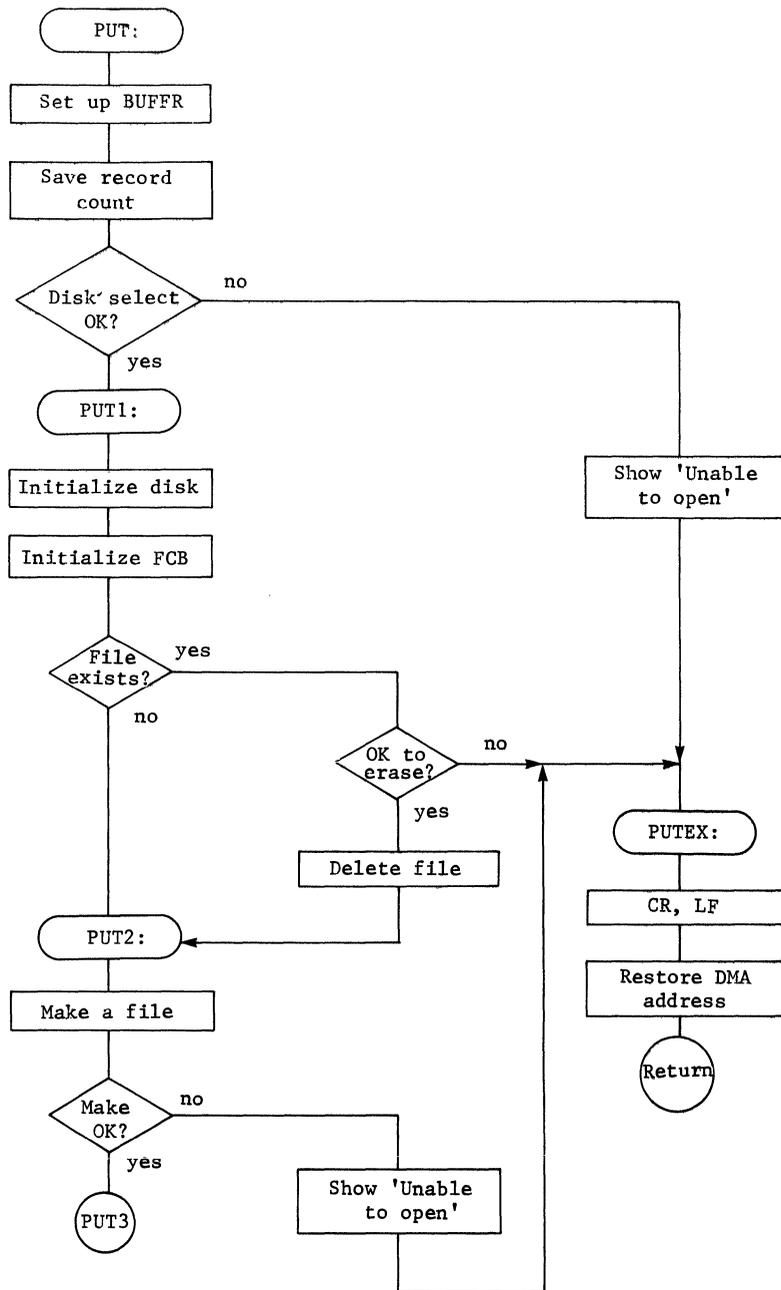
```
                    ┌──────────────┐
                   (    PUT:        )
                    └──────┬───────┘
                    ┌──────┴───────┐
                    │ Set up BUFFR │
                    └──────┬───────┘
                    ┌──────┴───────┐
                    │ Save record  │
                    │    count     │
                    └──────┬───────┘
                           │
                        ╱──┴──╲          no
                       ╱ Disk  ╲────────────────────────────────┐
                       ╲ select ╱                               │
                        ╲ OK? ╱                                 │
                         ╲─┬─╱                                  │
                          yes                                   │
                    ┌──────┴───────┐                            │
                   (    PUT1:       )                           │
                    └──────┬───────┘                  ┌─────────┴─────────┐
                    ┌──────┴───────┐                  │  Show 'Unable     │
                    │Initialize disk│                 │   to open'        │
                    └──────┬───────┘                  └─────────┬─────────┘
                    ┌──────┴───────┐                            │
                    │Initialize FCB│                            │
                    └──────┬───────┘                            │
                           │                                    │
                        ╱──┴──╲      yes                        │
                       ╱ File  ╲──────────────┐                 │
                       ╲exists?╱              │                 │
                        ╲─┬─╱                 │                 │
                          no              ╱───┴───╲     no       │
                           │             ╱  OK to  ╲─────────────┤
                           │             ╲ erase?  ╱             │
                           │              ╲───┬───╱              ▼
                           │                 yes         ┌──────────────┐
                           │          ┌───────┴──────┐  (   PUTEX:       )
                           │          │ Delete file  │   └──────┬───────┘
                           │          └───────┬──────┘   ┌──────┴───────┐
                    ┌──────┴───────┐          │          │   CR, LF     │
                   (    PUT2:       )◄─────────┘          └──────┬───────┘
                    └──────┬───────┘                     ┌──────┴───────┐
                    ┌──────┴───────┐                     │ Restore DMA  │
                    │ Make a file  │                     │   address    │
                    └──────┬───────┘                     └──────┬───────┘
                           │                                 ╱──┴──╲
                        ╱──┴──╲      no                     ( Return )
                       ╱ Make  ╲──────────────┐             ╲──────╱
                       ╲  OK?  ╱              │
                        ╲─┬─╱                 │
                          yes                 │
                        ╱─┴─╲         ┌───────┴──────┐
                       ( PUT3 )       │ Show 'Unable │
                        ╲───╱         │  to open'    │
                                      └──────────────┘
```

FIGURE 17-1. The flowchart of subroutine PUT:. This routine is complex enough to require the use of a flowchart to insure understanding of all of the optional paths within the routine. Simpler and more straightforward programs can be followed through their listings without the need for a flowchart.
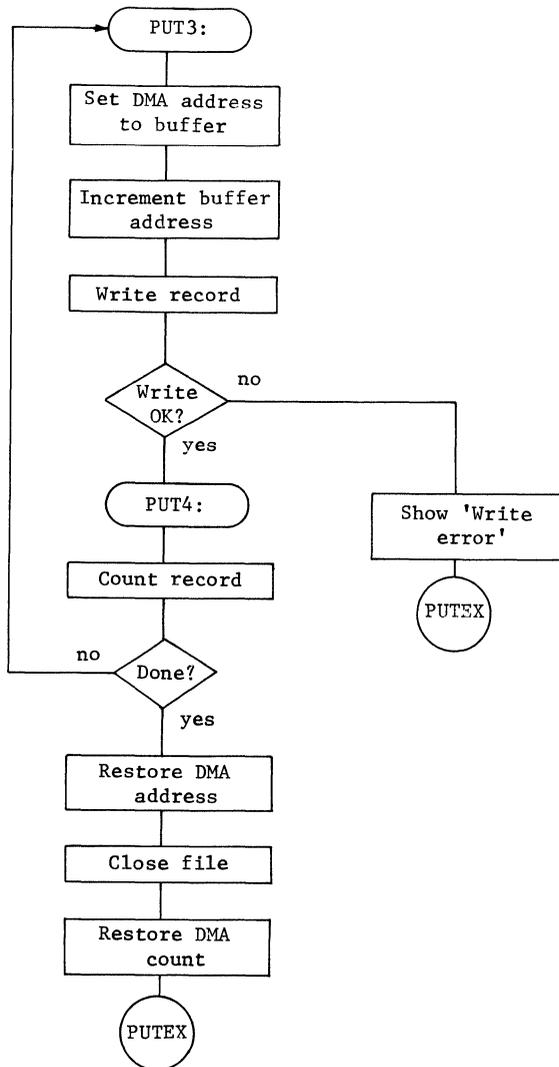
for read or write. With the selected disk now guaranteed to be
R/W, the routine next zeros out the record counters and extension
bytes in TFCB. Next it looks for a file of the same name on the disk
to be written.

