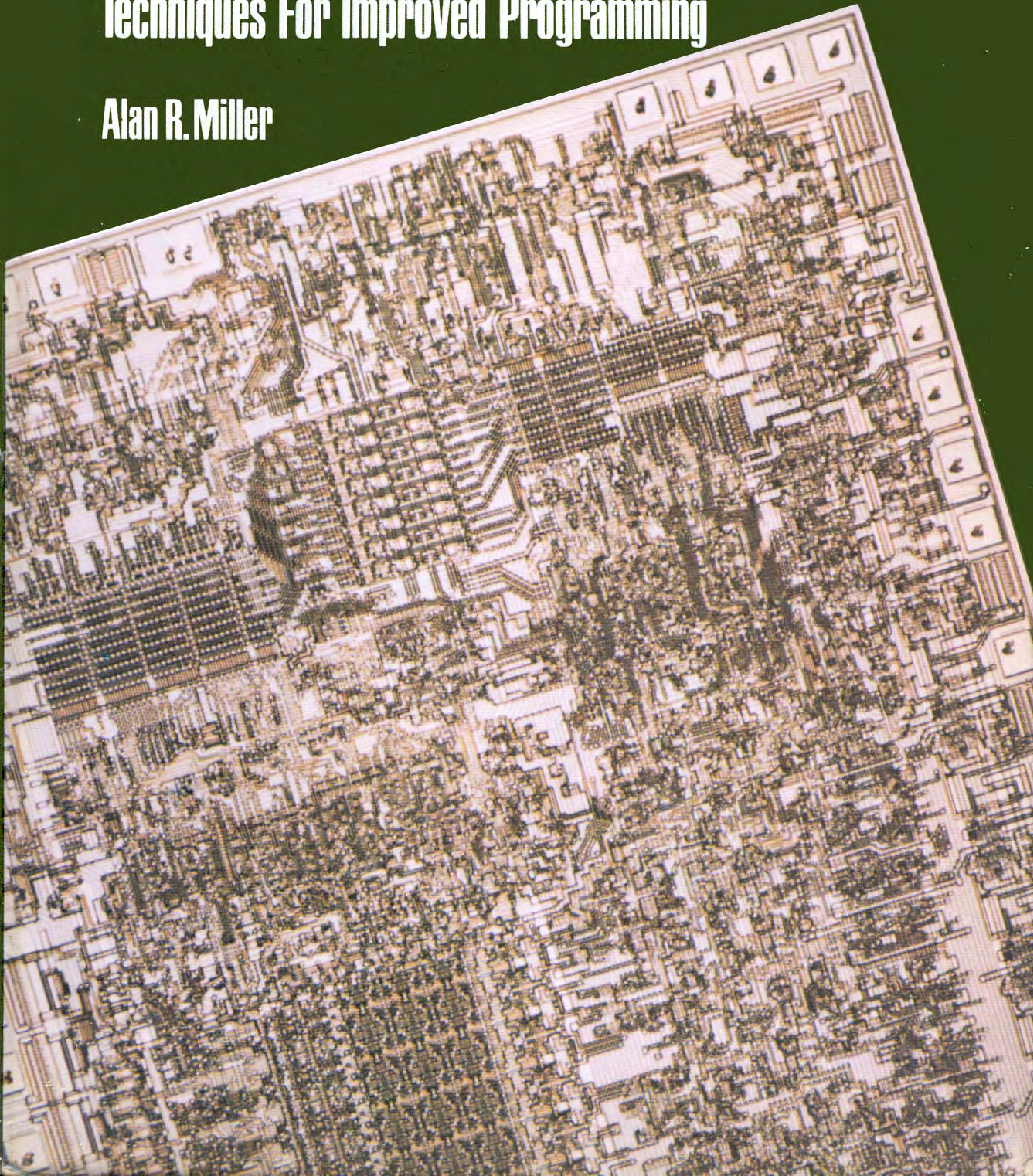
8080/Z80 Assembly Language

Techniques For Improved Programming

Alan R. Miller



THE 8080/Z-80 ASSEMBLY LANGUAGE

TECHNIQUES FOR IMPROVED PROGRAMMING

ALAN R. MILLER

Professor of Metallurgy
New Mexico Institute of Mining and Technology
Socorro, New Mexico

Software Editor, Interface Age
Cerritos, California

a7

John Wiley & Sons, Inc., Publishers

New York • Chichester • Brisbane • Toronto • Singapore

Publisher: Judy Wilson
Production Manager: Ken Burke
Editorial Supervision: Winn Kalmon
Line Artist: Carl Brown
Page Makeup: Meredythe

Library of Congress Cataloging in Publication Data

Miller, Alan R. 1932–
8080/Z-80 assembly language.

Includes index.
1. INTEL 8080 (Computer)—Programming. 2. Zilog model Z-80 (Computer)—Programming. 3. Assembler language (Computer program language) I. Title.
QA76.8.I28 M53 001.64′2 80-21492
ISBN 0-471-08124-8

Printed in the United States of America

81 82 10 9 8 7 6 5

Preface

On first thought, it might seem strange that another book on the 8080 and Z-80 should appear at this time. Z-80 CPU cards generally became available in 1977 and the 8080 CPU is even older. But the Z-80 computer seems to become more popular with time. For example, the TRS-80 Model II announced recently by Radio Shack, and Heath's H-89 both use the CPU. High-level languages such as Pascal, APL, BASIC, FORTRAN, and C are now run on the 8080 and Z-80. Furthermore, Microsoft has available a Z-80 CPU card that can be easily inserted into the Apple II computer. There should be an increasing interest in the 8080 and Z-80 CPUs in the coming years, and I believe, a great increase in the number of 8080 and Z-80 programmers. So, there is a growing need for a book that covers programming for the 8080 and Z-80 assembly languages.

The combination of 8080 and Z-80 programming concepts into a single work is quite natural. The Z-80 CPU is upward compatible from the 8080 so that all commercially available 8080 software will run on the Z-80. Furthermore, 8080 assemblers, such as ASM provided with CP/M, can be used to create programs that will run on either an 8080 system or a Z-80 system.

The purpose of this book is twofold. First, I want to provide a single reference source for both 8080 and Z-80 assembly language programmers. The appendixes are designed with this goal in mind. They begin with the ASCII character set and a 64K memory map. These two appendixes are as useful to those using higher level languages as they are to assembly language programmers.

The 8080 and Z-80 instruction sets are listed both alphabetically and numerically in the next four appendixes. This is followed by a cross reference between the 8080 and the Z-80 mnemonics. An appendix describing each instruction in detail then follows. Common acronyms are identified next in Appendix I, and some undocumented Z-80 instructions are discussed in the final appendix. Collectively, the appendixes contain all of the reference material needed to write 8080 or Z-80 assembly language programs.

The second purpose of this work is to demonstrate some useful techniques of assembly language programming. As an editor for Interface Age, I have seen numerous examples of inefficient or improper programming. General principles of assembly language programming are discussed in Chapters One through Five; specific programming examples are given in Chapters Five through Ten. The reader can actually assemble the programs and try them out.

The organization and operation of the 8080 and Z-80 CPUs is covered in Chapter One. This includes a discussion of the general-purpose registers, the flag registers, logical operations, branching, double-register operations, rotation and shifting. The concepts of hexadecimal, octal, and binary numbers, one's and two's complement arithmetic, and the use of logical operations are presented in Chapter Two.

Stack operations with PUSH, POP, CALL and RET commands and the passing of data between calling program and subroutine are given in Chapter Three. Chapter Four is devoted to input and output techniques, including an interrupt-dirven keyboard routine and a telephone transmission program. Assembler macros are discussed in Chapter Five. Examples show how to generate Z-80 instructions with an 8080 macro assembler, and how to emulate Z-80 instructions on an 8080 CPU.

The reader can develop a small, powerful monitor in Chapter Six using the top-down programming method. The monitor contains the usual commands of dump, load, and go. In addition, there is a memory test, a routine to search for one or two hex bytes or ASCII characters, a routine to replace all occurrences of one byte with another, and a routine to perform input and output through any port.

In Chapter Seven the monitor is converted to Z-80 instructions and some additional features are added. Assembly-language subroutines for interconverting between binary numbers and ASCII characters coded in one of the common number bases are given in Chapter Eight. These routines perform all of the input and output through the system monitor developed in Chapter Six. Paper tape and magnetic tape routines are given in Chapter Nine. This method of data transfer is still very popular. I frequently am asked to read information on paper tape into our Z-80 computer so that it can be transmitted over the telephone line to our campus Dec-20 computer.

CP/M is currently the most popular 8080/Z-80 operating system. Chapter Ten demonstrates how assembly language programs can utilize CP/M for all input and output by presenting three programs. One of these programs allows the user to branch to any address from the system level. Nevertheless, the use of CP/M is not the subject of this book. More information on the use of the CP/M operating system can be obtained from *Using CP/M: A Self-Teaching Guide* by Judi Fernandez and Ruth Ashley (John Wiley and Sons, Inc., 1980).

The assembly language programs in this book have all been assembled on an Altair 8800, with an Ithaca Z-80 CPU card and North Star double-density disks. The Lifeboat 2.0 version of CP/M was used as the operating system. The system monitor given in Chapter Six was additionally programmed to run on a TRS-80 Model II, using a Lifeboart 2.2 version CP/M operating system. The alternate version of the input and output routines was used in this case. The Digital Research assembler MAC was used for the 8080 instructions and the Microsoft assembler MACRO-80 was used for the Z-80 code. All of the assembly listings have been reproduced directly from the original computer printouts. The manuscript was created and edited with MicroPro's Word-Master and formatted with Organic Software's Textwriter.

Thanks to Heidi for typing the manuscript. Also, I should like to acknowledge the programmers at Microsoft, Digital Research, and Lifeboat Associates for the many things they have taught me about programming.

Alan R. Miller
June 1980

Contents

Chapter One	**Introduction**	1
	The 8080 CPU, 3	
	The Memory Register, 5	
	The Flag Register, 5	
	Flags and Arithmetic Operations, 6	
	Flags and Logical Operations, 7	
	Increment, Decrement, and Rotate Instructions, 8	
	Rotation of Bits in the Accumulator, 9	
	Flags and Double-Register Operations, 10	
	The Z-80 CPU, 10	
	Z-80 Relative Jumps, 12	
	Z-80 Double-Register Operations, 13	
	Z-80 Input and Output (I/O) Instructions, 14	
	Shifting Bits, 14	
Chapter Two	**Number Bases and Logical Operations**	16
	Number Representation in Binary, BCD, and ASCII, 19	
	Logical Operations, 20	
	The Two's Complement, 21	
	Logical OR and Logical AND, 23	
	Setting a Bit with Logical OR, 24	
	Resetting a Bit with Logical AND, 25	
	Logical Exclusive OR, 26	
	Logical NAND and NOR gates, 26	
	Making Other Gates, 28	
Chapter Three	**The Stack;**	30
	Storing Data on the Stack, 31	
	The Accumulator and PSW as a Double Register, 34	

Z-80 Index Registers, 35
Subroutine Calls, 35
Passing Data Improperly to a Subroutine, 37
Passing Data Properly to a Subroutine, 37
Passing Data Back from a Subroutine, 39
Setting up a New Stack, 40
Calling a Subroutine in Another Program, 41
Calling One Subroutine from Another, 42
Bypassing a Subroutine on Return, 43
A PUSH Without a POP, 44
Getting Back from a Subroutine, 44
Automatic Stack Placement, 45

Chapter Four Input and Output 48
Memory-Mapped I/O, 48
Distinct Data Ports, 49
Looping, 50
Polling, 52
Hardware Interrupts, 52
An Interrupt-Drive Keyboard, 55
Scroll Control and Task Abortion, 64
Data Transmission by Telephone, 64
Parity Checking, 66

Chapter Five Macros 69
Generating Three Output Routines with One Macro, 71
Generating Z-80 Instructions with an 8080 Assembler, 73
Emulating Z-80 Instructions with an 8080 CPU, 75
The Repeat Macros, 77
Printing Strings with Macros, 79

Chapter Six Development of a System Monitor 85
Program Development Details, 86
Version 1: The Input and Output Routines, 87
Version 2: A Memory Display, 95
Version 3: A CALL and GO Routine, 100
Version 4: A Memory-Load Routine, 101
Version 5: Useful Entry Points, 103
Version 6: Automatic Memory Size, 105
Version 7: Command-Branch Table, 107
Version 8: Display the Stack Pointer, 109
Version 9: ZERO and FILL routines, 110
Version 10: A Block-Move Routine, 111
Version 11: A Search Routine, 113
Version 12: ASCII Load, Search, and Display, 115

Version 13: Input and Output to Any Port, 117
Version 14: Hexadecimal Arithmetic, 119
Version 15: Memory-Test Program, 120
Version 16: Replace One Byte with Another, 121
Version 17: Compare Two Blocks of Memory, 123
Automatic Execution of the Monitor, 125

Chapter Seven — A Z-80 System Monitor — 128
Conversion of the Monitor to Z-80 Mnemonics, 143
Reducing the Monitor Size, 144
Getting More Free Space, 145
Peripheral Port Initialization, 146
Printer Output Routines, 147
Delay After a Carriage Return, 148

Chapter Eight — Number-Base Conversion — 150
The ASCII Code, 150
Conversion of ASCII-Encoded Binary Characters to an 8-Bit Binary Number in Register C, 152
Conversion of ASCII Decimal Characters to a Binary Number, 156
Conversion of ASCII Hexadecimal Characters to a 16-Bit Binary Number In HL, 159
Conversion of Two ASCII Hexadecimal Characters to an 8-Bit Binary Number in Register C, 162
Conversion of ASCII Octal Characters to a 16-Bit Binary Number in Register HL, 163
Conversion of Three ASCII Octal Characters to an 8-Bit Binary Number in Register C, 166
Conversion of Two ASCII BCD Digits to an 8-Bit Binary Number in Register C, 168
Conversion of an 8-Bit Binary Number in C to a String of Eight ASCII Binary Characters, 169
Conversion of an 8-Bit Binary Number into Three ASCII Decimal Characters, 172
Conversion of a 16-Bit Binary Number into Five ASCII Decimal Characters, 175
Conversion of an 8-Bit Binary Number into Two ASCII Hexadecimal Characters, 178
Conversion of a 16-Bit Binary Number into Six ASCII Octal Characters, 180
Conversion of an 8-Bit Binary Number into Three ASCII Octal Characters, 181
Conversion of a 16-Bit Binary Number to Split Octal, 182

Chapter Nine		**Paper Tape and Magnetic Tape Routines** The Checksum Method, 188 An ASCII-Hex Tape Program, 188 A Tape-Labeling Routine, 203 A Binary Tape Monitor, 204	187
Chapter 10		**Linking Programs to the CP/M Operating System** CP/M Memory Organization, 216 Changing the Peripheral Assignment, 217 Incorporating the IOBYTE into Your CBIOS, 219 Using STAT to Change the IOBYTE, 225 A Routine to Go Anywhere in Memory, 226 A List Routine with Date and Time, 229 Copy a Disk File into Memory, 242	214
Appendixes	A.	The ASCII Character Set	253
	B.	A 64K Memory Map	255
	C.	The 8080 Instruction Set (Alphabetic)	258
	D.	The 8080 Instruction Set (Numeric)	261
	E.	The Z-80 Instruction Set (Alphabetic)	264
	F.	The Z-80 Instruction Set (Numeric)	272
	G.	Cross-Reference of 8080 and Z-80 Instructions	280
	H.	Details of the Z-80 and 8080 Instruction Set	283
	I.	Abbreviations and Acronyms	311
	J.	Undocumented Z-80 Instructions	313
Index			317

CHAPTER ONE
Introduction

There was a time when computers were gigantic machines containing racks upon racks of vacuum tubes. The invention of the transistor and the development of the integrated circuit (IC) changed all that. Today, it is possible to place tens of thousands of transistors on a single "chip" of silicon that is smaller than a quarter of an inch square. As a result of this technology. computers have become smaller and cheaper.

Computers are commonly classified into three categories, based on size and capability. The largest are known as *main frame computers*, the middle-sized ones are called *minicomputers*, and the smallest are termed *microcomputers.* A computer consists of three parts: the *central processing unit* (CPU), the *main memory*, and the *peripherals.*

The CPU directs the activities of the computer by interpreting a set of *instructions* called *operation codes*, or *op codes* for short. These instructions are located in the main memory. The memory is also used for the storage of data.

```
                          !--------------!
                          !  main memory !
                          !--------------!
                                 ^
!--------!      buss             |
!  CPU   !  <=========>          |
!--------!                       v
                          !--------------!
                          !  peripherals !
                          !--------------!
```

The CPU communicates with the user through such peripherals as the console, the printer, the disks, and so on. There are several electrical lines which are used to connect the CPU to the memory and to the peripherals. These lines are collectively known as the *buss*, or *bus*.

The CPU contains a set of *registers*, which are internal memory locations used for data storage and manipulation. One of these is a special register called the *accumulator*. It receives the results of certain CPU operations. The CPU will also have a *status* register to indicate the nature of a previous operation, e.g., whether the result is zero or negative or positive. It will also indicate whether a carry or a borrow occurred during the operation.

Additional registers are used for auxiliary storage. They may contain general information such as a number that is about to be added to the accumulator. Alternately, a register may contain a number that refers to an *address* in the main memory. The value is called a *memory pointer* in this case. A special portion of main memory may be set aside for storing data. This area is called a *stack*. A special register called a *stack pointer* refers to this region. Another register, the *program counter*, tells the CPU where to find the next instruction in memory.

Computer operations are controlled by a computer *program*. Those programs which are used to solve engineering and physics problems are called *application* programs. On the other hand, computer programs which deal with the operation of the computer's own peripherals are known as *systems* programs.

The instruction set used by the CPU can be very large and difficult to use. Consequently, symbolic programming languages are commonly used instead. An application program may be written in a language such as BASIC, FORTRAN, or Pascal. This is called a *source* program. Then a separate processor program called a *compiler* or an *interpreter* is used to convert the user's source program into an *object* program that corresponds to the instructions needed by the computer.

A microcomputer's instruction set is relatively small compared to that of a larger computer. But even so, it is more convenient to write systems programs in a symbolic language called *assembly language*, rather than in the machine language of the computer. A processor program, called an *assembler*, is then utilized to translate the source program into the corresponding instructions of the computer. A major difference between assembly language and *higher-level* languages such as Pascal is that each line of an assembly language program represents one computer instruction. By contrast, one line of a Pascal source program might represent many computer instructions.

A line of an assembly language program can contain up to four elements: the *label*, the *mnemonic*, one or two *operands*, and a comment.

```
Label     Mnemonic   Operand     Comment
START:      CALL      FIRST     ;initialize data
```

The label, which is usually terminated by a colon, is used to transfer control from one portion of the source program to another. The mnemonic represents the desired CPU instruction. The operand might reference a CPU register, a memory location, or simply a constant. Finally, a comment, preceded by a semicolon, can be used to explain the instruction. The comment, of course, is ignored by the assembler.

The remainder of this chapter is devoted to a general discussion of some of the features of the 8080 and Z-80 CPUs. The complete instruction sets for these CPUs are listed in the appendix. Specific details of each instruction are given in Appendix H. If you are already familiar with these instruction sets, then you might want to go on to the next chapter.

THE 8080 CPU

The 8080 CPU is an integrated circuit that has 40 pins (legs). It requires three power supply voltages—12 V, 5 V, and −5 V, and a two-phase clock that runs at 2 Megahertz (MHz). There is an accumulator, a flag register, six general-purpose registers, a stack pointer, and a program counter. The accumulator is sometimes known as register A. The flag register is usually called the PSW (the letters being an acronym for *program status word*). The general-purpose registers are designated by the letters B, C, D, E, H, and L. Sometimes the registers are paired into 16-bit double registers known as BC, DE, and HL. The accumulator and flag register may also be paired. There are 78 different instruction types that produce a total of 245 different op codes.

```
Accumulator      !-----------!-----------! Flag Register
(Register A)     !  8 bits   !  8 bits   !  (PSW)
                 !-----------!-----------!
Register B       !  8 bits   !  8 bits   ! Register C
                 !-----------!-----------!
Register D       !  8 bits   !  8 bits   ! Register E
                 !-----------!-----------!
Register H       !  8 bits   !  8 bits   ! Register L
                 !-----------!-----------!
Stack Pointer    !       16 bits         !
                 !-----------------------!
Program          !       16 bits         !
Counter          !-----------------------!
```

Figure 1.1. The 8080 CPU registers.

Some of the 8080 instructions explicitly refer to the accumulator or to one of the general-purpose registers (B, C, D, E, H, and L).

```
mnemonic  operand   comment
  INR       A       ;increment accumulator
  DCR       B       ;decrement register B
  MOV       H,D     ;move contents of D to H
  MVI       C,4     ;put value of 4 into C
```

When there are two operands, data moves from the right operand (the *source*) into the left operand (the *destination*). There are additional 8080 commands that implicitly refer to the accumulator.

```
mnemonic  operand    comment
  RAR               ; rotate accumulator right
  RAL               ; rotate accumulator left
  IN        0       ; input a byte to A from port 0
  OUT       1       ; output a byte from A to port 1
  ANI       7       ; logical AND with A and 7
  ORI       3       ; logical OR with A and 3
```

For certain 8-bit operations, the accumulator is implicitly one of the source registers and will contain the result of the operation.

```
mnemonic  operand     comment
  ADD       C       ; add C to A
  SUB       D       ; subtract D from A
  ANA       H       ; logical AND of A with H
  ORA       B       ; logical OR of A and B
```

Other instructions refer to coupled pairs of 8-bit registers. These extended operations treat the BC, the DE, and the HL register pairs as 16-bit entities. Sometimes the stack pointer and program counter are included in these instructions. The X symbol in the mnemonic refers to these extended 16-bit operations.

```
mnemonic  operand     comment
  INX       H       ; increment HL register pair
  DCX       SP      ; decrement stack pointer
  LXI       D,0     ; load zero into DE pair
```

Additional instructions deal specifically with the HL register pair. The following two instructions move two bytes of data between memory and the HL double register.

```
mnemonic  operand   comment
  LHLD      3     ; addr 3, 4 to L and H
  SHLD      3     ; L,H to addr 3, 4
```

The LHLD instruction copies the value at memory location 3 into the L register and the value at location 4 into the H register. The SHLD operation reverses the process.

The XCHG operation interchanges the 16-bit HL register pair with the 16-bit DE register pair.

```
        !--------------!          !---------------!
XCHG    !  HL register !  <===>   !  DE register  !
        !--------------!          !---------------!

        !--------------!          !---------------!
SPHL    !  HL register !   ===>   ! stack pointer !
        !--------------!          !---------------!

        !--------------!          !---------------!
PCHL    !  HL register !   ===>   ! program       !
        !--------------!          ! counter       !
                                  !---------------!
```

The SPHL command copies the HL register into the stack pointer register. The PCHL instruction copies the HL register pair into the program counter register.