CP/M's own PIP program will copy a file from one disk to
another, and automatically overwrite a file of the same name on the
output disk. Our program wants the operator to reassure it that it is
OK to wipe out any pre-existing files of the same name. If one is
encountered, the operator is asked to OK its erasure. Note that in
implementing this single display line:

OK TO ERASE C:FILENAME.TYP (Y/N)?:

we make use of subroutines SPMSG:, SHOFN:, and GETYN: to
display a message, the file name, and a prompt; and to receive the
operator's response to the prompt. All accomplished with only four
lines of assembly language code. Using subroutines saves a lot of
programming effort, once they are debugged.

If an existing file is to be erased, another BDOS function is
called upon to plug that 0E5H into the first byte of that file's
directory entry, thus "erasing" the file. In this case, as opposed to
the operator's use of the CCP command ERA, that directory entry
will probably be immediately overwritten anyway, by the next
BDOS call to make a new file.

The MAKEF function immediately creates a directory entry
on the disk, but since no records have been written, the TFCB disk
allocation map (d0 through dF in Fig. 15–1) is empty, and such a
file would be displayed by STAT as being of zero length, with a
zero record count. A "file" can be less than 1K bytes long under
CP/M, but only an empty one.

Since creating a file entry in the directory involves a write
operation, it is possible for the MAKEF call to produce an error
return from BDOS. One reason could be that the disk is write
protected physically, by the notch in the cover provided for this
purpose. This would cause a failure-to-write error message, as
would an actual write error. In either case we can't write the file,
so an appropriate message is displayed by the call to subroutine
WROPN (List. 16–2). Then we give up on this disk and return to
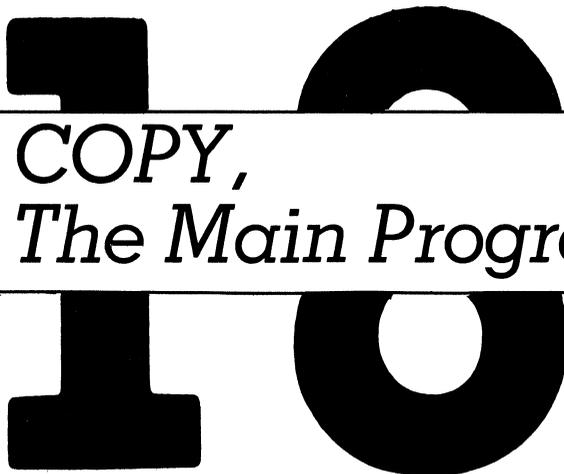the main program.

But if all is well at this point, the actual writing of the file now

takes place in the loop starting at PUT3:. This is similar to the read loop in GET:, except that the record count is decremented instead of incremented, and the loop terminates when the count gets to zero. Or when there is a write error, or we run out of disk space. In any case, successful write or error, the DMA address must be restored to TBUFF, just as was discussed in the last chapter.

   If the write was successful, the file is closed (CLOSF) and the original record count restored, in case another copy is required. Closing the file, of course, writes the TFCB image to disk as a directory entry for the file, overwriting the empty entry created by MAKEF.

Subroutines do it all

With all the action taking place within GET:, PUT:, and all those little subroutines, there isn't much left for a main program to do, as we will be seeing in the next chapter. When you have keyed in the text of PUT.LIB exactly as shown in List. 17–1, you will be ready to add the main program, and see how she runs. I won't stand in your way much longer. Get with it.

COPY,
The Main Program

Now at last it is time to put all the pieces together and create a complete program. COPY.ASM will be a merger of all the .LIB files (except IOEQU.LIB) shown in Table 15–1. They will be merged in the order shown in that table, and the complete program will then be assembled and tested.

The resulting program will be a useful utility, providing a more operator-oriented interface than does the much more powerful (and larger) PIP utility provided with your CP/M system. PIP is not being replaced. It provides greater flexibility than COPY, and also includes useful data conversion options. In the examples in this book, we have been using PIP for very simple file copying and merging operations. You can learn about the other features that PIP provides by studying the Digital Research manuals.

Operating COPY will be easier than using PIP, since all the computer user needs to enter is:

COPY FILENAME.TYP

in the simplest case, and the program will read in the named file and then prompt the operator for the output disk drive selection. Prompting and error messages have been used unsparingly in COPY, since there was no need to try to keep the program small.

The smallest block of disk space that can be allocated is 1K bytes, and since COPY.COM is much smaller than that, there was no effort made to simplify or shorten operator messages. The result is a program that can be operated by anyone, even a programmer.

COPY.LIB is the main program

If you haven't worn ED.COM completely off your disk by now, you can enter the text of the main program shown in List. 18-1. As you have already heard, the "main" program is short, because the subroutines do all the work. Since you already know what all the subroutines do, the logic of COPY.LIB should be obvious from the listing.

LISTING 18-1. COPY.LIB

```
        ORG     TPA                 ; ASSEMBLE PROGRAM FOR TPA

START:  LXI     SP,STAK             ; SET UP USER'S STACK
        LDA     DRIVE               ; SAVE INITIAL DRIVE SELECTED
        STA     DRSAV
START1: CALL    CCRLF               ; START A NEW LINE
        LXI     H,SINON             ; WITH SIGN-ON MESSAGE
        CALL    COMSG
        CALL    GET                 ; GET THE NAMED FILE
START2: CALL    CCRLF               ; BEGIN WRITE PORTION
START3: CALL    SPMSG               ; GET DRIVE FOR WRITE
        DB      'SELECT DRIVE FOR OUTPUT: ',0
        CALL    DRSEL               ; GET A VALID DRIVE SELECT
        JZ      START3              ; IF NO GOOD, TRY AGAIN
        STA     TFCB                ; SET DESIGNATOR INTO FCB
        CALL    CCRLF               ; AND ECHO ACCEPTANCE
        CALL    PUT                 ; WRITE THE FILE TO DISK
        CALL    SPMSG               ; PERMIT MORE WRITES
        DB      'WANT ANOTHER COPY',0
        CALL    GETYN
        JZ      START2              ; LOOP FOR MORE WRITES
DONE:   LDA     DRSAV               ; RESTORE INITIAL DRIVE
        STA     DRIVE
        JMP     RBOOT
```

The program is straightforward. First the stack pointer is initialized, and the current drive selection is saved, so that when all the copying is done the current disk drive will be returned to for the next CCP prompt.

These "housekeeping" chores completed, COPY then signs-on with the message text that is part of RAM.LIB, and then looks for the file to be copied. When the file has been read into the buffer space, the program prompts the operator for the write disk drive selection. The drive selection input subroutine DRSEL:, part of DISKSU.LIB, uses the buffered line input provided by CP/M, so the operator can abort the whole operation at this point with a CTRL C, if need be. Another friendly feature.