There are several instructions that perform double-register addition. The number in one of the 16-bit registers is added to the number in HL. The sum appears in the HL register pair.

```
DAD     D       ;add DE to HL
DAD     SP      ;stack pointer + HL
```

THE MEMORY REGISTER

There is another 8-bit register for the 8080 that is not shown in Figure 1.1. It is located in main memory. The 16-bit address contained in HL defines the location of this memory register, i.e., HL is a memory pointer. The instruction

```
MOV     M,E     ;move E to memory
```

will copy the contents of register E into the memory location pointed to by the HL register pair. The instruction

```
INR     M       ;increment memory
```

will increment this byte in memory.

THE FLAG REGISTER

Four bits of the PSW register can be used to control program flow. The bits or *flags* are used in conjunction with conditional jump, conditional call, and conditional return instructions. We say that a flag is *set* if it has a value of 1 or is *reset* if it has a value of zero.

The CPU sets the sign flag (S) if the result of a previous operation is positive; the flag is reset if the result is negative. The CPU sets a second flag, the zero flag (Z), if the result is zero; it is reset if not zero. A third flag, the carry flag (C), is set if there is a carry on addition or borrow on subtraction; it is reset otherwise. A fourth flag, the parity flag (P), indicates the parity of the result. Parity is even if there is an even number of ones (or zeros) and odd otherwise.

```
!---!---!---!---!---!---!---!---!
! S ! Z !   !   !   ! P !   ! C !
!---!---!---!---!---!---!---!---!
```

Figure 1.2. The PSW (flag) register.

The use of the flag register can be demonstrated with a simple routine. Suppose that a group of instructions is to be executed eight times. The following code will do this.

```
        MVI     B,8     ;put 8 into B
LOOP:   . . .
        . . .
        DCR     B       ;decrement B
        JNZ     LOOP    ;loop if not zero
```

The B register is initialized to the value of 8. The DCR B instruction near the end of the loop decrements the B register each time the loop is executed. This will reset the zero flag on each of the first seven passes through the loop since B has not reached zero. The following conditional jump instruction, JNZ LOOP, causes the CPU to return to the line labeled LOOP in this case.

On the eighth pass through the loop the original value of 8 in the B register will have been decremented to zero. Now the zero flag will be set and the conditional jump instruction will not cause a branch. The instruction immediately following the jump will be executed instead.

FLAGS AND ARITHMETIC OPERATIONS

The results of addition and subtraction operations can be characterized from the PSW flags. Three of the flags are of interest here: the carry flag, the zero flag, and the sign flag. If the sum of two numbers exceeds 255 (1 less than 2 to the eighth power), then the result is too large to fit into the 8-bit accumulator. The carry flag will be set to reflect this overflow. During subtraction, the carry flag is set when a larger number is subtracted from a smaller one. In this case, the flag becomes an indication that borrowing has taken place.

Sometimes all eight bits of a register or memory location are used to represent a number. This is then an *unsigned* number. At other times it is convenient to utilize only the low-order seven bits (bits 0–6) for the magnitude of a number. The remaining high-order bit (bit 7) is then used to indicate the sign.

```
      magnitude                sign     magnitude
!---------------!              !-!-------------!
!   8   bits    !              ! !    7 bits   !
! ! ! ! ! ! ! ! !              ! ! ! ! ! ! ! ! !
!-!-!-!-!-!-!-!-!              !-!-!-!-!-!-!-!-!
 7 6 5 4 3 2 1 0                7 6 5 4 3 2 1 0

  unsigned number                signed number
```

Numbers represented in this way are known as *signed* numbers. A 0 in bit 7 means that the number is positive and a 1 means that the number is negative. An 8-bit signed number can range in magnitude from −128 to 127, whereas an unsigned 8-bit number can range from 0 to 255.

The sign flag is set after certain operations if the value of bit 7 is 1 and it is reset if bit 7 is 0. If the sum of two numbers is exactly 256, the result in the 8-bit accumulator will be a zero. This occurs because 256 is 1 greater than the largest 8-bit number (255). The zero flag will be set in this case. In addition, the carry flag will be set because there is an overflow. The parity flag will be set, since there is an even number of ones. (Zero is an even number.) Finally, the sign flag will be reset because bit 7 is a zero.

FLAGS AND LOGICAL OPERATIONS

In the case of arithmetic operations such as addition and subtraction, there can be a carry or borrow from one bit to another. But the logical operations AND, OR, and XOR (exclusive OR) operate on each bit separately; there is never a carry from one bit to the next. These logical operations, therefore, always reset the carry flag. The zero and parity flags, however, will be set or reset according to the result of the particular operation. We will discuss logical operations more fully in Chapter 2.

A value in the accumulator can be compared to a value in another register or to the byte immediately following the instruction byte in memory. The CPU performs the comparison by subtracting the value of the operand from the value in the accumulator. In the case of a regular subtraction, the difference is placed in the accumulator. For example, the arithmetic instruction

```
SUB     C
```

subtracts the value in register C from the accumulator and places the difference into the accumulator. The logical comparison operation

```
CMP     C
```

also subtracts the value in register C from the value in the accumulator. However, unlike the regular subtraction operation, the difference in this case is not actually saved. The flags, of course, will reflect the result of the operation. If the value in C is equal to the value in the accumulator the difference between them will be zero. In this case the zero flag is set indicating the equality. The carry flag will be reset since there was no borrow during the subtraction.

If the two values are not equal, then A is either larger or smaller. If A is larger, the comparison operation will reset the carry flag (and, of course, the zero flag). If A is smaller, then the carry flag will be set, because a larger number has been subtracted from a smaller one. Thus, if the carry flag has been set after a comparison, then the value originally in the accumulator must have been smaller than the value with which it was compared.

The following instructions can be used to determine if the value in register C is less than, greater than, or equal to the value in the accumulator.

```
CMP     C       ; subtract A from C
JZ      ZERO    ; if A equals C
JC      LESS    ; if A less than C
. . .           ; if A greater than C
```

The comparison instruction is executed first. This operation subtracts the value of C from the value in the accumulator. If the two numbers are equal, then their difference is zero. In this case, the zero flag is set and the JZ instruction causes a branch to the label ZERO. Otherwise, the next instruction is executed. Another possibility is that the value in C is greater than the value in the accumulator. The subtraction in this case requires a borrow so the carry (borrow) flag is set. The JC instruction then causes a branch to the label LESS. The last possibility is that the value in the accumulator is larger than that in register C. For this case, both the zero and the carry flags are reset, and the program continues.

INCREMENT, DECREMENT, AND ROTATE INSTRUCTIONS

The 8-bit increment and decrement instructions present an interesting case. Mathematically, the increment operation simply adds 1 to the current value in a register. Likewise, the decrement operation subtracts 1 from the present value. Thus, the two instructions

```
INR     A     and
ADI     1
```

both increase the value in the accumulator by 1 and the operations

```
DCR     A     and
SUI     1
```

both decrease the value in the accumulator by 1. The zero, parity, and sign flags correctly reflect the result in all cases.

The carry flag, however, responds differently for the two cases. The flag correctly reflects the result of the operation in the case of addition, but it is unaffected in the case of an increment or decrement operation. Thus, if you need to increment or decrement a value without disturbing the carry flag, then you should use the INR or DCR instructions. On the other hand, if you need to know whether a carry or borrow occurred during an increment or decrement, then use an add or subtract operation.

The instructions following the label GETCH in Listing 6.1 (in Chapter 6) are used to set ASCII characters from the console input buffer. As each character is obtained, the count of the remaining characters is decremented. When the count has been decremented past 0, then the routine is finished. Subtracting 1 from 0 requires a borrow so the carry flag should be set. But since the regular decrement operation doesn't alter the carry flag, the subtract instruction must be used instead.

ROTATION OF BITS IN THE ACCUMULATOR

There are four 8080 instructions that rotate the bits in the accumulator. The operations move each bit by one position. Two instructions rotate the bits to the right and two rotate them to the left. The right circular rotation

```
RRC
```

moves each bit one position to the right. The rightmost (low-order) bit is moved to the high-order bit and into the carry flag.

```
!-------<-------------<-----------------<-!
!                                         !
!      !---!---!---!---!---!---!---!---!  !      !---!
!-->   !   ->  ->  ->  ->  ->  ->  ->  ->--!------>   !
       !---!---!---!---!---!---!---!---!         !---!
                  accumulator                    carry
```

The left circular rotation

```
RLC
```

moves the bits the other way.

```
                 !->------------->----------------->-------!
                 !                                         !
!---!            !     !---!---!---!---!---!---!---!---!   !
!   <------!---<-  <-  <-  <-  <-  <-  <-  <-  <-----!
!---!                  !---!---!---!---!---!---!---!---!
carry                         accumulator
```

Each bit is moved one position to the left. The high-order bit goes to both the low-order bit and to the carry flag.

The rotate accumulator right instruction

```
RAR
```

moves each bit one position to the right. But this time, the carry flag moves into the high-order bit and the low-order bit moves into the carry flag.

```
!-------<-------------<-----------------<---------<-!
!                                                   !
!     !---!---!---!---!---!---!---!---!        !---! !
!->   !   ->  ->  ->  ->  ->  ->  ->  ------->  !   !-!
      !---!---!---!---!---!---!---!---!        !---!
              accumulator                      carry
```

The instruction

```
RAL
```

moves each bit one position to the left. The carry flag moves into the low-order bit and the high-order bit moves into the carry flag.

```
!->------------>-------------->------------------>------!
!                                                        !
!   !---!         !---!---!---!---!---!---!---!---!     !
!-  !   ! <-------<-  <-  <-  <-  <-  <-  <-  <-   ! <-!
    !---!         !---!---!---!---!---!---!---!---!
    carry                    accumulator
```

FLAGS AND DOUBLE-REGISTER OPERATIONS

Double-register, or extended, operations involving HL, DE, and BC affect the flags very differently from the single-register operations. We saw that single-register increment and decrement operations did not alter the carry flag. The extended increment and decrement commands never alter any of the flags. This means that if a program is to loop until a double register has been decremented to zero, the following set of instructions will not work.

```
LOOP:   . . .
        . . .
        DCX     H       ;16-bit decrement
        JNZ     LOOP    ;if not zero
```

The proper procedure is to compare the two 8-bit halves with each other. This can be done by moving one of the registers to the accumulator.

```
MOV     A,L
```

Then the accumulator is compared to the other half by performing a logical OR. The result of this operation will set the zero flag only if both halves are zero. The complete operation looks like this.

```
LOOP:   . . .
        . . .
        DCX     H       ;16-bit decrement
        MOV     A,L     ;move L to A
        ORA     H       ;OR with H
        JNZ     LOOP    ;if not zero
```

The double-register add instruction correctly sets the carry flag if there is an overflow from the 16 bits, but zero, parity, and sign flags are not altered.

THE Z-80 CPU

The Z-80 CPU is a 40-pin IC just like the 8080. All of the 8080 instructions are common to the Z-80, thus we say the Z-80 is upward compatible from the 8080. In general, any program that runs on an 8080 will also run on a

Z-80. The one exception is that the 8080 parity flag is affected by arithmetic operations, while the Z-80 parity flag is not. Thus, one can use an 8080 assembler to generate 8080 code on a Z-80.

The Z-80 requires only a single 5-volt power supply and a single-phase clock that can run as fast as 6 MHz. There are 158 instruction types that give a very large number of total commands with all variations. These are given briefly in Appendixes E and F and in more detail in Appendix H. The Z-80 contains all of the 8080 general-purpose registers, plus an alternate set for easy interrupt processing. The alternate set is indicated with a prime symbol: A′, B′, and so on. Only one of the two general sets of registers can be used at any time, therefore, data cannot be transferred directly from one set to the other. There are also two 16-bit index registers called IX and IY, an 8-bit interrupt register (I), and an 8-bit refresh register (R).

```
        Primary registers             Alternate registers
     !---------!---------!            !--------!--------!
  A  ! 8 bits  ! 8 bits  ! PSW     A' ! 8 bits ! 8 bits ! PSW'
     !---------!---------!            !--------!--------!
  B  ! 8 bits  ! 8 bits  ! C       B' ! 8 bits ! 8 bits ! C'
     !---------!---------!            !--------!--------!
  D  ! 8 bits  ! 8 bits  ! E       D' ! 8 bits ! 8 bits ! E'
     !---------!---------!            !--------!--------!
  H  ! 8 bits  ! 8 bits  ! L       H' ! 8 bits ! 8 bits ! L'
     !---------!---------!            !--------!--------!
  SP !      16 bits      !
     !-------------------!
  PC !      16 bits      !
     !-------------------!


           Index               !-----------------------!
           Register X          !        16 bits        !
                               !-----------------------!
           Index               !        16 bits        !
           Register Y          !-----------------------!

           Interrupt           !----------!
           Register I          !  8 bits  !
                               !----------!
           Refresh             !  8 bits  !
           Register R          !----------!
```

Figure 1.3. The Z-80 CPU registers.

An operand for an assembly-language instruction may consist of a value that is used directly, or it may refer to a location that contains the value. For example, the command

```
LD      A,6
```

instructs the CPU to place the value of 6 into register A. Similarly, the instruction

```
LD      A,D
```

will move the contents of register D into register A. Alternately, the operand may be a pointer to another location. Thus the command

```
LD      A,(6)
```

will move the byte located at address 6 into the accumulator. Similarly, the instruction

```
LD      A,(HL)
```

tells the CPU to move the byte pointed to by the HL register into the accumulator. The Z-80 mnemonics clearly differentiate a pointer by means of the parentheses, whereas the corresponding 8080 mnemonics do not make such a clear distinction.

Z-80 RELATIVE JUMPS

Computer instructions are generally executed in order, one after the other. But it is sometimes necessary to branch out of the normal sequence of statements. Branching statements can be classified as either *conditional* or *unconditional*. An unconditional or absolute branch always causes the computer to execute instructions at a new location, out of the normal flow. Conditional branching, on the other hand, is based upon the condition of one of the flags.

Programs utilizing the Z-80 instruction set can be significantly shorter than those written with 8080 operation codes, especially if the relative jump instructions are used. Relative jumps are performed by branching forward or backward relative to the present position. Absolute jumps, on the other hand, are made to a specific memory location. Furthermore, there are both unconditional and conditional branch instructions. The absolute, unconditional jump op code and the conditional jump codes based on the state of the zero and parity flags are all three-byte instructions.

```
JP      ADDR1      ; unconditional jump
JP      Z,ADDR2    ; jump if zero flag set
JP      NZ,ADDR3   ; jump if zero flag reset
JP      C,ADDR4    ; jump if carry flag set
JP      NC,ADDR5   ; jump if carry flag reset
```

The above instructions are available on both the 8080 and the Z-80 CPUs. In addition, the Z-80 has a relative, unconditional jump and five relative, conditional jumps.

```
JR      ADDR       ; unconditional jump
JR      Z,ADDR6    ; zero
JR      NZ,ADDR7   ; not zero
JR      C,ADDR8    ; carry
JR      NC,ADDR9   ; not carry
DJNZ    ADDR10     ; decr, jump not zero
```

The relative jumps are only two bytes long as opposed to three bytes for the regular jumps, but the relative jump is limited to a displacement of less than 126 bytes forward or 128 bytes backward from the address of the current instruction. These numbers derive from the magnitude of the signed 8-bit displacement. Bit 7 is used for the sign of the number. A 0 in bit position 7 means a forward or positive displacement, a 1 in this bit position means a backward or negative displacement. The remaining seven bits are used for the magnitude of the jump.

Absolute jumps are specified with a 16-bit address that gives the new location. Relative jumps on the other hand are position-independent. The resulting code can be placed anywhere in memory. The last operation above, DJNZ, is a combination of two instructions. The B register is decremented. If the result is not zero, then there is a relative jump to the given argument ADDR10. This two-byte instruction requires four bytes on an 8080 CPU.

Z-80 DOUBLE-REGISTER OPERATIONS

While some of the Z-80 instructions appear to be shorter than their 8080 counterparts, they may not actually reduce the program size. Suppose, for example, that we want to move a block of data from one memory location to another. There is a single Z-80 instruction for accomplishing this task. The problem is that no verification is performed during the move. Thus, if there were no memory at the new location, or if the memory were defective, this fact would not immediately be discovered. If you want to check each location as the data are moved, then the Z-80 block-move instruction cannot be used.

A better way to move data is to define the beginning of the original memory block with HL and the end with DE. The BC register defines the beginning of the new block. We can work our way through the original block by incrementing HL and BC at each step along the way.

The end of the block can be detected when HL exceeds DE. We subtract the two 16-bit registers and observe the carry flag. The HL register pair will initially be less than the DE pair. Therefore, if we subtract DE from HL, we will set the carry (borrow) flag.

Eventually, the number in the HL register will equal the value in the DE pair. This time, the subtraction will not set the carry flag and the task will be completed. Since the 8080 doesn't have a 16-bit subtract instruction, the routine might look like this.

```
LOOP:   . . .                 ; 8080 version
        . . .
        MOVE    A,L           ; GET L
        SUB     E             ; SUBTRACT E
        MOV     A,H           ; GET H
        SBB     D             ; SUBTRACT D AND BORROW
        JC      LOOP          ; IF NOT DONE
        RET                   ; DONE
```

As long as HL is less than DE, the subtraction will set the carry flag and the loop will be repeated. But as soon as HL equals DE, the carry flag will be reset and the subroutine is finished.

The Z-80 has a 16-bit subtract instruction that can simplify the operation. But since the result of the subtraction is placed in the HL register pair rather than in the accumulator, the data originally present in HL will have to be saved somewhere else, say, on the stack. The Z-80 code is:

```
LOOP:   . . .               ; Z-80 version
        OR      A           ; RESET CARRY
        PUSH    HL          ; SAVE HL ON STACK
        SBC     HL,DE       ; SUBTRACT DE FROM HL
        POP     HL          ; RESTORE  ORIGINAL HL PAIR
        JR      C,LOOP      ; IF NOT DONE
        RET
```

The necessary Z-80 instructions require just as many bytes as the corresponding 8080 code. And if the carry flag on the Z-80 has not been reset by a previous instruction, it will have to be reset at the beginning with a logical OR instruction. This latter problem occurs because the Z-80 16-bit subtraction includes the carry flag in its calculations.

Z-80 INPUT AND OUTPUT (I/O) INSTRUCTIONS

A useful pair of Z-80 instructions deals with input and output, i.e., the transfer of data between the CPU and peripherals such as the console, the printer, and the disk. The 8080 can only input and output data from the accumulator, and the address of the peripheral device must immediately follow the IN or OUT instruction in memory.

```
OUT     10
IN      11
```

This usually means that for *read-only memory* (ROM), there must be separate input and output routines for each peripheral.

In contrast, the Z-80 can input or output a byte from any of the general-purpose registers when the peripheral address is in the C register. In this case, it may be possible to use a single set of I/O routines for all peripherals. This approach is discussed more fully in Chapter 4.

SHIFTING BITS

The Z-80 CPU extends the four 8080 rotate instructions to the general-purpose registers B, C, D, E, H, and L. The memory byte referenced by HL, IX, and IY is also included.

The Z-80 instruction set includes three shift operations. Shifts are similar to rotations since each bit moves one position and the bit that is shifted out of the register is moved into the carry flag. The difference is in the bit that is shifted into the register.

The arithmetic shift left

```
SLA
```

shifts all bits to the left. A zero bit moves into the low-order bit.

```
!---!          !---!---!---!---!---!---!---!---!
!   ! <-------<-  <-  <-  <-  <-  <-  <-  <-   ! <-- 0
!---!          !---!---!---!---!---!---!---!---!
carry                    accumulator
```

The operation doubles the original 8-bit value. This operation can be performed on the accumulator of an 8080 by using an

```
ADD    A
```

instruction.

A logical shift right

```
SRL
```

is the inverse of the arithmetic shift left operation. Each bit shifts one position to the right. A zero bit is shifted into the high-order position.

```
       !---!---!---!---!---!---!---!---!              !---!
0 -->  !   ->  ->  ->  ->  ->  ->  ->  ->-------> !   !
       !---!---!---!---!---!---!---!---!              !---!
           accumulator                                carry
```

This operation halves the original 8-bit value. The carry flag is set if the original value was odd, that is, if there is a remainder from the division.

The arithmetic shift right

```
SRA
```

shifts each bit one position to the right, but the original high-order bit is unchanged.

```
      !---!---!---!---!---!---!---!---!              !---!
!-->  !   ->  ->  ->  ->  ->  ->  ->  ->-------> !   !
!     !---!---!---!---!---!---!---!---!              !---!
!     !          accumulator                         carry
!---<---!
```

This operation can be used to divide a signed number in half. The high-order bit, the sign bit, is unchanged. As with the logical shift right, the carry flag is set if the original number was odd.

CHAPTER TWO

Number Bases and Logical Operations

In this chapter we will consider how numbers are stored in a computer. We will also look at some of the operations that can be performed on these numbers. But first we will review the representation of numbers in general. When we write a number such as 245, we usually mean the quantity 5 plus 40 plus 200.

```
2  4  5        (decimal)
|  |  |
|  |  └──────  5 X   1 =   5
|  └─────────  4 X  10 =  40  (4 X the base)
└────────────  2 X 100 = 200  (2 X the base squared)
                         ---
                         245  (decimal)
```

This is the ordinary decimal or base-10 representation of a number. The rightmost digit gives the number of units. The digit immediately to the left is the number of tens (the base). The next digit to the left is the number of 100s (the base squared).

In assembly language programs it is sometimes convenient to represent numbers with a base of 2, 8, or 16. In the octal, or base-8, system, for example, the number 245 is equivalent to the decimal number 165 (5 plus 32 plus 128).

```
2  4  5        (octal)
|  |  |
|  |  └──────  5 X  1 =   5
|  └─────────  4 X  8 =  32  (4 X the base)
└────────────  2 X 64 = 128  (2 X the base squared)
                        ---
                        165  (decimal)
```

This example demonstrates how to convert numbers from other bases into the decimal representation by adding up the decimal equivalent of each digit.

In the binary, or base-2, system, only the digits 0 and 1 are used. The individual digits are called *bits*, an acronym for binary digits. The rightmost bit represents the units. The bit immediately to the left is the number of 2s (the base). The next bit to the left is the number of 4s (the base squared). We continue in this way through all of the bits. For example, the binary number

10100101

is equivalent to the decimal number 165. The conversion is obtained in the following way.

```
1  0  1  0  0  1  0  1        (binary)
|  |  |  |  |  |  |  └──────  1 ×   1 =   1
|  |  |  |  |  |  └─────────  0 ×   2 =   0
|  |  |  |  |  └────────────  1 ×   4 =   4
|  |  |  |  └───────────────  0 ×   8 =   0
|  |  |  └──────────────────  0 ×  16 =   0
|  |  └─────────────────────  1 ×  32 =  32
|  └────────────────────────  0 ×  64 =   0
└───────────────────────────  1 × 128 = 128
                                        ───
                                        165  (decimal)
```

We have seen that the decimal system utilizes ten different digits (0–9). The octal system, however, utilizes only eight digits (0–7), and the binary system uses only two (0–1). The hexadecimal, or base-16, system is also commonly used in computer programs. With this method, we need 16 different digits. The problem is that if we use all of the digits (0–9) from the decimal system, we will still be six digits short. The solution is to use the letters A through F to represent the digits beyond 9. Thus, the hexadecimal number A5 is equivalent to the decimal number 165. We can convert a hexadecimal number into decimal in the usual way if we remember that A stands for decimal 10, B for 11, and so on.

```
A    5         (hexadecimal)
|    └──────   5 ×  1 =   5
└───────────  10 × 16 = 160  (10 times the base)
                        ───
                        165  (decimal)
```

The first 16 integers of the decimal, binary, octal, and hexadecimal systems are shown in Table 2.1.

Table 2.2. The first 16 integers represented in various number systems.

decimal	binary	octal	hex
0	0000	000	00
1	0001	001	01
2	0010	002	02
3	0011	003	03
4	0100	004	04
5	0101	005	05
6	0110	006	06
7	0111	007	07
8	1000	010	08
9	1001	011	09
10	1010	012	0A
11	1011	013	0B
12	1100	014	0C
13	1101	015	0D
14	1110	016	0E
15	1111	017	0F

Table 2.1 shows the common practice of displaying leading zeros on numbers expressed in bases other than 10. Thus we write 5 for a decimal number, but we may write 005 if it is an octal number or 05 if it is a hexadecimal number. We may explicitly represent the base by a suffix. In books, for example, we typically utilize a subscript in smaller size type. Thus we will write:

1010_2 (binary)
17_8 (octal)
17_{10} (decimal)
17_{16} (hexadecimal)

Alternately, we use suffixes of B, Q, D, and H to designate, respectively, binary, octal, decimal, or hexadecimal mode in computer programs where subscripts are not available.

```
1010B    (binary)
17Q      (octal)
17D      (decimal)
17H      (hexadecimal)
```

(The letter Q is used instead of an O for an octal number to avoid confusion with zero.)

Binary numbers such as

011001101111

can be difficult to read, so it is common practice to represent them in octal or hexadecimal form. Conversion to octal is easy if the bits are grouped by threes.

011	001	101	111	(binary)
3	1	5	7	(octal)

Grouping by fours facilitates the conversion to hexadecimal.

0110	0110	1111	(binary)
6	6	F	(hexadecimal)

NUMBER REPRESENTATION IN BINARY, BCD, AND ASCII

All information is ultimately stored in computers as a series of binary digits. There are, however, several different coding schemes for representing the original data. The simplest method is to use straight binary coding, as shown in Table 2.1. Notice that we might choose to represent a binary value in decimal, octal, or hexadecimal notation. The number itself is unchanged by this. The decimal number 12, for example, is stored as the binary number 1100.

A different method of representing data is called *binary coded decimal* (BCD). Actually, there are two types of BCD: unpacked and packed. With unpacked BCD, each byte contains a single decimal digit from 0 to 9. Packed BCD can have one or two decimal digits in each byte. Thus, a packed BCD number can range from 0 to 99. By comparison, an 8-bit binary number can range from 0 to 255. Table 2.2 shows the first 16 integers in BCD. The first column gives the decimal equivalent, the second column the corresponding bit pattern.

Table 2.2. The first 16 integers represented in decimal and binary-coded decimal (BCD).

decimal	BCD
0	0000 0000
1	0000 0001
2	0000 0010
3	0000 0011
4	0000 0100
5	0000 0101
6	0000 0110
7	0000 0111
8	0000 1000
9	0000 1001
10	0001 0000
11	0001 0001
12	0001 0010
13	0001 0011
14	0001 0100
15	0001 0101

Notice that the binary representation for the decimal numbers 0 through 9 is the same for both binary and for BCD.

A third method for encoding data is called ASCII. This scheme is commonly used with peripherals such as printers and video terminals. When the key labeled 2 of an ASCII console is pressed, the bit pattern

0011 0010

is generated. Table 2.3 gives the bit patterns for the ASCII digits 0-9 in binary and hexadecimal notation.

Table 2.3. The bit pattern for the ASCII digits 0-9.

digit	binary	hexadecimal
0	0011 0000	30
1	0011 0001	31
2	0011 0010	32
3	0011 0011	33
4	0011 0100	34
5	0011 0101	35
6	0011 0110	36
7	0011 0111	37
8	0011 1000	38
9	0011 1001	39

LOGICAL OPERATIONS

The fundamental operations of a computer involve electrical signals that can have only one of two values. The two voltage levels might be zero and 5 volts, for example, or they might be something else. The actual value is unimportant at this point. Instead, we refer to the two allowable states as TRUE and FALSE. The TRUE state is also called a logical 1, or high state, and the FALSE state is also known as a logical 0, or low state.

TRUE = 1 (high)
FALSE = 0 (low)

Computers store numbers in binary form as a series of 1s and 0s. These two possible values correspond to the two possible voltage levels of the electronic circuitry. We can therefore utilize the expressions TRUE and FALSE to describe the state of each bit.

The collection of transistors, resistors, and so forth that makes up the physical computer is called the *hardware*. The computer program used to direct the activities of the computer is termed the *software*. In this sense, the hardware and software are distinctly different. But sometimes we use these terms a little differently.

Consider, for example, one of the major differences between minicomputers and microcomputers. Minicomputers contain electronic circuitry for the multiplication of two numbers. Since microcomputers do not contain such circuitry, multiplication is performed instead by executing a special computer program. We say that minicomputers perform multiplication by hardware, but that microcomputers must do multiplication by software.

Hardware operations are performed by electronic devices called *gates.* The internal structure of the gate is unimportant if we are only interested in the logic of its operation. There are input signal lines that are sampled by the gate, and there is an output signal that is generated by the gate. When we consider the logical operations that are performed by a computer, we can imagine that they are accomplished either by hardware or by software. The answer is the same.

A common logical operation is the complement or inversion of a binary digit. The complement of 0 is 1 and the complement of 1 is 0. The hardware complement is performed with an inverter or NOT gate. The electronic symbol for this gate, shown in Figure 2.1, is a triangle with one apex to the right (usually) or to the left (sometimes). A small circle or triangle at this apex completes the symbol.

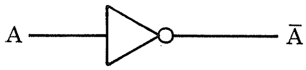


Figure 2.1. The electronic symbol for the NOT or inverter gate.

Letters of the alphabet are used to designate input or output signals. These binary signals can have one of two states, termed TRUE (1) or FALSE (0). The letter A with a bar over it ($\bar{A}$) represents the complement of A and is called NOT A. A truth table is used to summarize the possible states.

A	$\bar{A}$	or	A	$\bar{A}$
0	1		FALSE	TRUE
1	0		TRUE	FALSE

THE TWO'S COMPLEMENT

If each bit of an 8-bit byte is complemented, we produce a result that is termed the one's complement of the byte.

0000 1001 = 9
1111 0110 = one's complement of 9

Both the 8080 and the Z-80 CPUs provide an operation code for complementing the accumulator. A slightly different operation is the two's complement. It is obtained by incrementing (adding 1 to) the one's complement of a number. For example:

```
  0000 1001  =  9

  1111 0110  =  one's complement of 9
+ 0000 0001     add one
  ---------
  1111 0111  =  two's complement of 9
```

It is interesting to note that the sum of a number and its two's complement is zero.

```
  1010 1010  =  170

  0101 0101  =  one's complement of 170
+ 0000 0001     add one
  ---------
  0101 0110  =  two's complement of 170

  0101 0110  =  two's complement of 170
+ 1010 1010  =  170
  ---------
  0000 0000     sum
```

Adding the two's complement of a number produces the same result as subtracting the number itself. For example, we can subtract 170 from 223 by adding the two's complement of 170. The result is the same.

```
  1101 1111  =  223
- 1010 1010  =  170
  ---------
  0011 0101  =  53
```

or

```
  1101 1111  =  223
+ 0101 0110  =  2's complement of 170
  ---------
  0011 0101  =  53
```

The 8080 CPU can perform both addition and subtraction with 8-bit numbers and it can add 16-bit numbers, but there is no 16-bit subtraction operation. We can effectively perform a 16-bit subtraction, however, by adding the two's complement. Suppose that the HL register pair contains the decimal value 10,005 and we want to subtract 10,000 from it. The difference between 10,005 and 10,000 can be obtained by adding the two's complement. Consider the bit pattern for the number 10,000.

```
0010 0111 0001 0000  =  10,000
```

We first form the one's complement, then increment the result to form the two's complement.

```
  1101 1000 1110 1111  =  one's complement of 10,000
+ 0000 0000 0000 0001     add one
  -------------------
  1101 1000 1111 0000  =  two's complement of 10,000
```

Finally, we add this two's complement to the value in HL.

```
  0010 0111 0001 0101  =  10,005 in HL
+ 1101 1000 1111 0000  =  two's complement of 10,000
  -------------------
  0000 0000 0000 0101     difference (sum) is 5
```

When an assembler encounters a negative argument, it will automatically calculate the corresponding two's complement. Thus the 8080 expression

```
LXI  =  D,-10000
```

will place the bit pattern

```
1101 1000 1111 0000
```

in the DE register pair. The instruction

```
DAD     D
```

will then effectively perform a 16-bit subtraction on the number in HL.

LOGICAL OR AND LOGICAL AND

In the previous section, we considered the logical operation of NOT. Two other important logical operations are OR and AND. Both of these operations reflect the usual English meaning. The logical OR of two bits results in a value of TRUE (1) if either or both the original values are TRUE. The result is FALSE otherwise. The logical AND of two values gives an answer of TRUE (1) if and only if both of the original values are TRUE. If either or both the original values are FALSE, then the answer is FALSE.

Equations of logical operations can be written using the appropriate symbols. Two OR operators are in common use: a plus symbol and a V-shaped symbol. The AND operator is either a dot or an inverted V. The schematic representations of the OR and AND gates are shown with their corresponding mathematical representations in Figure 2.2.

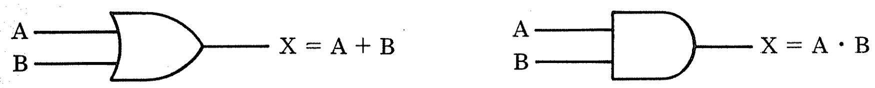


Figure 2.2. The OR and the AND gates.

The truth table is

		(OR)	(AND)
A	B	A+B	A·B
0	0	0	0
0	1	1	0
1	0	1	0
1	1	1	1

where zero means FALSE and 1 means TRUE. The origin of the + symbol for the OR operation and · symbol for the AND operation can be seen from the truth table. Logical operations are performed separately on each bit, and there is never a carry. The logical OR (sum) of A and B gives zero if both bits are zero, but 1 otherwise. (Binary digits can't be larger than 1.) The logical AND (product) of A and B gives zero if either or both bits are zero and unity otherwise.

SETTING A BIT WITH LOGICAL OR

Sometimes, we need to set one or more bits of the accumulator. We can use the logical OR operation for this purpose. From the truth table in the previous section, we can see that a logical OR of 1 with either a 0 or a 1 will give a result of 1.

A	B	A+B
1	0	1
1	1	1

Thus, a logical OR of any bit with a 1 will set that bit. On the other hand, a logical OR of 0 and another bit gives the result of that other bit.

A	B	A+B
0	0	0
0	1	1

In this case, the second bit is not changed.

Suppose that the accumulator contains a binary 5 and we want to convert it to an ASCII 5.

```
0000 0101 = binary 5
0011 0101 = ASCII 5
```

If we compare the two bit patterns, we can see that they are the same except for bits 4 and 5. These bits can be set by executing a logical OR with an ASCII zero.

```
   0000 0101 = binary 5
OR 0011 0000 = ASCII zero
   ---------
   0011 0101 = ASCII 5
```

The OR operation has set the bit corresponding to the location of the 1, but it has left the other bits unchanged.

A logical OR of a register with itself does not change the value.

```
   0101 1010 = 5A hex
OR 0101 1010 = 5A hex
   ---------
   0101 1010 = 5A hex
```

But this operation can be used to set the flags. In this example, the zero, carry, and sign flags are reset and the parity flag is set.

RESETTING A BIT WITH LOGICAL AND

A logical AND operation can be used to reset any particular bit of the accumulator; the truth table shows how. A logical AND of 0 and either a 0 or a 1 will always give a result of 0.

A	B	A·B
0	0	0
0	1	0

Thus, the bit is reset. On the other hand, a logical AND of 1 and another bit will give the value of the other bit.

A	B	A·B
1	0	0
1	1	1

Thus the AND instruction can be used to reset or "turn off" particular bits. This step is sometimes called a *masking* AND operation.

When the CPU reads an ASCII character from the console, it gets an 8-bit byte. But since the ASCII code contains only 7 bits, the high-order bit is not needed. The console-input routine typically resets this bit by performing a masking AND operation. Suppose that the console transmitted an ASCII 5 with the high-order bit set. The bit pattern looks like this.

```
1011 0101
```

The high-order bit can be reset with an AND operation.

```
     1011 0101 (original byte)
AND  0111 1111 (mask)
     ---------
     0011 0101 (ASCII 5)
```

LOGICAL EXCLUSIVE OR

The ordinary OR operation is sometimes called an inclusive-or operation to distinguish it from the exclusive OR (XOR) operation. For this latter operation, the result is TRUE only if the corresponding bits of both values are different. Either A or B must be TRUE, but not both. The XOR operation is represented by a plus symbol surrounded by a circle. The complement of the XOR is the exclusive NOR or XNOR. It can be used as a comparator. The hardware implementation is sometimes used in circuitry to enable memory boards. The result is TRUE if and only if both corresponding bits are identical. The result is FALSE otherwise. The truth table is:

A	B	$A \oplus B$	$\overline{A \oplus B}$
0	0	0	1
0	1	1	0
1	0	1	0
1	1	0	1

The exclusive OR of a bit with itself will always be FALSE. Therefore the XOR of the accumulator with itself will set it to zero.

```
     0111 1100 = 7C hex
XOR  0111 1100 = 7C hex
     ---------
     0000 0000 = zero
```

The corresponding electronic symbols for the hardware implementation of the XOR and XNOR are shown in Figure 2.3.

Figure 2.3. The exclusive or (XOR) and comparator (XNOR) gates.

LOGICAL NAND AND NOR GATES

By combining an inverter gate in series with the AND and OR gates, a new set of gates is formed. The NOT AND gate is called a NAND gate; it is shown in Figure 2.4. The NOT OR gate is known as a NOR gate; it is shown in Figure 2.5.

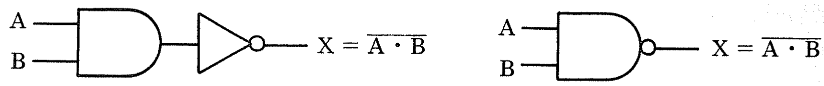


Figure 2.4. The NAND gate can be produced from an AND gate and a NOT gate.

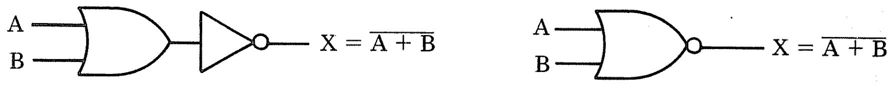


Figure 2.5. The NOR gate can be formed from the OR gate and the NOT gate.

From the truth table, it can be seen that the outputs of the NOR and NAND gates are the inverse of the corresponding OR and AND gates.

A	B	$\overline{A+B}$	$\overline{A \cdot B}$
0	0	1	1
0	1	0	1
1	0	0	1
1	1	0	0

If both inputs of the NOR gate are connected together, then the gate behaves like a NOT gate. The same is true for the NAND gate. This can be seen by comparing the first and last rows of the truth tables. In this way, two NOR gates can be combined serially to produce an OR gate. The result is a NOT NOT OR gate that is equivalent to an OR gate. This is shown in Figure 2.6. In a similar way, two NAND gates can be used to make an AND gate as shown in Figure 2.7. Since OR and AND gates cannot be similarly combined to produce the NOR and NAND gates, we will find that NAND and NOR gates are more common.

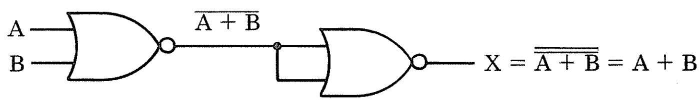


Figure 2.6. An OR gate is formed from two NOR gates.

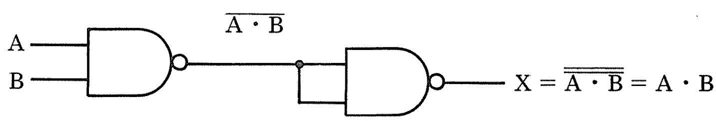


Figure 2.7. Two NAND gates are combined to produce an AND gate.

MAKING OTHER GATES

NOR and NAND gates are very versatile. NOR gates or NAND gates can be combined to produce all of the other gates. This can be seen from the following truth table.

A	B	$\overline{A}$	$\overline{B}$	A+B	A·B	$\overline{A}+\overline{B}$	$\overline{A}\cdot\overline{B}$	$\overline{A+B}$	$\overline{A\cdot B}$
0	0	1	1	0	0	1	1	1	1
0	1	1	0	1	0	1	0	0	1
1	0	0	1	1	0	1	0	0	1
1	1	0	0	1	1	0	0	0	0

Notice that column 7 of the truth table has the same values as the last column. Similarly, columns 8 and 9 are identical. These relations follow De Morgan's theorem, which can be expressed mathematically as:

$$\overline{A}+\overline{B}=\overline{A\cdot B} \quad \text{and}$$
$$\overline{A}\cdot\overline{B}=\overline{A+B}$$

The corresponding digital gates are shown in Figures 2.8 and 2.9.

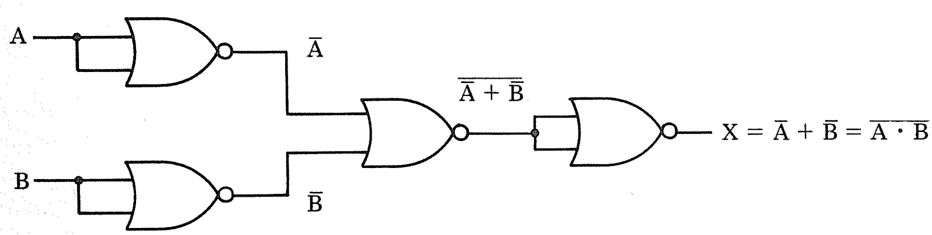


Figure 2.8. A NAND gate is formed from four NOR gates.

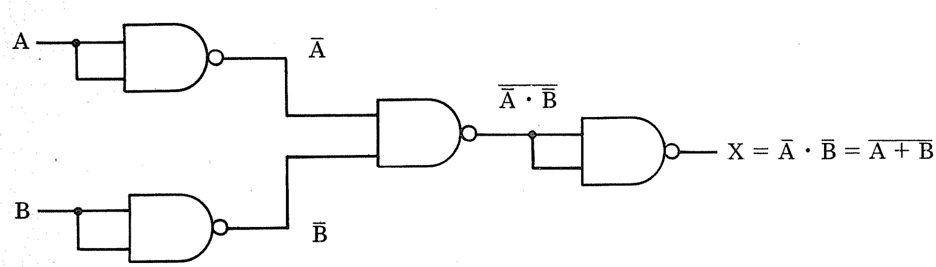


Figure 2.9. A NOR gate is obtained from four NAND gates.

The use of a small circle to represent inverted output brings up another approach to the understanding of digital logic gates. In the more commonly used system, the small circles are used only on the output side of the gate.

Another approach, however, is to always connect active-high outputs to active-high inputs, and active-low outputs to active-low inputs. For this latter system, NAND gates will sometimes appear as OR gates with inverted inputs, and NOR gates will sometimes appear as AND gates with inverted inputs. According to De Morgan's theorem, the NAND gate is equivalent to the OR gate with inverted input signals. This is demonstrated in Figure 2.10. The circuit shown is logically the same as the one shown in Figure 2.9. Notice that the active-low outputs of the first NAND gates are connected to the active-low inputs of the next OR gate. That is, there are small circles on the outputs of the first gates and on the inputs of the second gate.

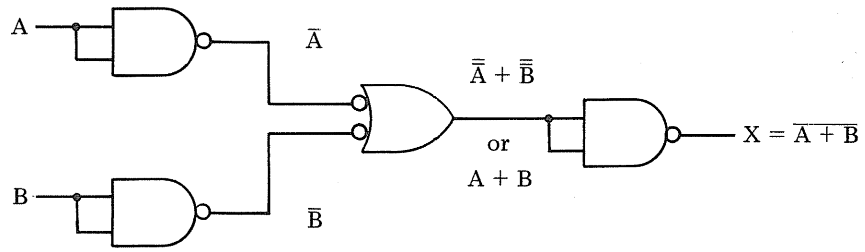


Figure 2.10. A NOR gate is produced from four NAND gates. The middle NAND gate is shown in its alternate representation.

CHAPTER THREE

The Stack

When main memory is used to store a collection of data, each member of the data set is individually accessible. This type of storage is termed *random access memory* (RAM). Magnetic tape storage, by contrast, is serial or *sequential access memory.* In this latter case, only one item of the set is available at any one time. There are two ways of storing and retrieving the items in a serial memory buffer: one is by means of a first-in, first-out (FIFO) buffer, and the other is by means of a last-in, first-out (LIFO) buffer. We can visualize the serial buffer as a long string of information. With the FIFO buffer, items are added at one end and removed from the other. This buffer is analogous to an escalator: the people who ride the escalator are like the data—those who get on first, get off first.

```
      In
      |
      v
!----------!
!   data   !
!----------!
!   data   !
!----------!
!   data   !
!----------!
!   data   !
!----------!
      |
      v
     Out
```

Figure 3.1. The first-in, first-out (FIFO) buffer.

With the LIFO buffer, on the other hand, the data are added and removed at the same place. This arrangement is analogous to a very long, narrow elevator. Those who get on first, have to wait until everyone else is off before they can get off. It can be seen that magnetic tape is a FIFO medium.