If an illegal drive was specified, the program loops back to START3: and repeats the prompt. And it will make you do it over and over until you get it right. Or give up with CTRL C. When you do get it right, the program calls PUT: and writes the file to disk. Whether or not any write errors occur, the operator is next asked if another copy is desired. If so, back to START2: we go, and a new drive can be specified.

When the copying is over, the program exits through DONE:, restoring the original current drive selection and rebooting CP/M. DONE: is also used by the error exit from GET:, since if the file to be copied cannot be read, there is no need to continue the program. During writing, a bad disk will cause a write error message, but the program will continue, giving the operator the chance to replace the disk and try again.

Computers can be friends

"Ergonomics" is the current buzzword refering to making your computer a friendly place to work. This used to be called human engineering, but that sounds too Frankensteinish. Buzzwords proliferate in the computer world; "buzzword" is one of them. No matter what you call it, you should always try to write programs that interact with the operator in such a way that the fallible human knows what is going on, and is told what to do next.

COPY includes features that let the operator know what is going on within the program. Aside from prompts and error messages, strategically placed carriage returns and line feeds are included to show the progress of the program. You may have noticed

lots of CALL CCRLFs in the listings. They provide responses to operator inputs, and mark progress through the program, as well as providing places for you to set traps during the debugging process.

Put it all together and go

You are about to solo as an assembly language programmer, and so, like the little bird being kicked out of the nest or the new pilot being sent off into the sky all alone, you shouldn't expect much more hand-holding from this point on. You are probably ready for that freedom, anyway.

Key in the text of List. 18–1 as COPY.LIB, and then use PIP to merge all the .LIB files into the assembly source program:

PIP COPY.ASM=DISKEQU.LIB,COPY.LIB,GET.LIB,......RAM.LIB

which, when assembled, should produce a COPY.PRN file identical to that shown in List. 18–2. The complete print file has been included here to help you in getting your version running.

No new debugging techniques are needed. If ASM didn't find any errors to report, you may have gotten it all right the first try. Don't take any chances, though. COPY.COM is an assembly language program that includes disk accesses. The combination can be fatal to your valuable programs. Start your debugging with a couple of disks that you can afford to have wiped out. Just in case.

The best way to test COPY is to use it to make a copy of one of your .LIB files. Since they are ASCII files, you can TYPE the file and see if it got copied correctly. With COPY.COM and some ASCII file on a "scratch" disk, you might start your debugging session by simply making a copy from the scratch disk back to itself. This will exercise most of the features of the program.

If your first try fails, it is time to start setting traps and seeing at what point the first error occurs. Use DDT to trap each step in the main program. If a program error results in a complete "bomb," you will have discovered the offending subroutine, since they are all called in sequence. Setting traps, examining registers, and dumping memory contents are all the tools you need. Just check out each part of the program, one step at a time.

A more complete test will involve copying longer files from one disk drive to another. When you get COPY running to the

point that it appears to be working completely, use it to copy COPY.ASM, ASM.COM, and LOAD.COM onto a blank disk. If you can then successfully assemble, load, and run the program from that disk, you can feel confident that COPY is up and running.

Exercises, experiments, and future projects

COPY has its limits. The record count maintained in RAM is only a single eight-bit value, limiting the size of file that can be copied. On your own, now, you might fix that by making it a 16 bit counter. Simpler changes to the program might be made to help you exercise your new powers of programming prowess. Add progress messages to the program, like "READING FILENAME.TYP" and "WRITING ON DRIVE x:," for example.

When you have done all the damage to COPY that you can think of, use the library subroutines to write a file COMPARE program. Read in one file, just as in copy. Then read the second file one record at a time, using TBUFF to store the records from the second file. Then compare the record in TBUFF against the appropriate portion of the file in BUFFR, and display any differences. If you confine this program to comparing ASCII files, the comparisons are easy to make and display.

With your knowledge of the internal structure of CP/M, you are well equipped to try modifying CBIOS. Adding drivers for new peripherals is usually the first task for a modified CBIOS. This will require further studying of the Digital Research manuals and the sample programs they contain. Those sample programs are a good place to start examining and analyzing other programmer's work, to enhance your understanding of assembly language programming, and to provide a source of usable techniques. The DUMP.ASM program supplied with CP/M is another excellent source of usable techniques.

On your own, now

It would not have been possible to include every aspect of 8080 assembly language programming in this book, so no effort was made to do so. Some topics, like decimal arithmetic, have been

ignored because of space limitations. What has been included, instead, is a picture of how computer hardware, operating systems, and user assembly language programs interact. This provides you with the background that, combined with your CP/M based computer, will allow you to continue the learning process on your own.

CP/M provides a friendly environment and the powerful tools necessary to permit that learning to continue.

LISTING 18-2. The complete result of all the programming effort: COPY.PRN lists the entire program after assembly.

```
              ; MULTI-WRITE FILE COPY PROGRAM  12 SEPT 82

              ; ASCII CHARACTERS
000D =        CR      EQU    0DH              ; CARRIAGE RETURN
000A =        LF      EQU    0AH              ; LINE FEED
001A =        CTRLZ   EQU    1AH              ; OPERATOR INTERRUPT

              ; CP/M BDOS FUNCTIONS
0001 =        RCONF   EQU    1                ; READ CON: INTO (A)
0002 =        WCONF   EQU    2                ; WRITE (A) TO CON:
000A =        RBUFF   EQU    10               ; READ A CONSOLE LINE

              ; CP/M DISK ACCESS FUNCTIONS
000D =        INITF   EQU    13               ; INITIALIZE BDOS FUNCTION
000F =        OPENF   EQU    15               ; OPEN FILE FUNCTION
0010 =        CLOSF   EQU    16               ; CLOSE FILE FUNCTION
0011 =        FINDF   EQU    17               ; FIND FILE FUNCTION
0013 =        DELEF   EQU    19               ; DELETE A FILE FUNCTION
0014 =        READF   EQU    20               ; READ ONE RECORD FUNCTION
0015 =        WRITF   EQU    21               ; WRITE ONE RECORD FUNCTION
0016 =        MAKEF   EQU    22               ; CREATE FILE FUNCTION
001A =        SDMAF   EQU    26               ; SET DMA FUNCTION

              ; CP/M ADDRESSES
0000 =        RBOOT   EQU    0                ; RE-BOOT CP/M SYSTEM
0004 =        DRIVE   EQU    4                ; CURRENT DRIVE SELECTION
0005 =        BDOS    EQU    5                ; SYSTEM CALL ENTRY
0007 =        MEMAX   EQU    7                ; MSB OF TOP OF MEMORY
005C =        TFCB    EQU    5CH              ; TRANSIENT FILE CONTROL BLOCK
0065 =        FCBTY   EQU    TFCB+9           ; FILE TYPE IN FCB
0068 =        FCBEX   EQU    TFCB+12          ; FILE EXTENT IN FCB
006A =        FCBS2   EQU    TFCB+14          ; SYSTEM USE IN FCB
006B =        FCBRC   EQU    TFCB+15          ; RECORD COUNT IN FCB
007C =        FCBCR   EQU    TFCB+32          ; CURRENT RECORD IN FCB
0080 =        TBUFF   EQU    80H              ; TRANSIENT BUFFER
0100 =        TPA     EQU    100H             ; TRANSIENT PROGRAM AREA

              ; CP/M FLAGS
0000 =        BDAOK   EQU    0                ; BDOS RETURN FOR ALL OK
0001 =        BDER1   EQU    1                ; BDOS RETURN ONE
0002 =        BDER2   EQU    2                ; BDOS RETURN TWO
00FF =        BDERR   EQU    255              ; BDOS RETURN ERROR FLAG
```

LISTING 18-2. *Continued*

```
0100                    ORG    TPA              ; ASSEMBLE PROGRAM FOR TPA

0100 317104     START:  LXI    SP,STAK          ; SET UP USER'S STACK
0103 3A0400             LDA    DRIVE            ; SAVE INITIAL DRIVE SELECTED
0106 322C04             STA    DRSAV
0109 CD7703     START1: CALL   CCRLF            ; START A NEW LINE
010C 217204             LXI    H,SINON          ; WITH SIGN-ON MESSAGE
010F CD8103             CALL   COMSG
0112 CD6801             CALL   GET              ; GET THE NAMED FILE
0115 CD7703     START2: CALL   CCRLF            ; BEGIN WRITE PORTION
0118 CDAA03     START3: CALL   SPMSG            ; GET DRIVE FOR WRITE
011B 53454C4543         DB     'SELECT DRIVE FOR OUTPUT: ',0
0135 CD4203             CALL   DRSEL            ; GET A VALID DRIVE SELECT
0138 CA1801             JZ     START3           ; IF NO GOOD, TRY AGAIN
013B 325C00             STA    TFCB             ; SET DESIGNATOR INTO FCB
013E CD7703             CALL   CCRLF            ; AND ECHO ACCEPTANCE
0141 CDF401             CALL   PUT              ; WRITE THE FILE TO DISK
0144 CDAA03             CALL   SPMSG            ; PERMIT MORE WRITES
0147 57414E5420         DB     'WANT ANOTHER COPY',0
0159 CDB603             CALL   GETYN
015C CA1501             JZ     START2           ; LOOP FOR MORE WRITES
015F 3A2C04     DONE:   LDA    DRSAV            ; RESTORE INITIAL DRIVE
0162 320400             STA    DRIVE
0165 C30000             JMP    RBOOT

                ; READ A FILE FROM DISK INTO "BUFFR"

0168 219404     GET:    LXI    H,BUFFR          ; GET BUFFER START
016B 222F04             SHLD   NEXT             ;   ADDRESS FOR DMA
016E 115C00             LXI    D,TFCB           ; SEE IF FILE IS ON DISK
0171 0E0F               MVI    C,OPENF          ; AND OPEN FOR READ
0173 CD0500             CALL   BDOS
0176 FEFF               CPI    BDERR            ; IS IT THERE?
0178 C29801             JNZ    GET1             ; YES, READ IT IN
017B CD7403             CALL   TWOCR            ; NO, SHOW ERROR
017E CDAA03             CALL   SPMSG
0181 43414E204E         DB     'CAN NOT FIND ',0
018F CDAB02             CALL   SHOFN            ; SHOW FILE NAME
0192 CD7403     ERREX:  CALL   TWOCR            ; ERROR EXIT TO CP/M
0195 C35F01             JMP    DONE
0198 AF         GET1:   XRA    A                ; ZERO RECORD COUNTER
0199 322D04             STA    RECCT            ; AND READ A FILE INTO BUFFR
019C 2A2F04     GET2:   LHLD   NEXT             ; SET BUFFER ADDRESS
019F EB                 XCHG
01A0 0E1A               MVI    C,SDMAF
01A2 CD0500             CALL   BDOS
01A5 115C00             LXI    D,TFCB           ; READ ONE RECORD INTO
01A8 0E14               MVI    C,READF          ; BUFFER
01AA CD0500             CALL   BDOS
01AD FE00               CPI    BDAOK            ; READ OK?
01AF CABD01             JZ     GET3             ; YES, DO MORE
01B2 FE01               CPI    BDER1            ; MAYBE, END OF FILE?
01B4 CAED01             JZ     GETEX            ; YES, NO PROBLEM
```

LISTING 18–2. *Continued*

```
01B7 CDDC02              CALL    REMSG           ; NO, SHOW ERROR
01BA C39201              JMP     ERREX           ; AND ALL DONE

01BD 3A2D04     GET3:    LDA     RECCT           ; COUNT THE RECORD
01C0 3C                  INR     A
01C1 322D04              STA     RECCT
01C4 2A2F04              LHLD    NEXT            ; INCREMENT BUFFER ADDRESS
01C7 118000              LXI     D,128           ; BY RECORD SIZE
01CA 19                  DAD     D
01CB 222F04              SHLD    NEXT
01CE 3A0700              LDA     MEMAX           ; ROOM LEFT IN RAM?
01D1 3D                  DCR     A               ; STOP BELOW CCP
01D2 BC                  CMP     H               ; COMPARE MSB
01D3 C29C01              JNZ     GET2            ; CONTINUE IF NOT EQUAL
01D6 CD7403              CALL    TWOCR           ; ELSE SHOW OUT OF MEMORY
01D9 CDAA03              CALL    SPMSG
01DC 4F5554204F          DB      'OUT OF MEMORY',0
01EA C39201              JMP     ERREX           ; AND GIVE UP

01ED CD7703     GETEX:   CALL    CCRLF           ; NORMAL EXIT
01F0 CD3903              CALL    CPDMA           ; RESTORE CP/M DMA
01F3 C9                  RET

                ; WRITE A FILE FROM "BUFFR" TO DISK

01F4 219404     PUT:     LXI     H,BUFFR         ; SET UP BUFFER START
01F7 222F04              SHLD    NEXT
01FA 3A2D04              LDA     RECCT           ; SAVE RECORD COUNT
01FD 322E04              STA     CTSAV
0200 3A5C00              LDA     TFCB            ; LOG-IN SELECTED DISK
0203 B7                  ORA     A               ; IS IT LEGAL?
0204 C20D02              JNZ     PUT1
0207 CD1903              CALL    WROPN           ; NO, SHOW UNABLE TO OPEN
020A C3A402              JMP     PUTEX           ; AND TRY AGAIN
020D 0E0D       PUT1:    MVI     C,INITF         ; ENABLE WRITE ON ANY DISK
020F CD0500              CALL    BDOS
0212 AF                  XRA     A               ; INITIALIZE FCB
0213 327C00              STA     FCBCR           ; CURRENT RECORD
0216 210000              LXI     H,0
0219 226800              SHLD    FCBEX           ; EXTENT AND S1
021C 226A00              SHLD    FCBS2           ; S2 AND RECORD COUNT
021F 115C00              LXI     D,TFCB          ; SEE IF FILE EXISTS
0222 0E11                MVI     C,FINDF         ; FIND FUNCTION
0224 CD0500              CALL    BDOS
0227 FEFF                CPI     BDERR           ; IS IT ALREADY?
0229 CA5002              JZ      PUT2
022C CD7703              CALL    CCRLF           ; YES, OK TO ERASE?
022F CDAA03              CALL    SPMSG
0232 4F4B20544F          DB      'OK TO ERASE ',0
023F CDAB02              CALL    SHOFN
0242 CDB603              CALL    GETYN
0245 C2A402              JNZ     PUTEX           ; IF NO, TRY AGAIN
0248 115C00              LXI     D,TFCB          ; IF YES, ERASE IT
```