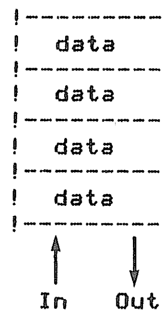


Figure 3.2. The last-in, first-out (LIFO) buffer.

Sometimes, a special area of main memory is designated as a LIFO buffer even though each member of the buffer is individually accessible. This region is known as a *stack*. As an example, Hewlett-Packard calculators utilize a very short LIFO stack, consisting of registers known by the letters Y, Z, and T. An item in any of the registers is individually accessible, yet the stack as a whole can be manipulated. As data is entered from the keyboard, it is placed into the X register. This information can then be transferred to the stack (register Y in this case) by pressing the ENTER key. We say that the contents of the X register are *pushed* onto the stack. Items can be retrieved from the stack and placed in the X register with the roll-down (R) key. We say that data are *popped* from the stack into the X register by this means. Another stack operation is performed by the EXCHANGE key which is used to swap the contents of the X and Y registers.

STORING DATA ON THE STACK

We have seen in the previous chapters that the 8080 and Z-80 microprocessors incorporate general-purpose registers for the storage of information. But these registers are limited in number. Consequently, a special area of main memory is designated for the additional storage of information. This area, called the stack, is implemented on the Z-80 and 8080 as a last-in, first-out serial buffer even though each item in the stack is individually accessible. One of the CPU registers, the stack pointer, references the current location in memory. This is the address of the most recently added item. The stack pointer is decremented as items are added and incremented as items are removed. The programmer may place the stack anywhere in memory by loading the stack pointer with the desired address. For example, the instruction

```
LD      SP,4000H        (Z-80)    or
LXI     SP,4000H        (8080)
```

initializes the stack to location 4000 hex.

Data can be placed on the stack with one of the PUSH operations. A command of

```
(Z-80)              (8080)
PUSH     HL         PUSH     H
```

will move a copy of HL to the stack. Since main memory is addressed eight bits at a time, the PUSH operation is actually performed in two stages. The stack pointer is decremented, then the H register (the high half) is copied to the stack. The stack pointer is decremented a second time and the L register (the low half) is copied to the stack. The stack pointer register now contains the address of the low byte. Figure 3.3 demonstrates the action of a PUSH HL command. The region of memory devoted to the stack is shown with higher memory upward. The arrow represents the stack pointer.

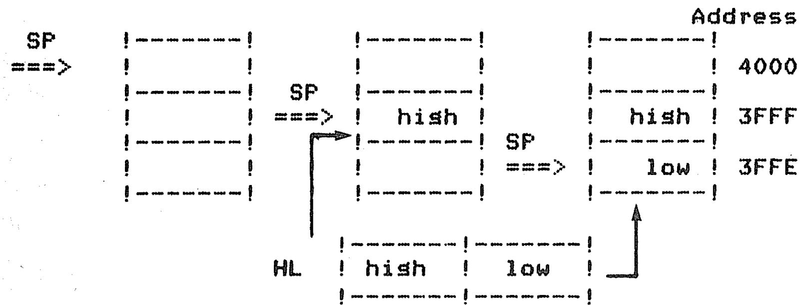


Figure 3.3. The HL register is pushed onto the stack.

The POP instruction reverses the PUSH process. For example, a POP DE command copies 16 bits from the stack into the DE register. Because the stack operates in a LIFO manner, the most recently added byte is removed first. This is placed into register E (the low half of the DE pair). The stack pointer is automatically incremented and the next byte is transferred from memory to register D (the high half). The stack pointer is then incremented a second time. Figure 3.4 demonstrates the operation. Notice that the data originally pushed onto the stack is still present.

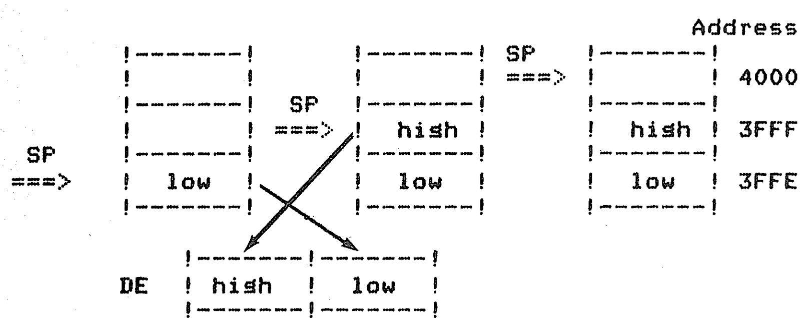


Figure 3.4. Two bytes are popped from the stack into DE.

It can be seen that the stack grows downward in memory as data are pushed into it, and it moves back up as data are popped off. For this reason, it is common practice to initialize the stack pointer to the top of usable memory. Actually, the stack pointer can start at one address above the top of memory since the stack pointer is always decremented before use.

If the general-purpose registers contain important information but they are needed for a calculation, it will be necessary to save the original data. This can be easily done by pushing the contents onto the stack. The registers are restored at the end of the calculation with the three corresponding POP commands. The operation goes like this.

```
PUSH    HL      ;save HL
PUSH    DE      ;save DE
PUSH    BC      ;save BC
. . .
. . .           ;do the calculation
. . .
POP     BC      ;restore BC
POP     DE      ;restore DE
POP     HL      ;restore HL
```

Notice that the order of the POP commands is reversed from that of the PUSH sequence. This is necessary because of the stack's LIFO operation.

```
         STACK
       !---------!           !---------!           !---------!
       !    H    !           !    H    !           !    H    !
 SP    !---------!           !---------!           !---------!
 ===>  !    L    !           !    L    !           !    L    !
       !---------!           !---------!           !---------!
                             !    D    !           !    D    !
                       SP    !---------!           !---------!
                       ===>  !    E    !           !    E    !
                             !---------!           !---------!
                                                   !    B    !
                                             SP    !---------!
                                             ===>  !    C    !
                                                   !---------!

 PUSH H                PUSH D                PUSH B
```

Figure 3.5. The contents of the general-purpose registers are saved on the stack.

```
 STACK
          !---------!          !---------!   SP   !---------!
          !         !          !         !  ===>  !         !
          !---------!          !---------!        !---------!
          !    H    !          !    H    !        !    H    !
          !---------!   SP     !---------!        !---------!
          !    L    !  ===>    !    L    !        !    L    !
          !---------!          !---------!        !---------!
          !    D    !          !    D    !        !    D    !
   SP     !---------!          !---------!        !---------!
  ===>    !    E    !          !    E    !        !    E    !
          !---------!          !---------!        !---------!
          !    B    !          !    B    !        !    B    !
          !---------!          !---------!        !---------!
          !    C    !          !    C    !        !    C    !
          !---------!          !---------!        !---------!

  POP B                POP D                POP H
```

Figure 3.6. The original contents of the general-purpose registers are restored from the stack.

THE ACCUMULATOR AND PSW AS A DOUBLE REGISTER

The 8-bit accumulator and the 8-bit flag register are treated as a 16-bit double register for the PUSH AF and POP AF instructions. In this case, the accumulator is treated like the high byte since it is pushed onto the stack first. The flag register is pushed onto the stack second. Figure 3.7 demonstrates this.

```
               stack               stack
   SP        !-------!           !-------!
  ===>       !       !           !       !
             !-------!           !-------!
             !       !           !   A   !  <-!
             !-------!   SP      !-------!    !
             !       !  ===>     ! flags !    !
             !-------!   !->     !-------!    !
                         !                    !
                         !       !-------!    !
                         !       !   A   !  >-!
                         !       !-------!
                         !-<     ! flags !
                                 !-------!

                  PUSH PSW              POP PSW
```

Figure 3.7. Contents of the accumulator and flag registers are pushed onto the stack.

Data can be moved from one register pair to another by using a PUSH/POP combination. For example, the two 8080 commands

```
PUSH  H
POP   D
```

will move H to D and L to E. This is not the most efficient way to accomplish the move, however. The sequence requires access to main memory and so is slower than the direct register moves

```
MOV   D,H
MOV   E,L
```

Z-80 INDEX REGISTERS

The Z-80 has two, 16-bit index registers that can participate in the PUSH and POP operations. However, the instructions each require two bytes compared to the other PUSH and POP instructions which only require one byte each. As a result, the execution time is slower than the other PUSH and POP instructions. There are no official instructions for moving data between the index registers and the general-purpose registers. This transfer can be performed, however, by use of the PUSH and POP commands. The two instructions

```
PUSH  IX
POP   BC
```

will copy the IX register into the BC register.

SUBROUTINE CALLS

We have seen that the PUSH instructions can be used by the programmer to store data on the stack. The 8080 and Z-80 CPUs use the stack for a second purpose: storing the return address when a subroutine is called. Subroutines are used to efficiently code a set of instructions needed at several different places in a computer program. A subroutine is called by using the assembly-language mnemonic CALL. At the end of the subroutine, indicated by the return statement, control is automatically returned to the calling program.

```
!---------!   Call      !------------!
! calling !  --------->  ! subroutine !
! program !  <---------  !------------!
!---------!   Return
```

The input and output routines which control the console may be needed at several locations in a program. Consequently, they are coded as subroutines. The 8080 assembly language subroutine for the console might look like this.

```
OUTPUT: IN      STATUS   ; CHECK STATUS
        ANI     INMSK    ; INPUT MASK
        JZ      OUTPUT   ; NOT READY
        MOV     A,B      ; GET DATA
        OUT     DATA     ; SEND DATA
        RET              ; DONE
```

Data can be output from anywhere in a program by placing the byte in the B register and calling the output subroutine. The following examples of 8080 assembly language mnemonics show how a question mark and a colon can be printed by calling the console output routine.

```
WHAT:   MVI     B,'?'    ;OUTPUT A ?
        CALL    OUTPUT
. . .
        . . .
COLON:  MVI     B,':'    ;OUTPUT A COLON
        CALL    OUTPUT
        . . .
```

The above examples utilize the unconditional subroutine call and unconditional return instructions. Conditional call and return instructions are also available. These commands perform the appropriate call or return only if the referenced PSW flag is in the desired state. The four flags—zero, sign, carry, and parity—give rise to eight conditions.

zero
not zero
plus
minus
carry
not carry
parity even
parity odd

These instructions are discussed in more detail in Appendix H.

The stack provides the mechanism for subroutine operation. When a CALL instruction is encountered, the address immediately following the CALL statement is automatically pushed onto the stack. The subroutine address is then loaded into the program counter register. The program counter tells the CPU which instruction to execute next. Since a subroutine CALL uses the stack, the programmer must be sure that the stack is properly defined prior to a subroutine CALL. When a return instruction is subsequently encountered, the return address is popped off the stack and placed into the program counter. After return from a subroutine, program execution continues with the instruction following the CALL statement.

PASSING DATA IMPROPERLY TO A SUBROUTINE

Since the stack can be used for storing both data and subroutine return addresses, the programmer must ensure that there are no conflicts. First, there should normally be as many POP instructions as PUSH instructions. Second, one must be careful not to PUSH data onto the stack, CALL a subroutine, then POP data off the stack. The LIFO nature of the stack will cause trouble in this case.

```
        PUSH    H
        CALL    ORDER >---!
        . . .             !
        . . .             !
ORDER:  . . .         <---!
        POP     H
        . . .
        RET     >----> ???? CRASH !
```

Figure 3.8 shows an example of improper mixing of data and the return address on the stack. Higher memory is upward and lower memory is downward. The arrow indicates the current stack pointer position.

```
          STACK
SP      !----------!          !----------!  SP   !----------!
===>    !   data   !          !   data   ! ===>  !   data   !
        !----------!   SP     !----------!       !----------!
                      ===>    ! address  !
                              !----------!
                                   !             !----------!
                                   !--->   H,L   ! address  !
                                                 !----------!

PUSH H              CALL ORDER              POP H              RET
```

Figure 3.8. Improper mixing of data and the return address on the stack.

In this example, the data is first pushed onto the stack while in the main program. The return address is then pushed onto the stack next, when the CALL instruction is encountered. The POP instruction in the subroutine will actually load the HL register pair with the subroutine return address rather than the data that was expected. This occurs because the data was pushed onto the stack before the return address. Worse yet, the RET instruction will load the program counter with the data, rather than with a useful address. Strange things are likely to happen when the CPU attempts to execute instructions at an address defined by the data.

PASSING DATA PROPERLY TO A SUBROUTINE

This section demonstrates a proper way to pass data into a subroutine by using the stack. The task can be accomplished with the 8080 XTHL instruction or the Z-80 EX (SP),HL instruction. This operation exchanges the HL register pair with the two bytes at the current stack position. The instruction is analogous to the X/Y EXCHANGE key on an HP calculator.

The method works in the following way. The data is pushed onto the stack while control is in the calling program. When the subroutine is called, the return address is pushed onto the stack, just after the data. A POP instruction, executed in the subroutine, delivers the return address to the HL register. Now, the XTHL instruction exchanges the HL register with the stack. The desired data is now in HL and the return address is on the stack. Finally, a return instruction will correctly return control to the calling program.

```
        PUSH    H        ; main program
        CALL    ORDER    ; call subroutine
        . . .
        . . .
ORDER:  . . .            ; start of subroutine
        POP     H        ; get return address
        XTHL             ; exchange with data
        . . .
        RET              ; return to main program
```

```
STACK !---------!  SP   !---------!  SP   !---------!
      !  data   ! ===>  !  data   ! ===>  ! address !
 SP   !---------!       !---------!       !---------!
 ===> ! address !
      !---------!

                        !---------!       !---------!
                  HL    ! address !       !  data   !
                        !---------!       !---------!

CALL              POP H                XTHL              RET
```

Figure 3.9. Proper mixing of data and return address on the stack.

It is important to note that the XTHL command only works with the HL register. There is no equivalent instruction for the DE or BC registers.

PASSING DATA BACK FROM A SUBROUTINE

A variation of the XTHL technique is also possible. Data can be pushed onto the stack from within a subroutine, then retrieved after returning to the calling program.

```
        CALL    FETCH
        POP     H       ;GET THE DATA
        . . .
        . . .
FETCH:  . . .
        LXI     H,DATA  ;PUT IN H,L
        XTHL            ;SWITCH STACK
        PUSH    H       ;RET ADDR
        . . .
        RET
```

DATA in this case is predefined and is part of the LXI instruction.

```
            stack                stack
 SP    !---------!    SP    !---------!
===>   ! address !   ===>   !  data   !
       !---------!          !---------!

       !---------!          !---------!
 HL    !  data   !          ! address !
       !---------!          !---------!

  CALL                  XTHL


STACK
       !---------!    SP    !---------!
       !  data   !   ===>   !  data   !
 SP    !---------!          !---------!
===>   ! address !
       !---------!

PUSH H                 RET
```

Figure 3.10. Using the stack to pass data back from a subroutine.

An extension of the XTHL technique allows additional data to be passed on the stack.

```
         CALL    FETCH
         POP     B       ;DATA 3
         POP     D       ;DATA 2
         POP     H       ;DATA 1
         . . .
         . . .
FETCH:   . . .
         LHLD    DATA1   ;DATA1 TO H,L
         XTHL            ;SWITCH STACK
         XCHG            ;STACK TO DE
         LHLD    DATA2   ;GET DATA2
         PUSH    H       ;PUT ON STACK
         LHLD    DATA3   ;GET DATA3
         PUSH    H       ;PUT ON STACK
         PUSH    D       ;RET ADDR TO STACK
         RET
```

In this example, the return address is first moved to the HL pair with the XTHL command. Then it is moved to the DE register pair with the XCHG instruction. Three sets of 16-bit data are obtained from the memory addresses pointed to by the arguments of the LHLD instructions DATA1, DATA2, and DATA3. The first set is placed on the stack with the XTHL command. Then the other two are pushed onto the stack. Next, the return address, previously saved in the DE register pair, is pushed onto the stack. A final RET instruction pops the return address from the stack into the program counter.

SETTING UP A NEW STACK

Sometimes it is desirable to save the current stack pointer and set up a new one. When this happens, the original stack pointer is restored at the conclusion of the task. The technique is particularly useful when one independent program is executed by another. The original stack pointer is saved in a memory location, then retrieved at the end of the program.

If the current program was reached through a subroutine call, the return address for the calling program should be the current address on the original stack. It is this address that must be saved.

There is a Z-80 instruction that allows the old stack pointer to be stored directly in main memory. The instruction looks like this.

```
; Z-80 VERSION
;
START:   LD      (OLDSTK),SP ;save stack
         LD      SP,STACK    ;new stack
         . . .
         . . .
         LD      SP,(OLDSTK) ;set  old stack
         RET                 ;done
```

At the conclusion of the task, the old stack pointer is restored. With an 8080 CPU, the job is more complicated since the stack pointer cannot be directly saved. In this case, the stack pointer is moved to the HL register pair which is in turn saved in memory. This is done by first zeroing HL, then adding in the stack pointer. At the end of the routine, the old stack pointer is loaded into the HL register pair then copied into the stack pointer register. Finally, a RET instruction is given.

```
START:  LXI     H,0         ;zero HL
        DAD     SP          ;SP to HL
        SHLD    OLDSTK      ;save stack
        LXI     SP,STACK    ;new stack
        . . .
        . . .
        LHLD    OLDSTK      ;get old stack
        SPHL                ;restore stack
        RET
```

CALLING A SUBROUTINE IN ANOTHER PROGRAM

A program may need to call a subroutine that resides in another program. But if the second program is revised, the subroutine address in the second program will change. This means that the argument of the CALL statement in the first program will also have to be changed.

There are two ways to solve this problem. One method is to provide a jump instruction near the beginning of the second program. The address of the jump instruction will always be the same. However, its argument, the internal subroutine address, can change from one version to the next. The first program simply calls CHEK2, and CHEK2 causes a jump to CHEK, the desired subroutine. The RET instruction at the end of CHEK will effect a proper return to program 1.

```
                 . . .                ; Program 1
!--------        CALL      CHEK2      ; call Program 2
!                . . .                        <-----------!
!                . . .       .                            !
!      START:    JMP       CONTIN     ;start of Program 2 !
!->    CHEK2:    JMP       CHEK         >-------------!   !
                 . . .                                !   !
                 . . .                                !   !
       CHEK:     . . .                        <-------!   !
                 RET                  ;to Program 1 >-----!
```

Of course, the second program may need to save the incoming stack, then restore it before returning to program 1.

A second solution is to place just the two-byte address of the subroutine near the beginning of the second program.

```
START:  JMP     CONTIN   ; Program 2
CHEK2:  DW      CHEK
```

Now the calling program must put its own return address on the stack and get the address of CHEK into the program counter. The following example is a way to do this. Notice that program 1 does not enter program 2 with a CALL instruction. It uses instead the PCHL instruction which copies the contents of HL into the program counter.

```
        PUSH   H        ;SAVE H,L
        LXI    H,NEXT   ;RET ADDR
        PUSH   H        ;ONTO STACK
        LHLD   CHEK2
        PCHL            ;INTO PC
NEXT:   POP    H        ;ORIG H,L
```

CALLING ONE SUBROUTINE FROM ANOTHER

A subroutine called by a main program may in turn call another subroutine. When the first subroutine, SUB1, is called, the return address to the main program, MAINA, is pushed onto the stack. When the second subroutine, SUB2, is called, the return address SUB1A is next pushed onto the stack. After the second subroutine has been called, there will be two return addresses on the stack: one to get back to SUB1 from SUB2, and the other to get back to the main program from SUB1.

```
          CALL    SUB1     ;MAIN
MAINA:    . . .    <---!
          . . .        !
; SUBROUTINE 1         !
;                      !
SUB1:     . . .        !
          CALL    SUB2 !
SUB1A:    . . .        !     <------!
          RET      >--!             !
;                                   !
; SUBROUTINE 2                      !
;                                   !
SUB2:     . .                       !
          . . .                     !
          RET                >------!

STACK
 SP    !---------!         !---------!  SP   !---------!
===>   !  MAINA  !         !  MAINA  ! ===>  !  MAINA  !
       !---------!   SP    !---------!       !---------!
                    ===>   !  SUB1A  !
                           !---------!

CALL SUB1      CALL SUB2             RET               RET
```

Figure 3.11. One subroutine calls another.

BYPASSING A SUBROUTINE ON RETURN

It may be that an operation in the sectond subroutine SUB2 makes it desirable to return directly to the main program from SUB2, bypassing SUB1. This is easily accomplished if the stack pointer is raised by two bytes before executing the return instruction. Of course, care should be taken to see if data has been pushed onto the stack after one or both return addresses were placed on the stack. The one-byte instruction to increment the stack pointer (INX SP) can be executed twice, to raise the stack pointer two bytes. Alternately, a one-byte POP command can be used if there is a free register pair available.

```
STACK
              !---------!  SP   !---------!
              !  MAIN2  !  ===> !  MAIN2  !
   SP         !---------!       !---------!
  ===>        !  SUB1A  !
              !---------!

CALL  SUB2              POP  H            RET
```

Figure 3.12. Skipping one level of subroutine during the return.

Suppose that an ordinary return from subroutine SUB2 back to subroutine SUB1 is desired if the zero flag is set to 1. On the other hand, an unusual return directly back to the main program is desired if the zero flag is reset to a value of zero. Here is a way to do this.

```
         CALL     SUB1     ;MAIN
MAIN1:   . . .                         <---------!
         . . .                                   !
; SUBROUTINE 1                                   !
;                                                !
SUB1:    . . .             ;SUBROUTINE 1         !
         CALL     SUB2                <---!      !
SUB1A:   . . .                            !      !
         RET                              !      !
                                          !      !
; SUBROUTINE 2                            !      !
;                                         !      !
SUB2:    . . .                            !      !
         . . .                            !      !
         RZ                ;NORMAL RETURN >-     !
         POP      PSW      ;RAISE STACK          !
         RET               ;SKIP TO MAIN >-------!
```

The POP PSW instruction raises the stack two bytes so that the final RET instruction delivers the return address of MAIN1 to the program counter. This effectively bypasses the intermediate subroutine.

A PUSH WITHOUT A POP

Near the beginning of the system monitor, explained in Chapter 6, there is a restart address called WARM. The program normally branches back to this point at the conclusion of each task. Thus the final instruction of each task could be:

```
JMP     WARM
```

A more efficient method, however, is to push this restart address WARM onto the stack at the beginning of the task. Then if the task does not terminate within a subroutine, a simple return instruction, rather than a jump, can be given at the end of the task. This causes a branch back to WARM.

```
WARM:   LXI     H,WARM   ;H,L = HERE  <----!
        PUSH    H        ;ONTO STACK       !
        . . .                              !
        . . .                              !
        JZ      OPORT                      !
        . . .                              !
OPORT:  . . .                              !
        RET        >-----------------------!
```

This example is an exception to the rule that we should have a POP instruction for every PUSH. Here, there is a PUSH but no POP. Of course there is also a RET with no CALL. So everything is all right—or is it? What happens if termination occurs from a subroutine?