LISTING 18–2. *Continued*

```
024B 0E13              MVI    C,DELEF       ; DELETE FUNCTION
024D CD0500            CALL   BDOS
0250 115C00    PUT2:   LXI    D,TFCB        ; OPEN FILE FOR WRITE
0253 0E16              MVI    C,MAKEF       ; MAKE A FILE FUNCTION
0255 CD0500            CALL   BDOS
0258 FEFF              CPI    BDERR         ; GOT IT MADE?
025A C26302            JNZ    PUT3
025D CD1903            CALL   WROPN         ; NO, SHOW UNABLE
0260 C3A402            JMP    PUTEX         ; AND TRY AGAIN

0263 2A2F04    PUT3:   LHLD   NEXT          ; WRITE BUFFER TO DISK
0266 EB                XCHG                 ; (FINALLY)
0267 0E1A              MVI    C,SDMAF       ; SET ADDRESS TO WRITE FROM
0269 CD0500            CALL   BDOS
026C 2A2F04            LHLD   NEXT          ; THEN INCREMENT BY 128
026F 118000            LXI    D,128
0272 19                DAD    D
0273 222F04            SHLD   NEXT
0276 115C00            LXI    D,TFCB
0279 0E15              MVI    C,WRITF       ; WRITE A RECORD FUNCTION
027B CD0500            CALL   BDOS
027E FE00              CPI    BDAOK         ; WRITE OK?
0280 CA8902            JZ     PUT4
0283 CDFA02            CALL   WEMSG         ; NO, WRITE ERROR MESSAGE
0286 C3A402            JMP    PUTEX         ; AND TRY AGAIN
0289 3A2D04    PUT4:   LDA    RECCT         ; COUNT THE RECORD
028C 3D                DCR    A
028D 322D04            STA    RECCT         ; IF NOT END OF RECORDS
0290 C26302            JNZ    PUT3          ; THEN DO ANOTHER
0293 CD3903            CALL   CPDMA         ; ELSE RESTORE CP/M DMA
0296 115C00            LXI    D,TFCB
0299 0E10              MVI    C,CLOSF       ; THEN CLOSE THE FILE
029B CD0500            CALL   BDOS
029E 3A2E04            LDA    CTSAV         ; RESTORE RECORD COUNT
02A1 322D04            STA    RECCT         ; FOR NEXT WRITE
02A4 CD7703    PUTEX:  CALL   CCRLF
02A7 CD3903            CALL   CPDMA         ; RESTORE CP/M DMA
02AA C9                RET                  ; AND ALL DONE

               ; DISPLAY FILENAME.TYP FROM TRANSIENT FCB
02AB C5        SHOFN:  PUSH   B             ; SAVE TEMP STORE
02AC E5                PUSH   H             ; AND INDEX
02AD 3A6500            LDA    FCBTY         ; SAVE FIRST CHAR OF TYPE
02B0 4F                MOV    C,A           ; IN TEMPORARY STORE
02B1 AF                XRA    A             ; FORCE TWO TERMINATORS
02B2 326500            STA    FCBTY         ; FOR FILE NAME
02B5 326800            STA    FCBEX         ; AND FILE TYPE
02B8 215C00            LXI    H,TFCB        ; SHOW DISK DRIVE
02BB 7E                MOV    A,M
02BC E60F              ANI    0FH           ; LIMIT TO 4 BITS
02BE F640              ORI    40H           ; CONVERT TO ASCII
02C0 CD6703            CALL   CO
02C3 3E3A              MVI    A,':'         ; SHOW THE COLON
```

LISTING 18–2. *Continued*

```
02C5 CD6703          CALL    CO
02C8 23              INX     H                       ; AND SHOW THE FILE NAME
02C9 CD8103          CALL    COMSG
02CC 79              MOV     A,C
02CD 216500          LXI     H,FCBTY                 ; RESTORE TYPE
02D0 77              MOV     M,A
02D1 3E2E            MVI     A,'.'                   ; SHOW SEPARATOR
02D3 CD6703          CALL    CO
02D6 CD8103          CALL    COMSG                   ; SHOW TYPE
02D9 E1              POP     H
02DA C1              POP     B                       ; RESTORE AND
02DB C9              RET                             ; RETURN

                     ; DISPLAY READ ERROR MESSAGE
02DC CD7403   REMSG: CALL    TWOCR
02DF CDAA03          CALL    SPMSG
02E2 5045524D41      DB      'PERMANENT READ ERROR',CR,LF,0
02F9 C9              RET

                     ; DISPLAY WRITE ERROR MESSAGE
02FA CD7403   WEMSG: CALL    TWOCR
02FD CDAA03          CALL    SPMSG
0300 5045524D41      DB      'PERMANENT WRITE ERROR',CR,LF,0
0318 C9              RET

                     ; DISPLAY WRITE OPEN ERROR MESSAGE
0319 CD7403   WROPN: CALL    TWOCR
031C CDAA03          CALL    SPMSG
031F 43414E204E      DB      'CAN NOT OPEN FOR WRITE',CR,LF,0
0338 C9              RET

                     ; RESTORE CP/M DMA ADDRESS TO THE TRANSIENT BUFFER
0339 118000   CPDMA: LXI     D,TBUFF
033C 0E1A            MVI     C,SDMAF
033E CD0500          CALL    BDOS
0341 C9              RET

                     ; GET A VALID DRIVE SELECT DESIGNATOR
0342 CD8B03   DRSEL: CALL    CIMSG                   ; INPUT THE SELECTION
0345 3ADB03          LDA     INBUF+2                 ; USE FIRST CHARACTER ONLY
0348 E65F            ANI     01011111B               ; CONVERT TO UPPER CASE
034A D640            SUI     '@'                     ; SET A=1, B=2, ETC.
034C FA5703          JM      DRERR                   ; CAN'T BE LESS THAN ZERO
034F D611            SUI     17                      ; OR GREATER THAN 16
0351 F25703          JP      DRERR
0354 C611            ADI     17                      ; RESTORE LEGAL NUMBER
0356 C9              RET                             ; AND RETURN WITH IT
0357 AF       DRERR: XRA     A                       ; ELSE SET ZERO FLAG
0358 C9              RET                             ; AND RETURN

                     ; CONSOLE CHARACTER INTO REGISTER A MASKED TO 7 BITS
0359 C5       CI:    PUSH    B                       ; SAVE REGISTERS
035A D5              PUSH    D
```

LISTING 18–2. *Continued*

```
035B E5                         PUSH    H
035C 0E01                       MVI     C,RCONF         ; READ FUNCTION
035E CD0500                     CALL    BDOS
0361 E67F                       ANI     7FH             ; MASK TO 7 BITS
0363 E1                         POP     H               ; RESTORE REGISTERS
0364 D1                         POP     D
0365 C1                         POP     B
0366 C9                         RET

                        ; CHARACTER IN REGISTER A OUTPUT TO CONSOLE
0367 C5                 CO:     PUSH    B               ; SAVE REGISTERS
0368 D5                         PUSH    D
0369 E5                         PUSH    H
036A 0E02                       MVI     C,WCONF         ; SELECT FUNCTION
036C 5F                         MOV     E,A             ; CHARACTER TO E
036D CD0500                     CALL    BDOS            ; OUTPUT BY CP/M
0370 E1                         POP     H               ; RESTORE REGISTERS
0371 D1                         POP     D
0372 C1                         POP     B
0373 C9                         RET

                        ; CARRIAGE RETURN, LINE FEED TO CONSOLE
0374 CD7703             TWOCR:  CALL    CCRLF
0377 3E0D               CCRLF:  MVI     A,CR
0379 CD6703                     CALL    CO
037C 3E0A                       MVI     A,LF
037E C36703                     JMP     CO

                        ; MESSAGE POINTED TO BY HL OUT TO CONSOLE
0381 7E                 COMSG:  MOV     A,M             ; GET A CHARACTER
0382 B7                         ORA     A               ; ZERO IS THE TERMINATOR
0383 C8                         RZ                      ; RETURN ON ZERO
0384 CD6703                     CALL    CO              ; ELSE OUTPUT THE CHARACTER
0387 23                         INX     H               ; POINT TO THE NEXT ONE
0388 C38103                     JMP     COMSG           ; AND CONTINUE

                        ; INPUT CONSOLE MESSAGE INTO BUFFER
038B C5                 CIMSG:  PUSH    B               ; SAVE REGISTERS
038C D5                         PUSH    D
038D E5                         PUSH    H
038E 21DA03                     LXI     H,INBUF+1       ; ZERO CHARACTER COUNTER
0391 3600                       MVI     M,0
0393 2B                         DCX     H               ; SET MAXIMUM LINE LENGTH
0394 3650                       MVI     M,80
0396 EB                         XCHG                    ; INBUF POINTER TO DE PAIR
0397 0E0A                       MVI     C,RBUFF         ; SET UP READ BUFFER FUNCTION
0399 CD0500                     CALL    BDOS            ; INPUT A LINE
039C 21DA03                     LXI     H,INBUF+1       ; GET CHARACTER COUNTER
039F 5E                         MOV     E,M             ; INTO LSB OF DE REGISTER PAIR
03A0 1600                       MVI     D,0             ; ZERO MSB
03A2 19                         DAD     D               ; ADD LENGTH TO START
03A3 23                         INX     H               ; PLUS ONE POINTS TO END
03A4 3600                       MVI     M,0             ; INSERT TERMINATOR AT END
```

LISTING 18-2. *Continued*

```
03A6 E1                     POP    H              ; RESTORE ALL REGISTERS
03A7 D1                     POP    D
03A8 C1                     POP    B
03A9 C9                     RET

              ; MESSAGE POINTED TO BY STACK OUT TO CONSOLE
03AA E3          SPMSG:     XTHL                  ; GET "RETURN ADDRESS" TO HL
03AB AF                     XRA    A              ; CLEAR FLAGS AND ACCUMULATOR
03AC 86                     ADD    M              ; GET ONE MESSAGE CHARACTER
03AD 23                     INX    H              ; POINT TO NEXT
03AE E3                     XTHL                  ; RESTORE STACK FOR
03AF C8                     RZ                    ; RETURN IF DONE
03B0 CD6703                 CALL   CO             ; ELSE DISPLAY CHARACTER
03B3 C3AA03                 JMP    SPMSG          ; AND DO ANOTHER

              ; GET YES OR NO FROM CONSOLE
03B6 CDAA03      GETYN:     CALL   SPMSG          ; PROMPT FOR INPUT
03B9 2028592F4E             DB     ' (Y/N)?: ',0
03C3 CD8B03                 CALL   CIMSG          ; GET INPUT LINE
03C6 CD7703                 CALL   CCRLF          ; ECHO CARRIAGE RETURN
03C9 3ADB03                 LDA    INBUF+2        ; FIRST CHARACTER ONLY
03CC E65F                   ANI    01011111B      ; CONVERT LOWER CASE TO UPPER
03CE FE59                   CPI    'Y'            ; RETURN WITH ZERO = YES
03D0 C8                     RZ
03D1 FE4E                   CPI    'N'            ; NON-ZERO = NO
03D3 C2B603                 JNZ    GETYN          ; ELSE TRY AGAIN
03D6 FE00                   CPI    0              ; RESET ZERO FLAG
03D8 C9                     RET                   ; AND ALL DONE

              ; RAM VARIABLES AND BUFFERS
03D9             INBUF:     DS     83             ; LINE INPUT BUFFER
042C             DRSAV:     DS     1              ; CURRENT DRIVE AT ENTRY
042D             RECCT:     DS     1              ; TOTAL RECORDS READ/TO WRITE
042E             CTSAV:     DS     1              ; SAVE LOCATION FOR COUNT
042F             NEXT:      DS     2              ; NEXT DMA ADDRESS

              ; SET UP STACK SPACE
0431                        DS     64             ; 40H LOCATIONS
0471 00          STAK:      DB     0              ; TOP OF STACK

0472 4D554C5449SINON:       DB     'MULTI-WRITE FILE COPY  12 SEPT 82',0

              ; FROM HERE THROUGH CCP IS BUFFER SPACE
                 BUFFR:
0494                        END
```
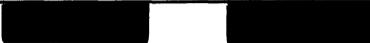
APPENDIXES

American Standard Code for Information Interchange (ASCII)

Nonprinting (control) codes

Control	Key-stroke(s)	Hex code	Function—Standard Usage (CP/M, BASIC, etc. usage may differ)
NUL	CTRL @	00	Null (Blank)
SOH	CTRL A	01	Start of header message
STX	CTRL B	02	Start of message text
ETX	CTRL C	03	End of message text
EOT	CTRL D	04	End of transmission
ENQ	CTRL E	05	Enquiry, i.e., "Did you get that?"
ACK	CTRL F	06	Acknowledge, i.e., "Yes I did."
BEL	CTRL G	07	Bell, operator alert
BS	CTRL H/BS	08	Backspace one character
HT	CTRL I/TAB	09	Horizontal tab
LF	CTRL J/LF	0A	Line feed
VT	CTRL K	0B	Vertical tab
FF	CTRL L	0C	Form feed
CR	CTRL M/CR	0D	Carriage return
SO	CTRL N	0E	Shift Out to alternate character set
SI	CTRL O	0F	Shift In, back to standard characters
DLE	CTRL P	10	Data Link Escape to alternate functions
DC1	CTRL Q	11	Device Control 1, user optional meaning
DC2	CTRL R	12	Device Control 2, as above
DC3	CTRL S	13	Device Control 3, as above
DC4	CTRL T	14	Device Control 4, as above
NAK	CTRL U	15	No Acknowledge, i.e., "I didn't get that."
SYN	CTRL V	16	Synchronization character
ETB	CTRL W	17	End of transmitted block
CAN	CTRL X	18	Cancel previous message
EM	CTRL Y	19	End of message
SUB	CTRL Z	1A	Substitute incorrect character
ESC	CTRL [	1B	Escape to alternate functions
FS	CTRL \	1C	File separator
GS	CTRL]	1D	Group separator
RS	CTRL ↑	1E	Record separator
US	CTRL __	1F	Unit separator
DEL	DEL	7F	Delete, or rubout