GETTING BACK FROM A SUBROUTINE

If a particular task terminates in a subroutine, then this subroutine's return address must be popped off the stack (or an INX SP instruction must be executed twice) before the return is issued.

```
WARM:   . . .                       <---------!
        JZ      DUMP                          !
        . . .                                 !
DUMP:   . . .                                 !
        CALL    TSTOP                         !
        . . .          <----------------!     !
        . . .                           !     !
TSTOP:  . . .                           !     !
        RNC            ;NORMAL RETURN >-!     !
        POP     H      ;RAISE STACK           !
        RET            ;TO WARM     >---------!
```

The stack pointer grows downward through memory during use. It is therefore common practice to place the stack as high as possible in available memory. But the system monitor may be located at the actual top of memory. In this case the stack can initially be placed lower in memory at the beginning of the monitor.

```
START:
          LXI       SP,START
```

On the other hand, the monitor may be placed in read-only memory (ROM). In this case, the stack can be located at the actual top of read/write memory. (While both read/write memory and ROM are random-access memory—RAM, it is customary to refer to read/write memory as RAM and read-only memory as ROM. This convention will be followed here.)

AUTOMATIC STACK PLACEMENT

The placement of the stack at the top of RAM can be done automatically, so that the total amount of RAM can be changed without having to reprogram the PROM monitor. A short routine can test each block of memory starting at zero until it finds a location that can't be changed. The stack is then put at the beginning of this block. Remember, the stack pointer is always decremented before use; therefore, it can be initially defined as one location above usable memory.

The first part of the program is a memory search routine that starts at address zero. It moves the byte from that location into the accumulator, complements it, then moves the complemented byte back to the original location. A comparison is made to see if the memory location does indeed contain the complemented byte. If it does, the accumulator is complemented back to the original byte and returned to memory. Such an algorithm is often called a nondestructive memory test.

The first byte of each subsequent block of memory is checked in this way until a failure is found. This will usually reflect the top of usable memory, but of course, it could indicate defective memory. The following program will work properly if placed in read-only memory.

```
;
; ROUTINE TO AUTOMATICALLY PLACE THE
; STACK AT THE TOP OF MEMORY
; 8080 CODE
;
          LXI       H,0       ;FIRST ADDR
NEXTP:    MOV       A,M       ;GET BYTE
          CMA                 ;COMPLEMENT
          MOV       M,A       ;PUT IT BACK
          CMP       M         ;COMPARE?
          JNZ       TOP       ;NO, DONE
          CMA                 ;BACK TO ORIG
          MOV       M,A       ;PUT IT BACK
          INR       H         ;NEXT BLOCK
          JMP       NEXTP     ;KEEP GOING
;
TOP:      SPHL                ;SET STACK
          CALL      OUTHL     ;PRINT IT
          . . .
```

This program might not work, however, if it is placed in read/write memory. The problem occurs because the routine is changing various locations in memory. If it happens to change its own instructions, then the results will be unpredictable.

The shortcomings of the previous program are solved with the following version. The improved version will operate properly no matter where it is placed. The stack will be placed at the top of contiguous RAM unless the routine itself is in that part of memory. In that case, the stack will be placed at the beginning of the program. The Z-80 version is shown, but the program can be run on an 8080 if two minor changes are made. The relative jump instruction must be changed to an absolute jump and the DJNZ instruction must be changed to the equivalent DCR B and JNZ combination.

```
;
; ROUTINE TO AUTOMATICALLY PLACE THE
; STACK AT THE TOP OF MEMORY
; FAILSAFE VERSION  (Z-80 CODE)
;
START:  LD      HL,0      ;START CHECK AT 0
        LD      B,START SHR 8
NEXTP:  LD      A,(HL)    ;GET BYTE
        CPL               ;COMPLEMENT IT
        LD      (HL),A    ;PUT IT BACK
        CP      (HL)      ;DID IT GO?
        JR      Z,TOP     ;NO, DONE
        CPL               ;BACK TO ORIG
        LD      (HL),A    ;RESTORE
        INC     H         ;NEXT BLOCK
        DJNZ    NEXTP     ;ARE WE HERE?
;
TOP:    LD      SP,HL     ;SET STACK
        . . .
```

The new version works in the following way. The B register initially contains the block number of the routine itself. The value in B is decremented as each successive block is checked. If the routine is in ROM, then the end of usable memory will be found, as in the previous version. The program will loop between the label NEXTP and the DJNZ NEXTP instruction. At some point, the CP (HL) instruction will reset the zero flag and the computer will jump to the address of TOP. The stack will then be placed at the top of RAM.

Alternately, if this routine is placed in the lower memory area, then the DJNZ instruction will decrement the B register all the way to zero. The zero flag will be set and the program will move on to TOP. Now the stack will be set to the beginning of the memory block that contains the program itself.

The START SHR 8 expression at the beginning of the routine instructs the assembler to calculate the high byte of the address of START and make it the second operand of the LD B instruction. It does this by shifting the address of START by eight bits to the right, then taking the low-order eight bits of the result. Some assemblers allow an equivalent operand of

```
HIGH    START
```

which is easier to comprehend. This automatic stack routine is incorporated into the system monitor explained in Chapter 6.

CHAPTER FOUR

Input and Output

Computers would not be very useful if they could not interact with the outside world. Commands and data are sent to the computer from the keyboard, magnetic tape, disk, and other peripherals. Results of computations are sent back from the computer to the printer, video terminal, tape unit, disk, and so on. Such input and output (I/O) transfers on a microcomputer are typically accomplished through special memory locations called I/O *ports.* One type of port is distinctly different from main memory. The other type of arrangement utilizes one of the regular main memory locations. The peripheral in this latter case is then said to use *memory-mapped* I/O. Each method has advantages and disadvantages. In either case, the I/O port will transfer eight bits, the natural word size for the 8080 and Z-80 CPUs.

MEMORY-MAPPED I/O

The I/O instructions on the 8080 microprocessor are rather limited compared to memory operations. There is a single IN and a single OUT instruction for transferring eight bits of data. In contrast, there is a much larger collection of memory operations available.

```
(8080 Mnemonics)           (Z-80 mnemonics)
    STA      80              LD       (80),A
    LDA      81              LD       A,(81)
    MOV      M,C             LD       (HL),C
    STAX     D               LD       (DE),A
    SHLD     84              LD       (84),HL
```

These additional instructions can be utilized with memory-mapped I/O, greatly increasing the versatility of the Z-80 and 8080 I/O operations.

The STA instruction stores the 8-bit accumulator value at the memory address specified by the operand. If this address corresponds to a memory-mapped port, then the byte is sent to the peripheral. The LDA command reverses the operation. It can be used to input a byte from a port. The MOV M,C instruction can be used to transfer a byte from the C register to the memory location designated by the HL register pair. The STAX D command moves a byte from the accumulator to the memory location designated by the DE register pair. The SHLD instruction opens a new dimension. Since this operation transfers 16 bits of data from the HL register pair directly into two consecutive memory locations, two adjacent ports can be simultaneously serviced.

The typical video console is a serial device that uses distinct ports. However, memory-mapped controller boards are commerically available. In this case, an ordinary TV set is then used for the video screen. There are also disk-controller boards that use memory-mapped operations to communicate with the disk drives. It is interesting to note that the Motorola 6600 CPU performs all of its I/O by memory mapping. There are no separate input or output instructions for this CPU.

DISTINCT DATA PORTS

Data ports may be designed to operate either in parallel or in serial fashion. Both the parallel and the serial I/O ports are connected to the computer through the system bus by a set of eight data lines. In addition, the parallel port is connected to the peripheral by another set of eight data lines. The serial port, by contrast, has only two data lines connecting it to the peripheral.

For some peripherals, such as a printer, data is transferred in only one direction. For others, such as the console and magnetic tape units, the peripheral is able to both send and receive data. In this latter case, there will be 16 data lines between the computer and the peripheral if a parallel port is used. Eight lines are used for sending data and eight are used for receiving data. The serial port, in contrast, will have three signal lines to the peripheral if there is two-way communication. One is for transmitting, one is for receiving, and the third is a common line for the other two.

There may be additional lines between the computer and the peripheral. One of these might indicate to the computer whether the terminal is operational. Another can be used to inform the terminal that the computer is ready. These extra lines are sometimes referred to as *handshake* lines.

The computer usually operates at a much higher speed than the peripherals. Consequently, there must be a mechanism for effectively slowing down the computer during I/O operations. For serial or parallel ports, this is typically accomplished by using two separate I/O ports for each peripheral device. One port is used for the data port and the other is used for the status port. Each of these two ports will have distinct addresses, one of the 256 values available to the 8080 or Z-80 CPU for this purpose. There are three general methods of performing I/O through data ports: looping, polling, and interrupting.

LOOPING

Looping is the simplest method of performing I/O through separate ports, and it is the one that is most commonly employed in 8080 and Z-80 programs. The CPU performs output by sending a byte to the data port using the OUT instruction. The corresponding status port is then read with an IN instruction. One bit of the 8-bit status port reflects the condition of the corresponding peripheral.

When the CPU places a byte in the data register, using the OUT command, the output status bit of the status register is set. This may actually result in a logical 1 or a logical zero, depending on the port design. When the peripheral utilizes the byte that was placed into the data register, the output status bit of the status register is reset. These changes in the status bit are automatically handled by the I/O interface hardware. However, the programmer must include in the software the appropriate routines for monitoring the status bits.

As an example of the looping method, consider the following subroutine:

```
COUT:   IN      10H     ;CHECK STATUS
        ANI     2       ;SELECT BIT
        JZ      COUT    ;NOT READY
        MOV     A,C     ;GET BYTE
        OUT     11H     ;SEND
        RET             ;DONE
```

This routine could be used to send a byte of data to the system console. The first instruction of the listing causes the CPU to read the 8-bit status port which has the address of 10 hex. The second instruction performs a masking AND operation to select the write-ready bit, bit 1. Remember that a logical AND with zero and anything else gives a result of zero. However, a logical AND with unity and a second logical value, gives the result of that second value.

Suppose that the output status is indicated by a logical 1 of bit 1, where bit 0 is the least-significant bit of the register. Then, a logical AND with the value in the status register and with the number 2 will result in a logical 1 if the peripheral is ready. If the device is not ready, however, the result is a logical 0.

```
        0101 0111     status     0101 0101
AND     0000 0010     = 2        0000 0010
        ---------                ---------
        0000 0000                0000 0010
          ready                  not ready
```

Thus, the logical AND with the value of 2 in the status register gives a result of zero if bit 1 (the second bit) is 0. Otherwise, a nonzero result is obtained.

The third instruction in the looping example is a conditional jump. If the peripheral is not ready, the JZ instruction will cause the computer to loop repeatedly through the first three lines until the peripheral is ready for another byte. At this point, the write-ready bit, bit 1, will be a logical 1. Then the logical AND operation, the second instruction of the subroutine, produces the nonzero value of 2. The MOV instruction following the conditional jump will then be executed. The byte to be outputted is moved to the accumulator, and then sent to data port 11 hex by use of the OUT command.

When the byte to be output is actually sent to the data port, the write-ready flag is reset to a logical zero. The output routine may be immediately reentered for outputting another byte, but now the peripheral is not ready. Looping will occur again through the first three instructions of the output routine since the write-ready flag has been reset to zero.

The CPU clock may be operating at 2 or 4 MHz. This rate is thousands of times faster than the speed of a typical printer. Consequently, if the looping method is used, the CPU will be spending over 99 percent of its time simply looping through the first three lines of the output subroutine. The computer will be spinning its wheels, so to speak, waiting on the peripheral.

Because the CPU is operating so much faster than the peripherals, it can, in principle, service many peripherals simultaneously. A very simple but useful implementation of this idea is found on the CP/M* operating system. In the CP/M system,* console output is normally sent only to the console. This terminal is typically a high-speed video device. But if the user types a Control-P, then the list device is also turned on. Console output will now appear simultaneously at both the console and the line printer.

This technique can be easily observed if the console video accepts data much faster than the line printer. Normally, as data is sent only to the console, it appears rapidly on the video screen. But when the list device is turned on, the output appears much more slowly. The reason for the slowdown is that both peripherals are operating at the speed of the slower one, in this case the printer.

A subroutine for accomplishing such a dual output might look like this.

```
LOUT:   IN      LSTAT   ;LIST STATUS
        ANI     2       ;OUTPUT MASK
        JZ      LOUT    ;LOOP UNTIL READY
        MOV     A,C     ;GET THE BYTE
        OUT     LDATA   ;SEND TO LIST
        OUT     CDATA   ;AND CONSOLE
        RET             ;DONE
```

*CP/M is a registered trademark of Digital Research, Inc., Pacific Grove, California.

This routine is not the one that is actually used in the CP/M system since, with our routine, the console will always display everything that is sent to the printer. This feature does not increase printing time as long as the console operates faster than the printer. Notice that there is no need to check the console status register. The output rate is set at the speed of the printer, and so the console, which operates so much faster, will always be ready if the printer is ready.

POLLING

One way to improve the performance, or throughput, of a CPU is with a technique known as polling. In this method, the CPU sends a byte to each of several different peripherals. Each peripheral operates at its full speed. Polling is more efficient than the looping method, and has been incorporated into several commercial 8080 software products. One product is a multiuser BASIC which can service up to four separate consoles. Each user can independently perform calculations using the same BASIC interpreter.

Another product that uses the polling technique is known as a *spooler.* The looping method is typically utilized for all output. In this case, all other activities must be halted while the printer is working. With a spooler program, however, things are different. When this program is incorporated into the system, the user can perform other tasks using the system console while a disk file is being printed.

In the polling method, the I/O routines are somewhat different from the corresponding routines of the looping method. The output-ready flag of the status register is checked periodically as with the looping method. But if the status flag indicates that the device is not ready, the CPU returns to perform some other task. Thus, the CPU does not waste time looping around the first three instructions of the input or output routine. A typical output routine using the polling method might look like tihis.

```
LOUT:   IN      LSTAT   ;CHECK STATUS
        ANI     LMASK   ;MASK FOR OUTPUT
        RZ              ;NOT READY
        MOV     A,C     ;GET THE BYTE
        OUT     LDATA   ;SEND IT
        RET
```

While the polling method is a great improvement over the looping method, there are still problems. For example, a decision must be made as to how often each status register will be polled. An even better method is to use hardware interrupts.

HARDWARE INTERRUPTS

The 8080 and Z-80 microprocessors incorporate a hardware interrupt system. This feature allows an external device, such as the system console or printer, to interrupt the current task of the processor. When the CPU is interrupted, it suspends its current task, and calls on one of several memory locations set aside for this purpose. The CPU services the request of the interrupting peripheral, then it returns to its previous task.

In this method, the CPU does not have to be programmed to check the peripherals on a regular basis as with the method of polling; nor does it have to waste time in a loop. Instead, the peripheral interrupts the processor when it needs service. If several peripherals are able to interrupt the CPU, then there must be a method for prioritizing the requests. This ordering is accomplished through a vectored interrupt system. For example, if a lower-priority device has interrupted the CPU for service, this phase can also be interrupted by a peripheral with a higher priority. On the other hand, a device with a lower priority cannot interrupt a higher-priority service, but must wait its turn.

Usually, the highest-priority interrupt will be assigned to updating the system clock. If the computer misses a beat, then the time will be incorrect. The next lower priority could be assigned to disk transfer. The printer could have a low priority since it is a relatively slow device, and it won't matter if it must slow down every so often.

Suppose that the printer is operated by interrupts rather than by looping or polling. The computer sends a byte to the printer, then continues with another task. When the current byte has actually been printed, the printer interrupts the CPU for another byte. In the time between the printing of two bytes, the CPU can perform many other tasks.

The console keyboard is another peripheral that can be readily serviced by an interrupt system. In this case, each time the user presses a key, the CPU is interrupted from its current task. Of course, if the CPU is currently servicing a higher-priority interrupt, then the console keyboard request will have to wait.

Both the 8080 and the Z-80 allocate eight addresses that can be used for the interrupt service routines. These addresses can be called by the eight, one-byte RST instructions.

Z-80	8080	Instruction code		Call
mnemonic	mnemonic	hex	binary	address
RST 00H	RST 0	C7	1100 0111	00H
RST 08H	RST 1	CF	1100 1111	08H
RST 10H	RST 2	D7	1101 0111	10H
RST 18H	RST 3	DF	1101 1111	18H
RST 20H	RST 4	E7	1110 0111	20H
RST 28H	RST 5	EF	1110 1111	28H
RST 30H	RST 6	F7	1111 0111	30H
RST 38H	RST 7	FF	1111 1111	38H

These instructions can be used as one-byte subroutine calls. As an example, suppose that the CPU executes an RST 5 instruction which corresponds to the instruction code EF hex. A subroutine call is then made to the corresponding address of 28 hex. The return address is pushed onto the stack, just as for a regular subroutine call. Subsequent execution of a return instruction will cause the program flow to return to the instruction immediately following the RST 5 instruction.

Hardware interrupts operate by emulating the software RST call. When an interrupt occurs, the CPU automatically disables the interrupt flip-flop, thus further interrupts are prevented. Then a subroutine call is made to the corresponding call address. This is done by jamming the desired RST code onto the data bus. The simplest implementation is to use a single interrupting device and the RST 7 instruction. (A normal interrupt always performs an RST 7.) The interrupting peripheral momentarily changes the state of the interrupt-request bus line. For the S-100 bus, this would require that bus line 73 be pulled to a zero-voltage state from the usual 5-volt level. The CPU responds by automatically calling memory address 38 hex. The programmer will have previously placed the service routine at this location. The service routine will conclude with a command to re-enable the interrupt flip-flop. Then a return instruction will be executed.

The trouble with this simple approach is that the RST 7 call to location 38 hex interferes with system debuggers because they also use this address. Consequently, another interrupt level is more suitable. Unfortunately, a single interrupt system always calls the RST 7 location. One solution to this problem is to use a vectored interrupt board. A vectored interrupt board allows the user to select up to eight separate interrupt levels corresponding to the RST 0 to 7 instructions. The disadvantage of this approach is the cost, since a vectored interrupt board may sell for several hundred dollars.

However, there is a low-cost solution. If only one interrupt level is required, a single hardware interrupt can be converted from an RST 7 to some other level such as an RST 5 by using only two logic gates. The circuit shown in Figure 4.1 will make the needed translation. The output of the two-input NAND gate IC-1 goes low when both of the input lines are high. One of these inputs is SINTA, line 96 on the S-100 bus. It is a CPU status signal that indicates acknowledgment of the interrupt request. The other input is PDBIN, bus line 78. This signal indicates that the data bus is in the input mode.

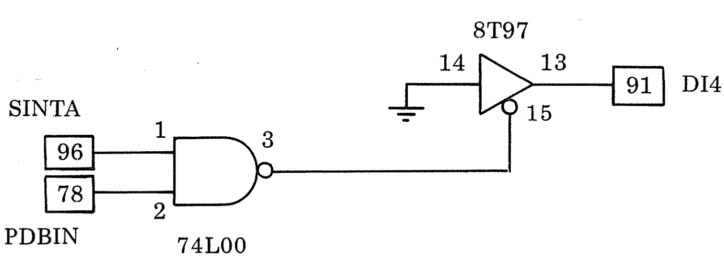


Figure 4.1. Circuit to convert an 8080 interrupt to an RST 5.

When the output of IC-1 goes low, it turns on the three-state buffer IC-2. This pulls the data-input bus line DI4 low. Since the remaining seven lines of the data-in bus are high, the CPU will see the value of

1110 1111

Notice that this is the bit pattern for the RST 5 instruction. The result is that the CPU executes an RST 5 instruction, by calling address 28 hex. The interrupt service routine, or a jump to it, is placed at this address.

AN INTERRUPT-DRIVEN KEYBOARD

We have seen that a printer operates considerably slower than a CPU. The console keyboard is even slower than the printer, especially if the operator is not an expert typist. Conversion to an interrupt-driven keyboard will considerably increase the effectiveness of a computer.

Characters entered on an interrupt-driven keyboard are temporarily stored in a memory buffer area. Each time a key is pressed on the console, the CPU is interrupted from its current task. The new byte is read and placed into the keyboard buffer. The computer then returns to its prior task. When the computer needs console input, it gets it from the input buffer, rather than from the console itself.

An interface program, utilizing a keyboard-interrupt approach, is shown in Listing 4.1. This program provides the necessary routines for interfacing the Lifeboart version of CP/M to a North Star disk system.* The portions of the program which specifically utilize the interrupt routines begin with a row of asterisks and end with a row of semicolons.

Computer pointer F400	Computer count F402	Keyboard count F403	Keyboard pointer F404	Buffer F406

Figure 4.2. The input buffer and pointers.

The layout of the memory buffer with its pointers is shown in Figure 4.2. The buffer area is arbitrarily chosen to start at the address of F400 hex. The location can be anywhere above the CP/M operating system. There is only one keyboard buffer, but there are two sets of pointers: one for the CPU and one for the keyboard. Two counters are also utilized; one shows how many characters have been entered from the keyboard and the other shows how many have been read by the computer. Since both sets of pointers grow larger, they need to be reset periodically. The two pointers are compared after each carriage return. If they are the same, then they are both reset to the beginning of the buffer.

Suppose that this interface program is incorporated into your system. CP/M might be printing something on the console video screen when a key on the console is pressed. A hardware interrupt will occur, causing the computer to stop its task and call address 28 hex (RST 5). A jump instruction at address 28 hex will transfer control to subroutine KEYBD. The keyboard

*Lifeboart Associates, 2248 Broadway, New York, N.Y. 10024.