Printing (character) codes

Key	Hex code	Key	Hex code	Key	Hex code
SPACE	20	@	40		60
!	21	A	41	à	61
"	22	B	42	b	62
#	23	C	43	c	63
$	24	D	44	d	64
%	25	E	45	e	65
&	26	F	46	f	66
'	27	G	47	g	67
(	28	H	48	h	68
)	29	I	49	i	69
*	2A	J	4A	j	6A
+	2B	K	4B	k	6B
,	2C	L	4C	l	6C
−	2D	M	4D	m	6D
.	2E	N	4E	n	6E
/	2F	O	4F	o	6F
0	30	P	50	p	70
1	31	Q	51	q	71
2	32	R	52	r	72
3	33	S	53	s	73
4	34	T	54	t	74
5	35	U	55	u	75
6	36	V	56	v	76
7	37	W	57	w	77
8	38	X	58	x	78
9	39	Y	59	y	79
:	3A	Z	5A	z	7A
;	3B	[	5B	{	7B
<	3C	\	5C	¦	7C
=	3D	]	5D	}	7D
>	3E	↑	5E	~	7E
?	3F	_	5F		

B

8080 Instruction Set

Functional listing
Single byte instructions—No flags affected—

Opcode	Hex	Opcode	Hex	Opcode	Hex
MOV B,B	40	MOV E,B	58	MOV M,B	70
MOV B,C	41	MOV E,C	59	MOV M,C	71
MOV B,D	42	MOV E,D	5A	MOV M,D	72
MOV B,E	43	MOV E,E	5B	MOV M,E	73
MOV B,H	44	MOV E,H	5C	MOV M,H	74
MOV B,L	45	MOV E,L	5D	MOV M,L	75
MOV B,M	46	MOV E,M	5E		
MOV B,A	47	MOV E,A	5F	MOV M,A	77
MOV C,B	48	MOV H,B	60	MOV A,B	78
MOV C,C	49	MOV H,C	61	MOV A,C	79
MOV C,D	4A	MOV H,D	62	MOV A,D	7A
MOV C,E	4B	MOV H,E	63	MOV A,E	7B
MOV C,H	4C	MOV H,H	64	MOV A,H	7C
MOV C,L	4D	MOV H,L	65	MOV A,L	7D
MOV C,M	4E	MOV H,M	66	MOV A,M	7E

Opcode	Hex	Opcode	Hex	Opcode	Hex
MOV C,A	4F	MOV H,A	67	MOV A,A	7F
MOV D,B	50	MOV L,B	68	POP B	C1
MOV D,C	51	MOV L,C	69	POP D	D1
MOV D,D	52	MOV L,D	6A	POP H	E1
MOV D,E	53	MOV L,E	6B	POP PSW	F1
MOV D,H	54	MOV L,H	6C	PUSH B	C5
MOV D,L	55	MOV L,L	6D	PUSH D	D5
MOV D,M	56	MOV L,M	6E	PUSH H	E5
MOV D,A	57	MOV L,A	6F	PUSH PSW	F5
RST 0	C7	RET	C9	XTHL	E3
RST 1	CF	RNZ	C0	PCHL	E9
RST 2	D7	RZ	C8	SPHL	F9
RST 3	DF	RNC	D0	XCHG	EB
RST 4	E7	RC	D8	CMA	2F
RST 5	EF	RPO	E0	NOP	00
RST 6	F7	RPE	E8	HLT	76
RST 7	FF	RP	F0	DI	F3
		RM	F8	EI	FB
INX B	03	DCX B	0B	LDAX B	0A
INX D	13	DCX D	1B	LDAX D	1A
INX H	23	DCX H	2B	STAX B	02
INX SP	33	DCX SP	3B	STAX D	12

Single byte instructions—Only Carry is affected—

Opcode	Hex	Opcode	Hex	Opcode	Hex
RLC	07	DAD B	09	STC	37
RRC	0F	DAD D	19	CMC	3F
RAL	17	DAD H	29		
RAR	1F	DAD SP	39		

Single byte instructions—All flags (not Carry) affected

Opcode	Hex	Opcode	Hex
INR B	04	DCR B	05
INR C	0C	DCR C	0D
INR D	14	DCR D	15
INR E	1C	DCR E	1D
INR H	24	DCR H	25
INR L	2C	DCR L	2D
INR M	34	DCR M	35
INR A	3C	DCR A	3D

Opcode	Hex	Opcode	Hex	Opcode	Hex
Single byte instructions—All flags affected					
ADD B	80	SBB B	98	ORA B	B0
ADD C	81	SBB C	99	ORA C	B1
ADD D	82	SBB D	9A	ORA D	B2
ADD E	83	SBB E	9B	ORA E	B3
ADD H	84	SBB H	9C	ORA H	B4
ADD L	85	SBB L	9D	ORA L	B5
ADD M	86	SBB M	9E	ORA M	B6
ADD A	87	SBB A	9F	ORA A	B7
ADC B	88	ANA B	A0	CMP B	B8
ADC C	89	ANA C	A1	CMP C	B9
ADC D	8A	ANA D	A2	CMP D	BA
ADC E	8B	ANA E	A3	CMP E	BB
ADC H	8C	ANA H	A4	CMP H	BC
ADC L	8D	ANA L	A5	CMP L	BD
ADC M	8E	ANA M	A6	CMP M	BE
ADC A	8F	ANA A	A7	CMP A	BF
SUB B	90	XRA B	A8	DAA	27
SUB C	91	XRA C	A9		
SUB D	92	XRA D	AA		
SUB E	93	XRA E	AB		
SUB H	94	XRA H	AC		
SUB L	95	XRA L	AD		
SUB M	96	XRA M	AE		
SUB A	97	XRA A	AF		
Double byte instructions—No flags affected					
MVI B,dd	06	MVI H,dd	26	OUT dd	D3
MVI C,dd	0E	MVI L,dd	2E	IN dd	DB
MVI D,dd	16	MVI M,dd	36		
MVI E,dd	1E	MVI A,dd	3E		
Double byte instructions—All flags affected					
ADI dd	C6	ANI dd	E6		
ACI dd	CE	XRI dd	EE		
SUI dd	D6	ORI dd	F6		
SBI dd	DE	CPI dd	FE		

Opcode	Hex	Opcode	Hex	Opcode	Hex

Triple byte instructions—No flags affected

Opcode	Hex	Opcode	Hex	Opcode	Hex
JMP αααα	C3	CALL αααα	CD	LXI B,αααα	01
JNZ αααα	C2	CNZ αααα	C4	LXI D,αααα	11
JZ αααα	CA	CZ αααα	CC	LXI H,αααα	21
JNC αααα	D2	CNC αααα	D4	LXI SP,αααα	31
JC αααα	DA	CC αααα	DC	LHLD αααα	2A
JPO αααα	E2	CPO αααα	E4	SHLD αααα	22
JPE αααα	EA	CPE αααα	EC	LDA αααα	3A
JP αααα	F2	CP αααα	F4	STA αααα	32
JM αααα	FA	CM αααα	FC		

where dd = 8 bits of data
αααα = 16 bits of (address) data

Numerical listing

Hex	Opcode	Hex	Opcode	Hex	Opcode
00	NOP	56	MOV D,M	AC	XRA H
01	LXI B,αααα	57	MOV D,A	AD	XRA L
02	STAX B	58	MOV E,B	AE	XRA M
03	INX B	59	MOV E,C	AF	XRA A
04	INR B	5A	MOV E,D	B0	ORA B
05	DCR B	5B	MOV E,E	B1	ORA C
06	MVI B,dd	5C	MOV E,H	B2	ORA D
07	RLC	5D	MOV E,L	B3	ORA E
08		5E	MOV E,M	B4	ORA H
09	DAD B	5F	MOV E,A	B5	ORA L
0A	LDAX B	60	MOV H,B	B6	ORA M
0B	DCX B	61	MOV H,C	B7	ORA A
0C	INR C	62	MOV H,D	B8	CMP B
0D	DCR C	63	MOV H,E	B9	CMP C
0E	MVI C,dd	64	MOV H,H	BA	CMP D
0F	RRC	65	MOV H,L	BB	CMP E
10		66	MOV H,M	BC	CMP H
11	LXI D,αααα	67	MOV H,A	BD	CMP L