Listins 4.1. Interrupt driven keyboard.

```
                        TITLE    'Interrupt CP/M BIOS'
                        ;
                        ; (Put todas's date here)
                        ;
                        ; LIFEBOAT VERSION WITH OPTION FOR
                        ; EITHER SINGLE OR DOUBLE DENSITY
                        ;
                        ; TERMINAL DEVICES SUPPORTED:
                        ;
                        ;  CONSOLE       10 HEX   CON:
                        ;  LIST.         12 HEX   LST:
                        ;  PHONE MODEM   14 HEX   PUN:
                        ;
0000 =                  FALSE    EQU      0
FFFF =                  TRUE     EQU      NOT FALSE
                        ;
FFFF =                  DOUBLE   EQU      TRUE      ;DOUBLE DENSITY
FFFF =                  INTRM    EQU      TRUE      ;INTERR VERSION
                        ;
                                 IF       DOUBLE
0036 =                  MSIZE    EQU      54        ;DECIMAL K
D600 =                  BIOS     EQU      MSIZE*1024-200H
DB00 =                  USER     EQU      BIOS+500H
4900 =                  OFFSET   EQU      1F00H-BIOS
                                 ELSE     ;SINGLE DENSITY
                        MSIZE    EQU      56
                        USER     EQU      MSIZE*1024-700H
                                 ENDIF
0003 =                  IOBYTE   EQU      3         ;I/O SETUP
000D =                  CR       EQU      0DH       ;CARRIAGE RET
000A =                  LF       EQU      0AH       ;LINEFEED
000C =                  FFEED    EQU      12        ;FORMFEED
0003 =                  CTRC     EQU      3         ;^C, KILL SCROLL
0004 =                  CTRD     EQU      4         ;^D, EMPTY BUFFER
0011 =                  CTRQ     EQU      17        ;^Q, SCROLL
0013 =                  CTRS     EQU      19        ;^S, FREEZE SCROLL
                        ;
                                 IF       DOUBLE
                        ; PATCH  DATE
D5B1                    ORG      BIOS-100H+0B1H
                                 ELSE
                        ORG      USER-600H+0AFH
                                 ENDIF
D5B1 2E4465                      DB       '.Jan 28,80' ;PATCH DATE
                        ;
DB00                    ORG      USER
                        ;
0010 =                  CSTAT    EQU      10H       ;CONSOLE STATUS
0011 =                  CDATA    EQU      CSTAT+1   ;CONSOLE DATA
0001 =                  CIMSK    EQU      1         ;INPUT MASK
0002 =                  COMSK    EQU      2         ;OUTPUT MASK
0012 =                  LSTAT    EQU      12H       ;LIST STATUS
0013 =                  LDATA    EQU      LSTAT+1   ;LIST DATA
0001 =                  LIMSK    EQU      1         ;INPUT MASK
0002 =                  LOMSK    EQU      2         ;OUTPUT MASK
0000 =                  LNULL    EQU      0         ;LIST NULLS
```

```
0014 =          MSTAT   EQU     14H       ;MODEM  STATUS
0015 =          MDATA   EQU     MSTAT+1   ;MODEM DATA
0040 =          MIMSK   EQU     40H       ;INPUT MASK
0080 =          MOMSK   EQU     80H       ;OUTPUT MASK
                ;
                ; INITIALIZE PORTS FOR COMPUTIME BOARD
                ;
00C4 =          ADATA   EQU     0C4H
00C5 =          ACONT   EQU     ADATA+1
00C6 =          BDATA   EQU     ADATA+2
00C7 =          BCONT   EQU     ADATA+3
                ;
                ;*************************************
                        IF      INTRM     ;INTERRUPTS
0095 =          STOP    EQU     95H       ;SET FOR INTERR
                ;
                ; CONSOLE INPUT-BUFFER LOCATION
                ;
F400 =          BUFFER  EQU     0F400H    ;INPUT BUFFER
F400 =          CPNTR   EQU     BUFFER    ;COMPUTER POINTER
F402 =          CCNT    EQU     CPNTR+2   ;BUFFER COUNT
F403 =          KCNT    EQU     CCNT+1    ;KEYBRD BUFF COUNT
F404 =          KPNTR   EQU     KCNT+1    ;KEYBOARD POINTER
F406 =          BUFF    EQU     KPNTR+2   ;INPUT BUFFER
0005 =          LEV     EQU     5         ;INTERR LEVEL
                        ENDIF             ;INTERRUPTS
                ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
                ;
                START:
DB00 C315DB             JMP     INIT      ;INITIALIZATION
DB03 C3ABDB             JMP     CONST     ;CONSOLE STATUS
DB06 C3D3DB             JMP     CONIN     ;CONSOLE INPUT
DB09 C312DC             JMP     CONOUT    ;CONSOLE OUTPUT
DB0C C326DC             JMP     LOUT      ;LIST OUTPUT
DB0F C348DC             JMP     PUNCH
DB12 C3D3DB             JMP     CONIN     ;FOR READER
                ;
                ; INITIALIZATION ROUTINES
                ;
DB15 3E03       INIT:   MVI     A,3
DB17 D310               OUT     CSTAT     ;RESET
DB19 D312               OUT     LSTAT     ;INTERFACE
DB1B 3E15               MVI     A,15H
DB1D D312               OUT     LSTAT
                ;
                        IF      INTRM
DB1F 3E95               MVI     A,STOP    ;SET FOR INTERR.
                        ENDIF
                ;
DB21 D310               OUT     CSTAT     ;INTERFACE
                ;
                ; COMPUTIME BOARD INITIALIZATION
                ;
DB23 AF                 XRA     A         ;GET A ZERO
DB24 D3C5               OUT     ACONT
DB26 D3C7               OUT     BCONT
DB28 3E70               MVI     A,70H
DB2A D3C4               OUT     ADATA
```

```
DB2C 3E77           MVI     A,77H
DB2E D3C6           OUT     BDATA
DB30 3E14           MVI     A,14H
DB32 D3C5           OUT     ACONT
DB34 3E04           MVI     A,4
DB36 D3C7           OUT     BCONT
              ; ***********************************
                    IF      INTRM
              ;
              ; PATCH RST LOCATION TO JUMP TO KEYBD
              ;
DB38 F3             DI              ;DISABLE INTERR
DB39 3EC3           MVI     A,0C3H  ;JMP INSTR
DB3B 322800         STA     8*LEV   ;PATCH RST
DB3E E5             PUSH    H
DB3F 215BDB         LXI     H,KEYBD ;INTERR ENTRY
DB42 222900         SHLD    8*LEV+1 ;JUMP HERE
DB45 2106F4         LXI     H,BUFF  ;BUFFER ADDR
DB48 2204F4         SHLD    KPNTR   ;RESET POINTERS
DB4B 2200F4         SHLD    CPNTR
DB4E 210000         LXI     H,0     ;2 ZEROS
DB51 2202F4         SHLD    CCNT    ;ZERO THE COUNTS
DB54 E1             POP     H
DB55 FB             EI              ;RE-ENABLE INTERR
                    ENDIF           ;INTERRUPTS
              ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
              ;
              ; INITIALIZE IOBYTE
              ;
DB56 AF             XRA     A       ;RESET IOBYTE
DB57 320300         STA     IOBYTE
DB5A C9             RET
              ;**************************************
                    IF      INTRM
              ;
              ; INTERRUPT ENTRY FOR KEYBOARD INPUT
              ;
DB5B F5       KEYBD: PUSH   PSW
DB5C DB10           IN      CSTAT   ;CONSOLE STATUS
DB5E E601           ANI     CIMSK
DB60 CA92DB         JZ      KEY2    ;NOT READY
DB63 DB11           IN      CDATA   ;GET DATA
DB65 E67F           ANI     7FH     ;MASK PARITY
              ;
              ; CHECK FOR ^S, ^Q SCROLL CONTROL
              ;
DB67 FE13           CPI     CTRS    ;^S
DB69 C27FDB         JNZ     KEY3    ;NO
DB6C DB10     KEY4: IN      CSTAT   ;CHECK KEYBOARD
DB6E E601           ANI     CIMSK   ;READY?
DB70 CA6CDB         JZ      KEY4    ;LOOP UNTIL READY
DB73 DB11           IN      CDATA   ;GET BYTE
DB75 E67F           ANI     7FH     ;STRIP PARITY
DB77 FE11           CPI     CTRQ    ;^Q?
DB79 C26CDB         JNZ     KEY4    ;NO
DB7C C392DB         JMP     KEY2
              ;
```

```
DB7F E5         KEY3:   PUSH    H
DB80 FE04               CPI     CTRD    ;EMPTY BUFFER?
DB82 CA95DB             JZ      KEY6    ;YES
DB85 2A04F4             LHLD    KPNTR   ;BUFFER POINTER
DB88 77                 MOV     M,A     ;PUT IT THERE
DB89 23                 INX     H       ;INR POINTER
DB8A 2204F4             SHLD    KPNTR   ;SAVE POINTER
DB8D 2103F4             LXI     H,KCNT  ;GET COUNT
DB90 34                 INR     M       ;INCREMENT IT
DB91 E1         KEY5:   POP     H
                ;
DB92 F1         KEY2:   POP     PSW
DB93 FB                 EI
DB94 C9                 RET
                ;
DB95 CD9BDB     KEY6:   CALL    RSETP   ;RESET POINTERS
DB98 C391DB             JMP     KEY5
                ;
                ; RESET BOTH POINTERS TO START
                ;
DB9B 210000     RSETP:  LXI     H,0
DB9E 2202F4             SHLD    CCNT    ;ZERO BOTH
DBA1 2106F4             LXI     H,BUFF
DBA4 2204F4             SHLD    KPNTR   ;RESET PNTRS
DBA7 2200F4             SHLD    CPNTR
DBAA C9                 RET
                        ENDIF           ;INTERRUPTS
                ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
                ;
                ; CHECK FOR CONSOLE INPUT READY
                ;
DBAB 3A0300     CONST:  LDA     IOBYTE
DBAE E602               ANI     2
DBB0 C2CBDB             JNZ     LISST   ;LIST
                ;
                ;**********************************
                        IF      INTRM
                ;
                ; CHECK INPUT BUFFER RATHER THAN KEYBOARD
                ;
DBB3 E5                 PUSH    H
DBB4 2A02F4             LHLD    CCNT    ;BOTH COUNTS
DBB7 7C                 MOV     A,H
DBB8 95                 SUB     L       ;DIFFERENCE
DBB9 E1                 POP     H
DBBA C8                 RZ              ;NO INPUT
                ;
DBBB E5                 PUSH    H
DBBC 2A00F4             LHLD    CPNTR   ;COMPUTER PNTR
DBBF 7E                 MOV     A,M     ;NEXT CHAR
DBC0 E1                 POP     H
DBC1 FE03               CPI     CTRC    ;^C?
DBC3 CAC8DB             JZ      QUIT    ;YES, QUIT
                ;
                ; MAKE CP/M THINK THERE IS NO INPUT
                ; SO SCROLLING WON'T BE ABORTED
                ;
```

```
DBC6 AF                 XRA     A           ;GET ZERO
DBC7 C9                 RET
                ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
                ;
                        ELSE                ;NOT INTERRUPTS
                        IN      CSTAT       ;GET STATUS
                        ANI     CIMSK
                        RZ                  ;NOT READY
                        ENDIF               ;INTERRUPTS
                ;
DBC8 3EFF       QUIT:   MVI     A,TRUE
DBCA C9                 RET                 ;INPUT READY
                ;
                ; LIST READY FOR CONSOLE
                ;
DBCB DB12       LISST:  IN      LSTAT
DBCD E601               ANI     LIMSK
DBCF C8                 RZ                  ;NOT READY
DBD0 3EFF               MVI     A,TRUE
DBD2 C9                 RET                 ;READY
                ;
                ; CONSOLE INPUT
                ;
DBD3 3A0300     CONIN:  LDA     IOBYTE
DBD6 E602               ANI     2
DBD8 C206DC             JNZ     LIN         ;LIST INPUT
                ;
                ;**********************************
                        IF      INTRM       ;INTERRUPTS
                ;
                ; GET INPUT FROM KEYBOARD BUFFER
                ; INSTEAD OF FROM CONSOLE
                ;
DBDB E5                 PUSH    H
DBDC 2A02F4     CIN3:   LHLD    CCNT        ;BOTH COUNTS
DBDF 7C                 MOV     A,H
DBE0 95                 SUB     L           ;SAME?
DBE1 CADCDB             JZ      CIN3        ;KEEP TRYING
DBE4 F3                 DI                  ;HOLD OFF
DBE5 2102F4             LXI     H,CCNT      ;COMPUTER COUNT
DBE8 34                 INR     M           ;INCREMENT IT
DBE9 2A00F4             LHLD    CPNTR       ;COMPUTER PNTR
DBEC 7E                 MOV     A,M         ;GET BYTE
                ;
                ; RESET BOTH POINTERS IF CARR RET FOUND
                ;
DBED 23                 INX     H           ;BUMP POINTER
DBEE 2200F4             SHLD    CPNTR       ;SAVE IT
DBF1 FE0D               CPI     CR          ;CARRIAGE RET?
DBF3 C203DC             JNZ     CIN4        ;NO
DBF6 2A02F4             LHLD    CCNT        ;GET BOTH COUNTS
DBF9 7C                 MOV     A,H
DBFA 95                 SUB     L           ;DIFFERENCE
DBFB C201DC             JNZ     CIN5        ;NOT SAME
                ;
                ; RESET BOTH POINTERS TO ZERO
                ;
```

```
DBFE CD9BDB             CALL    RSETP
DC01 3E0D       CIN5:   MVI     A,CR      ;RESTORE CR
DC03 E1         CIN4:   POP     H
DC04 FB                 EI                ;READY FOR MORE
DC05 C9                 RET
                ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
                        ELSE              ;NO INTERRUPTS
                ;
                CIN2:   IN      CSTAT     ;CHECK STATUS
                        ANI     CIMSK
                        JZ      CIN2
                        IN      CDATA     ;GET DATA
                        ANI     7FH       ;MASK PARITY
                        RET
                        ENDIF             ;INTRM
                ;
                ; CONSOLE INPUT FROM LIST
                ;
DC06 DB12       LIN:    IN      LSTAT
DC08 E601               ANI     LIMSK
DC0A CA06DC             JZ      LIN
DC0D DB13               IN      LDATA
DC0F E67F               ANI     7FH
DC11 C9                 RET
                ;
                ; CONSOLE OUTPUT
                ;
DC12 3A0300     CONOUT: LDA     IOBYTE    ;WHERE?
DC15 E603               ANI     3
DC17 B7                 ORA     A
DC18 E22EDC             JPO     LIST
                ;
DC1B DB10       CONW:   IN      CSTAT     ;CHECK STATUS
DC1D E602               ANI     COMSK
DC1F CA1BDC             JZ      CONW
DC22 79                 MOV     A,C       ;GET BYTE
DC23 D311               OUT     CDATA     ;SEND IT
DC25 C9                 RET
                ;
                ; LIST OUTPUT
                ;
DC26 3A0300     LOUT:   LDA     IOBYTE
DC29 E640               ANI     40H       ;BIT 6
DC2B C21BDC             JNZ     CONW      ;CONSOLE OUT
                ;
DC2E DB12       LIST:   IN      LSTAT     ;CHECK STATUS
DC30 E602               ANI     LOMSK
DC32 CA2EDC             JZ      LIST
DC35 79                 MOV     A,C       ;GET BYTE
DC36 D313               OUT     LDATA     ;SEND IT
                ;
                        IF      LNULL > 0
                ;
                ; NULLS FOR LIST DEVICE
                ;
```

```
                        ANI     7FH
                        CPI     CR
                        JNZ     FORM
                        MVI     C,0
                        CALL    LIST    ;1 NULL
                        CALL    LIST    ;2 NULLS
                        CALL    LIST    ;3 NULLS
                        JMP     LIST    ;4 NULLS
                        ENDIF
                ;
DC38 FEOC       FORM:   CPI     FFEED   ;FORMFEED?
DC3A CO                 RNZ             ;NO
                ;
                ; EMULATE FORMFEED WITH 9 LINES
                ;
DC3B C5                 PUSH    B
DC3C 010A09             LXI     B,900H+LF
DC3F CD2EDC     LSKIP:  CALL    LIST
DC42 05                 DCR     B
DC43 C23FDC             JNZ     LSKIP
DC46 C1                 POP     B
DC47 C9                 RET
                ;
                ; PUNCH OUTPUT SENT TO MODEM
                ;
DC48 79         PUNCH:  MOV     A,C     ;GET BYTE
DC49 E67F               ANI     7FH
DC4B B7                 ORA     A       ;NULL?
DC4C C8                 RZ              ;DON'T SEND
DC4D FEOA               CPI     LF
DC4F C8                 RZ              ;SKIP LINEFEED
DC50 CD69DC             CALL    MOUT    ;SEND
DC53 FEOD               CPI     CR
DC55 CA5EDC             JZ      MODCR   ;WAIT FOR CR
DC58 CD74DC             CALL    MIN     ;MODEM INPUT
DC5B D311               OUT     CDATA   ;SEND TO CONSOLE
DC5D C9                 RET
                ;
                ; SEND <CR> TO MODEM, WAIT FOR ONE BACK
                ;
DC5E CD74DC     MODCR:  CALL    MIN
DC61 D311               OUT     CDATA   ;TO CONSOLE
DC63 FEOD               CPI     CR
DC65 C25EDC             JNZ     MODCR   ;KEEP TRYING
DC68 C9                 RET
                ;
                ; MODEM OUTPUT
                ;
DC69 DB14       MOUT:   IN      MSTAT   ;CHECK STATUS
DC6B E680               ANI     MOMSK
DC6D CA69DC             JZ      MOUT
DC70 79                 MOV     A,C     ;GET BYTE
DC71 D315               OUT     MDATA   ;SEND IT
DC73 C9                 RET
                ;
                ; MODEM INPUT
                ;
```

```
DC74 DB14        MIN:     IN       MSTAT      ;CHECK STATUS
DC76 E640                 ANI      MIMSK
DC78 CA74DC               JZ       MIN
DC7B DB15                 IN       MDATA      ;GET BYTE
DC7D E67F                 ANI      7FH        ;MASK PARITY
DC7F C9                   RET
                 ;
DC80 31322D               DB       '1-28-80'  ;VERSION
                 ;
DC88                      END
```

```
symbol table

00C5 ACONT     00C4 ADATA     00C7 BCONT     00C6 BDATA
D600 BIOS      F400 BUFFER    F406 BUFF      F402 CCNT
0011 CDATA     0001 CIMSK     DBDC CIN3      DC03 CIN4
DC01 CIN5      0002 COMSK     DBD3 CONIN     DC12 CONOUT
DBAB CONST     DC1B CONW      F400 CPNTR     000D CR
0010 CSTAT     0003 CTRC      0004 CTRD      0011 CTRQ
0013 CTRS      FFFF DOUBLE    0000 FALSE     000C FFEED
DC38 FORM      DB15 INIT      FFFF INTRM     0003 IOBYTE
F403 KCNT      DB92 KEY2      DB7F KEY3      DB6C KEY4
DB91 KEY5      DB95 KEY6      DB5B KEYBD     F404 KPNTR
0013 LDATA     0005 LEV       000A LF        0001 LIMSK
DC06 LIN       DBCB LISST     DC2E LIST      0000 LNULL
0002 LOMSK     DC26 LOUT      DC3F LSKIP     0012 LSTAT
0015 MDATA     0040 MIMSK     DC74 MIN       DC5E MODCR
0080 MOMSK     DC69 MOUT      0036 MSIZE     0014 MSTAT
4900 OFFSET    DC48 PUNCH     DBC8 QUIT      DB9B RSETP
DB00 START     0095 STOP      FFFF TRUE      DB00 USER
```

entry is read with an IN instruction. The byte is then placed into the keyboard buffer and the buffer pointer and buffer count are both incremented. The interrupt flip-flop is enabled with an EI instruction, then the computer returns to its previous task.

When the CPU needs another byte, it gets it from the keyboard buffer in memory, rather than from the keyboard itself. The instructions starting at subroutine CONIN perform this step. The separate buffer pointer and buffer count, maintained for the CPU, are both incremented.

The interrupt-driven keyboard can be utilized with most of the CP/M systems programs. For example, if a BASIC interpreter has been loaded and a source program has been entered, then the source program can first be listed, then executed by typing the following two lines.

```
LIST
RUN
```

The second command can be given immediately following the first, even though the first task has not been completed. The second command will not be displayed on the console, however, until the completion of the first task. Therefore, the operator must type carefully.

SCROLL CONTROL AND TASK ABORTION

Data can appear (scroll) too rapidly on a high-speed video screen. With the usual CP/M arrangement, the user can type a Control-S to freeze the video display. Typing any other character will cause scrolling to resume. The interrupt-driven routine given in Listing 4.1 incorporates its own scroll control. Typing a Control-S freezes the screen, just as with the usual CP/M setup. However, scrolling can only be resumed by typing a Control-Q. The two commands, Control-S and Control-Q, are treated distinctly; they are not placed into the input buffer, but are acted upon immediately.

CP/M tasks are normally aborted by typing any keyboard character. On the other hand, a Control-C is required in Microsoft BASIC, and a Control-E is used by Xitan BASIC for aborting the current task. This protocol has been altered so that characters can be entered into the keyboard buffer during a scroll operation. Nevertheless, it may be desirable to abort a task.

If no characters have been typed ahead, that is, if the computer is executing the latest command, then a Control-C command will abort the current operation. Alternatively, if there are characters waiting in the console-input buffer, then these must be flushed out by typing a Control-D. At this point, a Control-C can be typed to abort the task. This arrangement will work with most programs, including Microsoft BASIC and Tarbell BASIC. If you use Xitan BASIC, then you must change the abort command character in the interface routine from a Control-C to a Control-E.

An additional alteration is necessary for the Word-Master text editor. First of all, Word-Master buffers the keyboard buffer using software routines. Consequently, a hardware-interrupt system is unnecessary. Secondly, Word-Master uses Control-C and Control-D for system commands. Control-C is used to display the next screen and Control-D is used to move the cursor to the next word. If you want to use hardware interrupts with Word-Master, you must change the Control-C and Control-D commands in either the interface routine or in Word-Master.

DATA TRANSMISSION BY TELEPHONE

The process of transmitting information between a peripheral and the computer may be simple or it may be complex. If the system console is wired into the computer, or if the computer itself is built into the console, then the integrity of the transmitted data is not likely to be much of a problem. It may be, however, that the console is connected to the computer through a telephone line. The computer may be located across town or across the country. In any case, connection through a telephone line complicates things.

In a typical telephone arrangement, the data is sent from the console by modulating an acoustical carrier for transmission over the telephone line. The conversion is performed by an electronic device called a *modem* (the name is an abbreviation for MODulator-DEModulator). Two modems are required, one at each end of the telephone line. One converts the transmitted signal to telephone frequencies, the other converts the signal back to the original data.

The modem may also have an *acoustical coupler*. This allows a standard telephone headset to be pressed into two rubber-lined openings in the modem, making a direct connection between the modem and the telephone line unnecessary.

There will usually be two data carrier signals at different frequencies. This allows simultaneous two-way, or *full duplex*, operation. The computer can transmit data to the console on one carrier while the console is transmitting data to the computer on the other carrier.

A microcomputer can produce a more effective link between a console and a large main-frame computer, especially if a relatively slow modem is utilized. A program can be developed using the microcomputer's editor, then the resulting file can be automatically transmitted to the larger computer. A subroutine that can be used to link a microcomputer to a large computer is given in Listing 4.2. This routine can be readily incorporated into the system monitor introduced in Chapter 6.

Listing 4.2 Connection to a large computer

```
                        ;
                        ; CONNECT TO ANOTHER COMPUTER
                        ; THROUGH PHONE MODEM
                        ; (Z-80 CODE)
                        ;
0014                    DSTAT   EQU     14H       ;STATUS
0015                    DDATA   EQU     DSTAT+1
0040                    DIMSK   EQU     40H       ;INPUT MASK
0080                    DOMSK   EQU     80H       ;OUT MASK
                        ;
0004                    CTRD    EQU     4         ;^D, COPY
57A1                    TYFLG   EQU     STACK+1   ;COPY FLAG
                        ;
5BB3 AF                 DEC:    XOR     A         ;ZERO
5BB4 32 57A1                    LD      (TYFLG),A ;RESET COPY
5BB7 DB 14              DECIN:  IN      A,(DSTAT) ;READY?
5BB9 E6 40                      AND     DIMSK
5BBB 28 13                      JR      Z,ALTIN   ;NO
5BBD 3A 57A1                    LD      A,(TYFLG) ;COPY FLAG
5BC0 B7                         OR      A         ;TO MEMORY?
5BC1 28 07                      JR      Z,DIN5    ;NO
5BC3 CD 5BF1                    CALL    DINPUT    ;GET BYTE
5BC6 77                         LD      (HL),A    ;TO MEMORY
5BC7 23                         INC     HL        ;POINTER
5BC8 18 03                      JR      DEC2
                        ;
5BCA CD 5BF1            DIN5:   CALL    DINPUT    ;GET BYTE
5BCD CD 5835            DEC2:   CALL    OUTT      ;TO CONSOLE
5BD0 CD 5827            ALTIN:  CALL    INSTAT    ;CONSOLE
5BD3 28 E2                      JR      Z,DECIN   ;NOT READY
5BD5 CD 581A                    CALL    INPUT2    ;CONSOLE
5BD8 FE 04              ALT2:   CP      CTRD      ;^D
5BDA 28 1A                      JR      Z,DCOPY   ;SET FLAG
```

```
5BDC CD 5BE1    ALT5:   CALL    DECOUT  ;TO DEC
5BDF 18 D6              JR      DECIN   ;DEC INPUT
                ;
                ; OUTPUT A BYTE TO DEC
                ;
5BE1 F5         DECOUT: PUSH    AF
5BE2 CD 5BEA            CALL    DORDY
5BE5 F1                 POP     AF
5BE6 D3 15              OUT     (DDATA),A
5BE8 18 CD              JR      DECIN   ;NEXT
                ;
                ; DEC INPUT READY
                ;
5BEA DB 14      DORDY:  IN      A,(DSTAT)
5BEC E6 80              AND     DOMSK
5BEE 28 F1              JR      Z,DECOUT
5BF0 C9                 RET
                ;
                ; INPUT FROM DEC MODEM
                ;
5BF1 DB 15      DINPUT: IN      A,(DDATA)
5BF3 E6 7F              AND     DEL     ;MASK PARITY
5BF5 C9                 RET
                ;
                ; SET DEC COPY FLAG. START COPYING
                ; INTO MEMORY AT 100 HEX
                ;
5BF6 21 0100    DCOPY:  LD      HL,100H
5BF9 3E 01              LD      A,1
5BFB 32 57A1            LD      (TYFLG),A
5BFE 18 B7              JR      DECIN
                ;
                        END     START
```

PARITY CHECKING

Parity checking provides a method of monitoring the integrity of data transmission. While there are several different schemes for digitally encoding the common characters, the ASCII method is frequently used for microcomputers. The ASCII code, shown in Appendix A, requires only seven bits for each character. Since each byte of data contains eight bits, there is one bit available for use as a check bit.

Consider the 7-bit pattern for the ASCII characters 2 and 3.

ASCII 2	011 0010
ASCII 3	011 0011

The value of 2 is encoded with four logical zero bits and three logical 1 bits. The value of 3 is encoded with three logical zero bits and four logical 1 bits. A parity check can be obtained by including an additional bit on the left (high-order) end.

There are two common methods of generating the parity bit. One encoding method is called *even parity.* In this case, a leading zero bit is added if there are an even number of logical ones among the other seven bits. On the other hand, the parity bit would be a logical 1 if there are an odd number of logical ones among the other seven bits. With even parity coding, the ASCII characters 2 and 3 would look like this.

```
2    1011 0010
3    0011 0011
```

Now the 8-bit representation of both the 2 and the 3 contains an even number of logical ones (and an even number of logical zeros).

An alternate approach is called *odd parity.* In this case, the operation is simply the inverse of even parity. The logic of the parity bit is chosen so that the resulting bit pattern contains an odd number of logical ones. Either even or odd parity encoding will provide a check on the integrity of the data transmission.

Suppose that during transmission of the character 2, the rightmost bit became inverted. The console sent the even-parity bit pattern

```
1011 0010
```

but the computer received the bit pattern

```
1011 0011
```

A parity check, performed at the computer, would be able to detect the fact that there was an error.

A typical console-input routine might look like this.

```
CONIN:  IN      CMASK   ;CHECK STATUS
        ANI     CIMSK   ;MASK FOR INPUT
        JZ      CONIN   ;LOOP UNTIL READY
        IN      CDATA   ;GET THE DATA
        ANI     7FH     ;REMOVE PARITY
        RET
```

The next to the last instruction in this subroutine performs a logical AND with 7F hex. This step is used to remove the high-order bit of the byte since it is not needed for ASCII data. Instead of ignoring this eighth bit, we could use it as a parity check. An input routine to perform a check for parity looks like the following list.

```
CONIN:  IN      CMASK   ;CHECK STATUS
        ANI     CIMSK   ;MASK FOR INPUT
        JZ      CONIN   ;LOOP UNTIL READY
        IN      CDATA   ;GET THE DATA
        ORA     A       ;SET PARITY FLAG
        JPO     PERROR  ;PARITY ERROR
        ANI     7FH     ;REMOVE PARITY
        RET
;
; PARITY-ERROR MESSAGE
;
PERROR: . . .
```

This routine is essentially the same as the one given immediately before, but after the data register has been read by the computer, the parity of the byte is determined.

The ORA A instruction performs a logical OR of the accumulator with itself. A logical OR of any byte with itself will not change the byte. However, it does affect the status flags in this case. After the OR operation, the parity flag will be set according to the parity of the accumulator. If the parity is found to be odd, then an error is present. The JPO instruction causes a jump to the parity-error routine in this case. However, if the parity is found to be even, then the byte in the accumulator does not contain a parity error.

Notice that a parity check will not detect an even number of bit errors in a byte. There may be two, four, or six errors, and the parity check will not detect an error. This is not likely to be a practical problem, however, since the likelihood of two errors is much less than the likelihood of single errors.

ASCII computer terminals usually have the ability to automatically transmit an eighth parity bit with the data. Furthermore, there will typically be a user-selectable switch for choosing either even or odd parity. There may also be the additional choices of always resetting or always setting the parity bit. The input routine can check for odd parity if the JPO instruction is changed to a JPE instruction.

There are much more sophisticated methods of checking for transmission errors. One of these is the checksum approach discussed in Chapter 9. With this method, the transmitted data are added together. At regular intervals, the sum, or its complement, is transmitted along with the data. When the data are decoded, the data are added up again and compared to the checksum.

The Hamming error-correction code is even better than the checksum method. It not only detects errors, but can also correct them. In the end, however, it is wise to find out why errors occur, and to take the appropriate action to correct the problem. A dirty tape head, for example, can produce errors. Cleaning the head is better than relying on an error-correction scheme.

CHAPTER FIVE

Macros

Sophisticated assemblers incorporate a macro processor. A macro is used to define a set of instructions which are associated with the macro name. Then whenever the macro name appears in the source program, the assembler substitutes the corresponding instructions. This is called a *macro expansion.*

Suppose that we want to interchange the contents of two memory locations with the following instructions.

```
LDA       FIRST     ;GET FIRST BYTE
PUSH      PSW       ;SAVE
LDA       SECOND    ;GET SECOND
STA       FIRST     ;PUT INTO FIRST
POP       PSW       ;GET FIRST
STA       SECOND    ;PUT INTO SECOND
```

This set of instructions can be defined in a macro called SWAP.

```
SWAP      MACRO               ;SWAP FIRST AND SECOND
          LDA       FIRST     ;GET FIRST BYTE
          PUSH      PSW       ;SAVE
          LDA       SECOND    ;GET SECOND
          STA       FIRST     ;PUT INTO FIRST
          POP       PSW       ;GET FIRST
          STA       SECOND    ;PUT INTO SECOND
          ENDM
```

The macro definition is placed near the top of the assembler source program. The first line defines the macro name; the last line terminates the definition. The name SWAP can now be used like an operation code. it is placed in the source program whenever the corresponding instructions are needed. When the assembler encounters the name SWAP, it substitutes the desired instructions. The final binary code generated by the assembler is the same as it would be if the instructions had originally been entered into the source program.

Each time the macro name SWAP appears in the source program, the same set of instructions will be generated and the same two memory locations will be interchanged. The SWAP macro becomes more versatile if the memory locations can be changed. If the names of the memory locations are placed on the first line of the macro definition, they become dummy variables.

```
SWAP    MACRO   FIRST, SECOND
        LDA     FIRST   ;GET 1ST BYTE
        PUSH    PSW     ;SAVE
        LDA     SECOND  ;GET 2ND
        STA     FIRST   ;PUT INTO 1ST
        POP     PSW     ;GET 1ST
        STA     SECOND  ;PUT INTO 2ND
        ENDM
```

The actual parameters in the macro call are substituted for the dummy parameters at assembly time. The macro call

```
SWAP    HIGH, LOW
```

generates the assembly language instructions

```
LDA     HIGH    ;GET 1ST BYTE
PUSH    PSW     ;SAVE
LDA     LOW     ;GET 2ND
STA     HIGH    ;PUT INTO 1ST
POP     PSW     ;GET 1ST
STA     LOW     ;PUT INTO 2ND
```

The statement

```
SWAP    LEFT, RIGHT
```

will produce the instructions

```
LDA     LEFT    ;GET 1ST BYTE
PUSH    PSW     ;SAVE
LDA     RIGHT   ;GET 2ND
STA     LEFT    ;PUT INTO 1ST
POP     PSW     ;GET 1ST
STA     RIGHT   ;PUT INTO 2ND
```

The structure of macros can be much more complicated than the above examples. One macro can be nested inside another.

```
OUTER   MACRO
        . . .
        IF        FAST
INNER   MACRO
        . . .
        . . .
        ENDM                ;;INNER
        . . .
        ENDIF               ;;FAST
        . . .
        ENDM                ;;OUTER
```

Conditional assembly directives can be used to create different versions. Comments in the macro definition which begin with a single semicolon are reproduced in the macro expansion along with the op codes. But if the comments are preceded by two consecutive semicolons, then they will appear only in the macro definition, not in the macro expansion.

GENERATING THREE OUTPUT ROUTINES WITH ONE MACRO

A subroutine can be used whenever a set of instructions is needed at several places in a program. But there are times when a similar but different group of instructions is needed. A subroutine cannot be used in this case. Consider the three 8080 output routines that follow. The first sends a byte to the console, the second sends a byte to the list device, and the third sends a byte to the phone modem.

```
COT:    IN      CSTAT
        ANI     COMSK
        JZ      COT
        MOV     A,C
        OUT     CDATA
        RET
;
LOT:    IN      LSTAT
        ANI     LOMSK
        JZ      LOT
        MOV     A,C
        OUT     LDATA
        RET
;
MOT:    IN      MSTAT
        ANI     MOMSK
        JNZ     MOT
        MOV     A,C
        OUT     MDATA
        RET
```

The structure of these three routines is very similar. Each begins by reading the appropriate status register. Then a logical AND is performed to select the output-ready bit. Looping occurs until the peripheral is ready. The byte is moved from the C register into the accumulator and sent to the appropriate peripheral. Finally, a return instruction is executed.

These three routines are slightly different, hence they cannot be replaced by a single subroutine. However, since they have similar structure they can be generated with a macro. The macro definition looks like this.

```
OUTPUT  MACRO   ?S,?Z    ;OUTPUT ROUTINES
?S&OT:  IN      ?S&STAT  ;CHECK STATUS
        ANI     ?S&OMSK  ;MASK FOR OUTPUT
        J&?Z    ?S&OT    ;NOT READY
        MOV     A,C      ;GET BYTE
        OUT     ?S&DATA  ;SEND IT
        RET
        ENDM
```

It would appear near the beginning of the source program. The macro name chosen is OUTPUT and the two dummy arguments are ?S and ?Z. Dummy arguments can have the same form as any other identifier. A question mark was chosen as the first character so that the dummy arguments would be easier to find in the macro definition. You must be careful not to use register names such as A, B, H, or L for dummy arguments if these register names also appear in the macro.

Each of the three output routines is generated by a one-line macro call.

```
        OUTPUT  C,Z      ;CONSOLE OUTPUT
;
        OUTPUT  L,Z      ;LIST OUTPUT
;
        OUTPUT  M,NZ     ;MODEM OUTPUT
```

Each line includes the appropriate parameters. At assembly time, the real arguments replace the dummy arguments of the macro. The ampersand character (&) is a concatenation operator. It separates a dummy argument from additional text. The macro processor substitutes the real parameter for the dummy argument, then joins it to the rest of the text. By this means the expression ?S&OT becomes LOT if the real argument is the letter L.

Macro assemblers may give the user three options for the assembly listing:

1. Show the macro call, the generated source line, and the resultant hex code.
2. Show the macro call and the hex code.
3. Show only the macro call.

If option 1 is chosen, then the above three macro calls to OUTPUT will produce the following.

```
                OUTPUT  MACRO   ?S,?Z     ;OUTPUT ROUTINES
                ?S&OT:  IN      ?S&STAT   ;CHECK STATUS
                        ANI     ?S&OMSK   ;MASK FOR OUTPUT
                        J&?Z    ?S&OT     ;NOT READY
                        MOV     A,C       ;GET BYTE
                        OUT     ?S&DATA   ;SEND IT
                        RET
                        ENDM
                ;
                        OUTPUT  C,Z       ;CONSOLE OUTPUT
4000+DB10       COT:    IN      CSTAT     ;CHECK STATUS
4002+E602               ANI     COMSK     ;MASK FOR OUTPUT
4004+CA0040             JZ      COT       ;NOT READY
4007+79                 MOV     A,C       ;GET BYTE
4008+D311               OUT     CDATA     ;SEND IT
400A+C9                 RET
                ;
                        OUTPUT  L,Z       ;LIST OUTPUT
400B+DB12       LOT:    IN      LSTAT     ;CHECK STATUS
400D+E602               ANI     LOMSK     ;MASK FOR OUTPUT
400F+CA0B40             JZ      LOT       ;NOT READY
4012+79                 MOV     A,C       ;GET BYTE
4013+D313               OUT     LDATA     ;SEND IT
4015+C9                 RET
                ;
                        OUTPUT  M,NZ      ;MODEM OUTPUT
4016+DB14       MOT:    IN      MSTAT     ;CHECK STATUS
4018+E680               ANI     MOMSK     ;MASK FOR OUTPUT
401A+C21640             JNZ     MOT       ;NOT READY
401D+79                 MOV     A,C       ;GET BYTE
401E+D315               OUT     MDATA     ;SEND IT
4020+C9                 RET
```

The first argument in the macro, ?S, is replaced by the actual argument. This is the letter C in the first call, the letter L in the second call, and the letter M in the third call. The second argument is used to select a JZ or JNZ instruction for the third line of the macro expansion.

Some assemblers automatically remove the ampersand symbol from the resultant assembly listing. Others leave the symbol in place. In this latter case, the first line of the first routine would look like this.

```
C&OT:   IN      C&STAT   ;CHECK STATUS
```

But this is a matter of style. The actual machine code generated is the same in either case.

GENERATING Z-80 INSTRUCTIONS WITH AN 8080 ASSEMBLER

If you have a Z-80 CPU but an 8080 macro assembler, such as the Digital Research MAC, you can run all of the 8080 programs just as they are given in this book. You can also do the Z-80 programs by using macros to generate the Z-80 instructions. For some of the instructions, the regular Zilog mnemonic can be used. For other instructions a slightly different format is necessary. Consider, for example, the Z-80 instruction that performs a two's complement on the accumulator. The Zilog mnemonic for this operation is NEG. A Z-80 assembler converts this mnemonic into the two hex bytes ED 44. With an 8080 macro assembler you can use the same mnemonic. Define the macro

```
NEG     MACRO   ; TWO'S COMPLEMENT
        DB      0EDH, 44H
        ENDM
```

Then, the macro call

```
NEG
```

is placed in the source program when the Z-80 NEG instruction is needed. The 8080 macro assembler will insert the desired hex bytes ED 44 at this point.

As another example, consider the Z-80 relative-jump instruction. This instruction can be implemented with a macro that uses the assembler's program counter, a dollar sign. The macro definition looks like this.

```
JR      ADDR    ;RELATIVE JUMP
        DB      18H, ADDR-$-1
        ENDM
```

The dummy parameter ADDR is the destination address of the jump. The macro call

```
JR      ERROR
```

will generate the correct Z-80 code. The first byte will be 18 hex. The second byte will be the required displacement for the jump.

The Z-80 instruction, DJNZ, can be generated in a similar way. This instruction decrements the B register and jumps relative to the address of the argument if the zero flag is not set. The macro definition is

```
DJNZ    MACRO   ADDR
        DB      10H,ADDR-$-1
        ENDM
```

and the macro call looks like

```
DJNZ    LOOP
```

This approach will work with most macro assemblers. There may be a problem, however, with the interpretation of the dollar sign. This symbol usually refers to the address of the beginning of the current instruction. But for some assemblers, it is interpreted as the address of the following instruction.

If your assembler uses the latter interpretation, you will have to change the macro accordingly. If in doubt, check the user manual.

Some Z-80 mnemonics are not compatible with the macro format. For example, the Z-80 instruction

```
PUSH     IX
```

cannot be generated with a macro called

```
PUSH     MACRO    REG
```

since PUSH is a regular 8080 mnemonic. One possibility is to name the macro PUSHIX instead.

```
PUSHIX   MACRO
         DB       0DDH,0E5H
         ENDM
```

Similar problems occur with the commands POP IX, ADD IX,BC, SUB (IX+dis), and SET. A format that is different from the Z-80 mnemonic must be chosen in each of these cases.

The Digital Research macro assembler has an added bonus. Frequently-used macros can be placed into a separate macro library and given the file extension of LIB. In fact, this assembler is supplied with a macro library called Z80.LIB that will generate all of the Z-80 instructions. The statement

```
MACLIB   Z80
```

is placed near the beginning of the regular source program. The assembler will then look in the file Z80.LIB for the required macros.

EMULATING Z-80 INSTRUCTIONS WITH AN 8080 CPU

The Z-80 CPU can execute many powerful instructions that are not available to the 8080. Some of these useful instructions are difficult to implement on an 8080, while others are simply combinations of regular 8080 instructions. The NEG instruction is one of the easiest to implement. The macro definition is

```
NEG      MACRO              ;8080 TWO'S COMPLEMENT
         CMA                ;;1'S COMPLEMENT
         INR      A         ;;2'S COMPLEMENT
         ENDM
```

Now, whenever a two's complement is needed, the macro call

```
NEG
```

is placed into the program. The assembler generates the required 8080 mnemonics

```
        CMA
        INR     A
```

Another useful Z-80 operation is the arithmetic shift. This operation shifts all bits of a register one position to the left. The high-order bit is moved into carry, that is, the carry flag is set to a 1 if bit 7 was originally a value of 1. The carry flag is reset to zero if bit 7 was zero. A value of zero is placed into the low-order bit (bit zero).

The 8080 instruction ADD A, which adds the value in the accumulator to itself, performs the arithmetic shift left. But for the 8080, this is the only register which can perform the shift. The Z-80 has an additional instruction which allows this operation to be performed on any of the general-purpose, 8-bit registers or on the memory byte referenced by HL, IX, or IY.

The following macro will generate a set of 8080 instructions for the arithmetic shift left operation.

```
SLA     MACRO   REG     ;SHIFT LEFT ARITH
        MOV     A,REG   ;;GET BYTE
        ADD     A       ;;SHIFT LEFT
        MOV     REG,A   ;;PUT BACK
        ENDM
```

The byte is first moved to the accumulator. The next step is to add the accumulator to itself. This doubling operation performs the needed shift into carry. Then the result is returned to the original register. The value in register C can be doubled by inserting the macro call

```
        SLA     C
```

This macro must be used with caution, since the accumulator will be changed during use. But the byte originally in the accumulator cannot be saved with a PUSH PSW instruction. The problem is that the subsequent POP PSW command will overlay the flag register, so that the carry result of the shift will be lost. One solution is to save the accumulator in memory.

```
SLA     MACRO   REG     ;SHIFT LEFT ARITH
        STA     SAVE    ;;SAVE A
        MOV     A,REG   ;;GET BYTE
        ADD     A       ;;SHIFT LEFT
        MOV     REG,A   ;;PUT BACK
        LDA     SAVE    ;;RESTORE A
        ENDM
```

THE REPEAT MACROS

There are times when several lines of identical or nearly identical lines of code are needed. Three repeat macros, REPT, IRP, and IRPC are provided for this purpose. The repeat macros differ from the regular macros in that they are placed directly into the source program where they are needed. The macro definition is the macro call. In the simplest form, an instruction or group of instructions can be replicated. The expression

```
REPT    4
RAR
ENDM
```

will generate the four lines

```
RAR
RAR
RAR
RAR
```

By using the SET directive, this operation can become more versatile. The SET instruction is like an EQU except that the value can be redefined. The lines

```
ADDR    SET     8000H
        REPT    4
ADDR    SET     ADDR+3
        DW      ADDR
        ENDM
```

will generate the code corresponding to

```
DW      8003H
DW      8006H
DW      8009H
DW      800CH
```

Such a series could refer to jump vectors that are spaced three bytes apart.