Hex	Opcode	Hex	Opcode	Hex	Opcode
12	STAX D	68	MOV L,B	BE	CMP M
13	INX D	69	MOV L,C	BF	CMP A
14	INR D	6A	MOV L,D	C0	RNZ
15	DCR D	6B	MOV L,E	C1	POP B
16	MVI D,dd	6C	MOV L,H	C2	JNZ aaaa
17	RAL	6D	MOV L,L	C3	JMP aaaa
18		6E	MOV L,M	C4	CNZ aaaa
19	DAD D	6F	MOV L,A	C5	PUSH B
1A	LDAX D	70	MOV M,B	C6	ADI dd
1B	DCX D	71	MOV M,C	C7	RST 0
1C	INR E	72	MOV M,D	C8	RZ
1D	DCR E	73	MOV M,E	C9	RET
1E	MVI E,dd	74	MOV M,H	CA	JZ aaaa
1F	RAR	75	MOV M,L	CB	
20		76	HLT	CC	CZ aaaa
21	LXI H,aaaa	77	MOV M,A	CD	CALL aaaa
22	SHLD aaaa	78	MOV A,B	CE	ACI dd
23	INX H	79	MOV A,C	CF	RST 1
24	INR H	7A	MOV A,D	D0	RNC
25	DCR H	7B	MOV A,E	D1	POP D
26	MVI H,dd	7C	MOV A,H	D2	JNC aaaa
27	DAA	7D	MOV A,L	D3	OUT dd
28		7E	MOV A,M	D4	CNC aaaa
29	DAD H	7F	MOV A,A	D5	PUSH D
2A	LHLD aaaa	80	ADD B	D6	SUI dd
2B	DCX H	81	ADD C	D7	RST 2
2C	INR L	82	ADD D	D8	RC
2D	DCR L	83	ADD E	D9	
2E	MVI L,dd	84	ADD H	DA	JC aaaa
2F	CMA	85	ADD L	DB	IN dd
30		86	ADD M	DC	CC aaaa
31	LXI SP,aaaa	87	ADD A	DD	
32	STA aaaa	88	ADC B	DE	SBI dd
33	INX SP	89	ADC C	DF	RST 3
34	INR M	8A	ADC D	E0	RPO
35	DCR M	8B	ADC E	E1	POP H
36	MVI M,dd	8C	ADC H	E2	JPO aaaa
37	STC	8D	ADC L	E3	XTHL
38		8E	ADC M	E4	CPO aaaa
39	DAD SP	8F	ADC A	E5	PUSH H

Hex	Opcode	Hex	Opcode	Hex	Opcode
3A	LDA aaaa	90	SUB B	E6	ANI dd
3B	DCX SP	91	SUB C	E7	RST 4
3C	INR A	92	SUB D	E8	RPE
3D	DCR A	93	SUB E	E9	PCHL
3E	MVI A,dd	94	SUB H	EA	JPE aaaa
3F	CMC	95	SUB L	EB	XCHG
40	MOV B,B	96	SUB M	EC	CPE aaaa
41	MOV B,C	97	SUB A	ED	
42	MOV B,D	98	SBB B	EE	XRI dd
43	MOV B,E	99	SBB C	EF	RST 5
44	MOV B,H	9A	SBB D	F0	RP
45	MOV B,L	9B	SBB E	F1	POP PSW
46	MOV B,M	9C	SBB H	F2	JP aaaa
47	MOV B,A	9D	SBB L	F3	DI
48	MOV C,B	9E	SBB M	F4	CP aaaa
49	MOV C,C	9F	SBB A	F5	PUSH PSW
4A	MOV C,D	A0	ANA B	F6	ORI dd
4B	MOV C,E	A1	ANA C	F7	RST 6
4C	MOV C,H	A2	ANA D	F8	RM
4D	MOV C,L	A3	ANA E	F9	SPHL
4E	MOV C,M	A4	ANA H	FA	JM aaaa
4F	MOV C,A	A5	ANA L	FB	EI
50	MOV D,B	A6	ANA M	FC	CM aaaa
51	MOV D,C	A7	ANA A	FD	
52	MOV D,D	A8	XRA B	FE	CPI dd
53	MOV D,E	A9	XRA C	FF	RST 7
54	MOV D,H	AA	XRA D		
55	MOV D,L	AB	XRA E		

Index

Note: 8080 instruction mnemonics are shown in **BOLDFACE.**

A: disk, logical, 32
Accumulator, 83
ADD, 163
Address:
  absolute, 83
  direct, 83, 177
  disk, 16
  memory, 79
  register, 92
  indexed, 93, 177
  relative, 86
ADI, 189
Allocation, disk space, 50, 173
ALU, 90
ANA, 127
AND, Boolean, 127
ANI, 127
ASCII, 20, 164
ASM, 28, 74, 115
  calculations, 164

  forward reference, 117
  options, 117
Assembly language, 73
Assignment, physical to logical, 31, 41

*****

B: disk, logical, 32
Basic Disk Operating System (see BDOS)
BAT: mode, 56
BDOS, 50
BIAS, 53
Binary, 70
Bit, 14, 69
Block:
  on disk, 50, 173
  program building, 122
BOOT, 49
Bootstrap loader, 23
Bootup sequence, 24

CP/M ASSEMBLY LANGUAGE PROGRAMMING

This valuable guide provides the beginning computer user with a hands-on method of learning assembly language programming and the CP/M operating system.

From the first simple exercise in the Introduction, you'll be learning the details of computer hardware—how to edit, assemble, and debug programs—and how to interface programs to the operating system. Once you've mastered techniques for interfacing, you'll be writing programs that read and write disk files.

No matter what make or model computer is used, you will be working in a familiar environment. Using facilities provided by the operating system, you will be able to write routines that communicate with input/output devices and mass storage units on **any** computer system running CP/M.

Written in an informal style—and requiring no previous CP/M or assembly language experience—**CP/M ASSEMBLY LANGUAGE PROGRAMMING** explains everything you need to know to construct readable, flexible, and portable programs using a CP/M operating system, including useful information on:

- hardware and software components of the computer system
- how to select I/O devices
- CP/M's built-in line editing feature
- characteristics of the 8080 microprocessor family
- register organization and data paths
- disk file access
- and much more.

Ken Barbier has more than thirty years of experience in electronics and computers and is self-employed as a writer and consultant. In addition to writing numerous articles on computer hardware, software, and applications, he has been involved in designing, constructing, and programming data acquisition systems for radio astronomy since 1969.