The repeat macro, combined with the conditional-assembly directive, can generate the required number of nulls after a carriage-return, line-feed pair. This will give the printer time to return to the left margin. Some printers need no nulls, whereas others may need as many as six or seven. The source code could be

```
;
; OUTPUT TO LIST DEVICE
;
LOUT:   IN      LSTAT
        ANI     LOMSK
        JZ      LOUT
        MOV     A,C     ;GET DATA
        OUT     LDATA   ;SEND BYTE
```

```
;
        IF      NULLS > 0
        ANI     7FH     ;REMOVE PARITY
        CPI     CR      ;CARRIAGE RETURN?
        RNZ             ;NO
        MVI     C,0     ;GET A NULL
;
        REPT    NULLS   ;HOW MANY?
        CALL    LOUT    ;SEND NULL
        ENDM            ;REPEAT MACRO
        ENDIF           ;NULLS
;
        RET
```

The first part is a typical output subroutine. A call is made with the byte in register C. When the output device is ready, the byte is moved from the C register to the accumulator. It is then sent to the printer. If no nulls are required, then the passage from

```
IF      NULLS > 0
```

to

```
ENDIF
```

is not assembled. On the other hand, if nulls are required, then this passage is assembled. If four nulls are needed, then the assembler will generate four lines of

```
CALL   LOUT       ;SEND NULL
```

The list output routine calls itself to produce the required nulls. The identifier called NULLS must be previously set to the necessary number of nulls.

There are two other repeat macros called IRP and IRPC. A set of one-character message routines can be generated by using the indefinite repeat macro IRPC. This example will introduce something called a programming trick. Some people think that it is a horrible example of programming. Others think it is very clever. Its purpose is to save two bytes of instruction each time it is used. In addition, less branching is required.

Suppose that we need five different message routines that each produce a single character. The instructions might look like

```
CHARC:  MVI     A,'C'
        JMP     OUTT
CHARM:  MVI     A,'M'
        JMP     OUTT
CHARR:  MVI     A,'R'
        JMP     OUTT
CHAR?:  MVI     A,'?'
        JMP     OUTT
CHAR$:  MVI     A,'$'
        JMP     OUTT
        . . .
OUTT:   <output routine>
```

If the B and C registers are not in use, we can shorten the above passage by replacing each line containing the instruction JMP OUTT with a line of DB 1.

```
CHARC:  MVI     A,'C'
        DB      1
CHARM:  MVI     A,'M'
        DB      1
CHARR:  MVI     A,'R'
        DB      1
CHAR?:  MVI     A,'?'
        DB      1
CHAR$:  MVI     A,'$'
;
OUTT:   . . .
```

Let's see how this works. Suppose that a branch is made to the label CHARC. The accumulator is loaded with an ASCII letter C. The next byte, a DB 1, looks like the start of an LXI B instruction. The following two bytes, corresponding to the MVI A,'M' instruction, will be interpreted as the argument for the LXI instruction. That is, they will be considered as data. The same will hold for the other occurrences of DB 1. By this means, we have effectively shortened the code. We no longer need the JMP statements. Caution: a disassembler is not likely to interpret this passage correctly. It looks like there are labels pointing into the middle of the LXI instructions. Notice that the second version has subroutine OUTT positioned directly under the CHAR$ routine, so that no JMP instruction is needed at this point.

The second version can be easily generated with the IRPC macro. Only five lines are needed in the source program.

```
        IRPC    X,CMPR?$
        DB      1          ;FAKE LXI B
CHAR&X: MVI     A,'&X'
        ENDM
```

The five different message routines are all generated with this single macro. One replication is made for each character of the second argument to IRPC.

PRINTING STRINGS WITH MACROS

Suppose that we want to send messages to the console from various points of a program. We could write a subroutine called SENDM for this purpose.

```
SENDM: LDAX    D          ;GET CHAR
       ORA     A          ;ZERO?
       RZ                 ;YES
       INX     D          ;POINTER
       MOV     C,A
       CALL    OUTT       ;SENT
       JMP     SENDM      ;NEXT
```

The address of the message is loaded into the DE register and subroutine SENDM is called.

```
LXI     D,MESS1
CALL    SENDM
```

Subroutine SENDM prints a message by sending each character to the output subroutine OUTT. When a binary zero, used to indicate the end of the message, is found, SENDM returns to the calling program.

We can simplify the sending of messages by using a macro called PRINT. At each point we write

```
PRINT   <CHECKSUM ERROR>
. . .
PRINT   <END OF FILE>
. . .
PRINT   <OUTPUT TO LIST?>
```

The macro called PRINT will generate the message given in the argument. The message is enclosed in angle brackets because the blanks are part of the argument.

If subroutine SENDM were placed into the macro body, then one copy of SENDM would be inserted for each occurrence of the PRINT statement. But we don't need more than one copy of SENDM. On the other hand, if we don't include SENDM in the macro, there may not be any copies at all. What we need is a mechanism for inserting one, and only one, copy of SENDM regardless of how many times we give the PRINT command.

The solution is to write a double macro—one nested inside the other. Both macros will be given the same name. Subroutine SENDM will be part of the outer macro which will be expanded only once. The layout looks like this.

```
PRINT   MACRO   <message>   ;OUTER MACRO
        . . .
        [define SENDM]
        . . .
PRINT   MACRO   <message>   ;INNER MACRO
        . . .
        [send message]
        . . .
        ENDM            ;INNER MACRO
        . . .
        ENDM            ;OUTER MACRO
```

The source program in Listing 5.1 demonstrates this technique. The outer macro PRINT has the argument ?TEXT, used for the first call to the macro. Subroutine SENDM is generated at this time. Additional macro calls to PRINT utilize the inner macro which has the argument ?TEXT2. Subroutine SENDM is not generated on these subsequent calls.

Listing 5.1. Source listing for a macro demonstration program.

```
;
PRINT   MACRO   ?TEXT
        LOCAL   AROUND
;
        JMP     AROUND  ;SENDM
;
; SUBROUTINE TO SEND A STRING TO
; THE CONSOLE. BINARY ZERO AT STRING END.
; D,E IS STRING POINTER.
;
SENDM:  LDAX    D       ;GET CHAR
        ORA     A       ;ZERO?
        RZ              ;YES
        INX     D       ;POINTER
        MOV     C,A
        CALL    OUTT    ;SENT
        JMP     SENDM   ;NEXT
;
AROUND:
;
; REDEFINE THE MACRO
;
PRINT   MACRO   ?TEXT2
        LOCAL   MESG,CONT
;
        PUSH    D       ;SAVE D,E
        LXI     D,MESG  ;POINT
        CALL    SENDM
        POP     D       ;RESTORE
        JMP     CONT    ;SKIP MESSAGE
;
MESG:
        DB      CR,LF,'&?TEXT2',0
;
CONT:   ENDM            ;INNER MACRO
        PRINT   <?TEXT>
        ENDM            ;OUTER MACRO
;
CSTAT   EQU     10H     ;CONSOLE STATUS
CDATA   EQU     CSTAT+1 ;CONSOLE DATA
CR      EQU     13      ;CARRIAGE RETURN
LF      EQU     10      ;LINE FEED
;
ORG     100H
START:
        PRINT   <CHECKSUM ERROR.>
;
        PRINT   <END OF FILE.>
;
        PRINT   <OUTPUT TO LIST?>
        JMP     0       ;RETURN TO CP/M
;
; SEND CHARACTER IN C TO THE CONSOLE
;
```

```
OUTT:   IN      CSTAT
        ANI     2
        JZ      OUTT
        MOV     A,C
        OUT     CDATA
        RET
;
        END
```

Subroutine SENDM is coded into the main flow of the program, that is, it is an inline routine. It is therefore necessary to jump around SENDM. Additionally, there must be a branch around each of the messages, since they too are coded inline. Labels for the required branches are uniquely generated in the macro by declaring the corresponding labels as LOCAL. The resulting assembly listing is given in Listing 5.2. The assembler places plus symbols between the address and the generated code of the assembly listing to designate those lines that were generated by macros. Thus, lines that contain plus symbols were not present in the original source listing.

Listing 5.2. Assembly listing for a macro demonstration program.

```
;
PRINT   MACRO   ?TEXT
        LOCAL   AROUND
;
        JMP     AROUND  ;SENDM
;
; SUBROUTINE TO SEND A STRING TO
; THE CONSOLE. BINARY ZERO AT STRING END.
; D,E IS STRING POINTER.
;
SENDM:  LDAX    D       ;GET CHAR
        ORA     A       ;ZERO?
        RZ              ;YES
        INX     D       ;POINTER
        MOV     C,A
        CALL    OUTT    ;SENT
        JMP     SENDM   ;NEXT
;
AROUND:
;
; REDEFINE THE MACRO
;
PRINT   MACRO   ?TEXT2
        LOCAL   MESG,CONT
;
        PUSH    D       ;SAVE D,E
        LXI     D,MESG  ;POINT
        CALL    SENDM
        POP     D       ;RESTORE
        JMP     CONT    ;SKIP MESSAGE
;
MESG:
        DB      CR,LF,'&?TEXT2',0
;
```

```
                        CONT:   ENDM                    ;INNER MACRO
                                PRINT   <?TEXT>
                                ENDM                    ;OUTER MACRO
                        ;
0010 =                  CSTAT   EQU     10H             ;CONSOLE STATUS
0011 =                  CDATA   EQU     CSTAT+1         ;CONSOLE DATA
000D =                  CR      EQU     13              ;CARRIAGE RETURN
000A =                  LF      EQU     10              ;LINE FEED
                        ;
0100                    ORG     100H
                        START:
                                PRINT   <CHECKSUM ERROR.>
0100+C30E01                     JMP     ??0001  ;SENDM
0103+1A                 SENDM:  LDAX    D               ;GET CHAR
0104+B7                         ORA     A               ;ZERO?
0105+C8                         RZ                      ;YES
0106+13                         INX     D               ;POINTER
0107+4F                         MOV     C,A
0108+CD6501                     CALL    OUTT            ;SENT
010B+C30301                     JMP     SENDM           ;NEXT
010E+D5                         PUSH    D               ;SAVE D,E
010F+111901                     LXI     D,??0002  ;POINT
0112+CD0301                     CALL    SENDM
0115+D1                         POP     D               ;RESTORE
0116+C32B01                     JMP     ??0003  ;SKIP MESSAGE
0119+0D0A434845                 DB      CR,LF,'CHECKSUM ERROR.',0
                        ;
                                PRINT   <END OF FILE.>
012B+D5                         PUSH    D               ;SAVE D,E
012C+113601                     LXI     D,??0004  ;POINT
012F+CD0301                     CALL    SENDM
0132+D1                         POP     D               ;RESTORE
0133+C34501                     JMP     ??0005  ;SKIP MESSAGE
0136+0D0A454E44                 DB      CR,LF,'END OF FILE.',0
                        ;
                                PRINT   <OUTPUT TO LIST?>
0145+D5                         PUSH    D               ;SAVE D,E
0146+115001                     LXI     D,??0006  ;POINT
0149+CD0301                     CALL    SENDM
014C+D1                         POP     D               ;RESTORE
014D+C36201                     JMP     ??0007  ;SKIP MESSAGE
0150+0D0A4F5554                 DB      CR,LF,'OUTPUT TO LIST?',0
0162 C30000                     JMP     0               ;RETURN TO CP/M
                        ;
                        ; SEND CHARACTER IN C TO THE CONSOLE
                        ;
0165 DB10               OUTT:   IN      CSTAT
0167 E602                       ANI     2
0169 CA6501                     JZ      OUTT
016C 79                         MOV     A,C
016D D311                       OUT     CDATA
016F C9                         RET
                        ;
0170                            END
```

If you are familiar with the operation of your assembler, type up the demonstration program and try it out. Branch to the beginning and three messages will appear at the console.

```
Checksum error
End of file
Output to list?
```

Assembler operation will be considered in the next chapter.

By constructing increasingly complicated macros, it is possible to develop some of the structure that is characteristic of higher-level languages such as Pascal. The common loop constructions

```
REPEAT
. . .
. . .
UNTIL   <condition true>
```

and

```
LOOP
. . .
EXITIF  <condition true>
. . .
ENDLOOP
```

can be realized with macros called REPEAT, UNTIL, and so on. The arguments to UNTIL and EXITIF will consist of three terms. The first and third will be numeric values. The middle term will represent a logical operation such as EQUALS or LESS THAN. The spelling of the logical operators in this case will have to be unusual, since the normal spellings

EQ
LT
GE

are already utilized by the macro assembler. Macros for all of the common structures are available commercially. Also, source programs for structured macros of this type may be given in the instruction manual for your macro assembler.

CHAPTER SIX

Development of a System Monitor

The best way to learn assembly language programming is to actually do it. Consequently, in this chapter you will develop a small but very powerful utility program called a *monitor*. There are many useful things that can be done with the monitor. There is a command to examine memory and another to change it. Other commands deal with memory blocks. These allow you to move a block from one location to another. Some of the features will duplicate those found in other programs, but other features, such as a search routine and a memory test routine, will be unique.

You will not program the entire monitor at one time. Instead, you will start with just the bare essentials. You will check the monitor after each major change to ensure that the new features have been added correctly. With this so-called top-down method, any error that develops is likely to be found in the most recently added instructions. As new features are incorporated, the monitor will increase in size until it reaches 1K bytes. This is a size that can be easily programmed into a single ROM. The monitor will then be immediately available as soon as the computer is turned on.

An editor and an assembler are required for the development of the monitor. In addition, a debugger will be helpful if you have problems along the way. Each phase of the development will require the same sequence of steps.

1. Generate an assembly language source file with the editor.
2. Assemble the source program to produce an object file.
3. Compare the hex code from your assembly listing to the listing given in this chapter.
4. Load the object program into memory.
5. Branch to the monitor and try it out.

The assembly listings given in this chapter are written with 8080 mnemonics. You can use an 8080 assembler for these programs whether you have an 8080 or a Z-80 CPU. The resulting code will run on both an 8080 and a Z-80 CPU. If you have only a Z-80 assembler, you will have to change the mnemonics. The cross-reference between the 8080 and Z-80 mnemonics, given in Appendix G, can be used to find the corresponding instructions. Alternately, you can define the 8080 mnemonics as macros.

PROGRAM DEVELOPMENT DETAILS

This section describes the details of program development. Skip to the next section if you are familiar with the operation of your editor and assembler. An editor is needed to create and alter the assembly language source file. If you have CP/M, you will have an editor called ED. Other editors, such as ED-80, EDIT80, and Word-Master, are separately available.

The session begins by giving the name of the editor and the name of the source program. The following discussion assumes that you have CP/M. If you have some other operating system, the approach will be similar, but the details may differ. Put the CP/M system diskette in drive A and a working diskette in drive B if you have more than one drive. Go to drive B with the command

```
A>B:
```

The response will be

```
B>
```

Type the name of the editor followed by the name of the monitor source program. The command line might look like this.

```
B>A:ED  MON1.ASM
```

for the first version. The digit 1 in the filename refers to the version number. The file type is ASM for the Digital Research assemblers ASM and MAC. The file type should be chosen as MAC, however, if the Microsoft assembler is used.

As you type the source program, be careful to include only the instructions and the comments shown in Listing 6.1A. Do not type the resulting hex code that is also given at the beginning of each line. For example, the line that defines the parameter TOP, on the first page of the listing, should be typed as

```
TOP     EQU     24      ;MEMORY TOP, K BYTES
```

rather than as:

```
0018 =  TOP      EQU      24        ;MEMORY TOP, K BYTES
```

Type a Control-I or tab to automatically generate the blank spaces between symbols.

Most of the assembly language symbols have five or fewer characters. This is acceptable to many assemblers. However, if your assembler only allows names to have a maximum of five characters, then several symbols will have to be shortened. The TITLE directive, on the first line, is another potential problem. The CP/M version is shown. The apostrophes should be removed if the Microsoft assembler is utilized. If the TITLE directive is not available on your assembler, place a semicolon at the beginning of this first line to convert it to a comment.

VERSION 1: THE INPUT AND OUTPUT ROUTINES

Refer to Listing 6.1A. This version will contain only the input and output routines. Generate an assembler source file with the system editor. The following variables will have to be tailored to your particular system.

TOP	(top of usable memory, decimal K)
HOME	(where to return when done)
CSTAT	(console input status address)
CDATA	(console input data address)
CSTATO	(console output status address)
CDATAO	(console output data address)
INMSK	(input-ready mask)
OMSK	(output-ready mask)
BACKUP	(console backspace character)

Normally, CSTATO will be the same as CSTAT, and CDATAO will be the same as CDATA. But if your console input address is different from your console output address, then each can be separately defined. Furthermore, the address of CDATA will typically have a value one larger or smaller than that of CSTAT.

Listing 6.1A. The beginning of a system monitor.

```
                TITLE   '8080 system monitor, ver 1'
                ;
                ; (put today's date here)
                ;
0018 =          TOP     EQU     24        ;MEMORY TOP, K BYTES
5800 =          ORGIN   EQU     (TOP-2)*1024  ;PROGRAM START
                ;
5800            ORG     ORGIN
                ;
                ;
0000 =          HOME    EQU     0         ;ABORT (VER 1-2)
                ;HOME   EQU     ORGIN     ;ABORT ADDRESS
0031 =          VERS    EQU     '1'       ;VERSION NUMBER
57A0 =          STACK   EQU     ORGIN-60H
0010 =          CSTAT   EQU     10H       ;CONSOLE STATUS
0011 =          CDATA   EQU     CSTAT+1   ;CONSOLE DATA
0010 =          CSTATO  EQU     CSTAT     ;CON OUT STATUS
0011 =          CDATAO  EQU     CSTATO+1  ;OUT DATA
0001 =          INMSK   EQU     1         ;INPUT MASK
0002 =          OMSK    EQU     2         ;OUTPUT MASK
                ;
57A0 =          PORTN   EQU     STACK     ;3 BYTES I/O
57A3 =          IBUFP   EQU     STACK+3   ;BUFFER POINTER
57A5 =          IBUFC   EQU     IBUFP+2   ;BUFFER COUNT
57A6 =          IBUFF   EQU     IBUFP+3   ;INPUT BUFFER
                ;
0008 =          CTRH    EQU     8         ;^H BACKSPACE
0009 =          TAB     EQU     9         ;^I
0011 =          CTRQ    EQU     17        ;^Q
0013 =          CTRS    EQU     19        ;^S
0018 =          CTRX    EQU     24        ;^X, ABORT
0008 =          BACKUP  EQU     CTRH      ;BACKUP CHAR
007F =          DEL     EQU     127       ;RUBOUT
001B =          ESC     EQU     27        ;ESCAPE
00F7 =          APOS    EQU     (39-'0') AND OFFH
000D =          CR      EQU     13        ;CARRIAGE RET
000A =          LF      EQU     10        ;LINE FEED
00DB =          INC     EQU     0DBH      ;IN OP CODE
00D3 =          OUTC    EQU     0D3H      ;OUT OP CODE
00C9 =          RETC    EQU     0C9H      ;RET OP CODE
                ;
                START:
5800 C34A58             JMP     COLD      ;COLD START
5803 C35358     RESTRT: JMP     WARM      ;WARM START
                ;
                ; CONSOLE INPUT ROUTINE
                ;
5806 CD1658     INPUTT: CALL    INSTAT    ;CHECK STATUS
5809 CA0658             JZ      INPUTT    ;NOT READY ***
580C DB11       INPUT2: IN      CDATA     ;GET BYTE
580E E67F               ANI     DEL
5810 FE18               CPI     CTRX      ;ABORT?
5812 CA0000             JZ      HOME      ;YES
5815 C9                 RET
                ;
                ; GET CONSOLE-INPUT STATUS
                ;
```

```
5816 DB10       INSTAT: IN      CSTAT
5818 E601               ANI     INMSK
581A C9                 RET
                ;
                ; CONSOLE OUTPUT ROUTINE
                ;
581B F5         OUTT:   PUSH    PSW
581C CD1658     OUT2:   CALL    INSTAT  ;INPUT?
581F CA3558             JZ      OUT4    ;NO ***
5822 CD0C58             CALL    INPUT2  ;GET INPUT
5825 FE13               CPI     CTRS    ;FREEZE?
5827 C21C58             JNZ     OUT2    ;NO
                ;
                ; FREEZE OUTPUT UNTIL ^Q OR ^X
                ;
582A CD0658     OUT3:   CALL    INPUTT  ;INPUT?
582D FE11               CPI     CTRQ    ;RESUME?
582F C22A58             JNZ     OUT3    ;NO
5832 C31C58             JMP     OUT2
                ;
5835 DB10       OUT4:   IN      CSTATO  ;CHECK STATUS
5837 E602               ANI     OMSK
5839 CA1C58             JZ      OUT2    ;NOT READY ***
583C F1                 POP     PSW
583D D311               OUT     CDATAO  ;SEND DATA
583F C9                 RET
                ;
5840 0D0A       SIGNON: DB      CR,LF
5842 205665             DB      ' Ver '
5847 3100               DW      VERS
5849 00                 DB      0
                ;
                ; CONTINUATION OF COLD START
                ;
584A 31A057     COLD:   LXI     SP,STACK
584D 114058             LXI     D,SIGNON ;MESSAGE
5850 CDE258             CALL    SENDM   ;SEND IT
                ;
                ; WARM-START ENTRY
                ;
5853 215358     WARM:   LXI     H,WARM  ;RETURN HERE
5856 E5                 PUSH    H
5857 CDB658             CALL    CRLF    ;NEW LINE
585A CD7758             CALL    INPLN   ;CONSOLE LINE
585D CDCC58             CALL    GETCH   ;GET CHAR
5860 FE44               CPI     'D'     ;DUMP
5862 CA5358             JZ      WARM    ;(VER 1)
                ;       JZ      DUMP    ;HEX/ASCII (2)
5865 FE43               CPI     'C'     ;CALL
5867 CA5358             JZ      WARM    ;(VER 1-2)
                ;       JZ      CALLS   ;SUBROUTINE (3)
586A FE47               CPI     'G'     ;GO
586C CA5358             JZ      WARM    ;(VER 1-2)
                ;       JZ      GO      ;SOMEWHERE (3)
586F FE4C               CPI     'L'     ;LOAD
5871 CA5358             JZ      WARM    ;(VER 1-3)
                ;       JZ      LOAD    ;INTO MEMORY (4)
5874 C35358             JMP     WARM    ;TRY AGAIN
                ;
```

```
                        ; INPUT A LINE FROM CONSOLE AND PUT IT
                        ; INTO THE BUFFER. CARRIAGE RETURN ENDS
                        ; THE LINE. RUBOUT OR ^H CORRECTS LAST
                        ; LAST ENTRY. CONTROL-X RESTARTS LINE.
                        ; OTHER CONTROL CHARACTERS ARE IGNORED
                        ;
5877 3E3E               INPLN:  MVI     A,'>'     ;PROMPT
5879 CD1B58                     CALL    OUTT
587C 21A657             INPL2:  LXI     H,IBUFF   ;BUFFER ADDR
587F 22A357                     SHLD    IBUFP     ;SAVE POINTER
5882 0E00                       MVI     C,0       ;COUNT
5884 CD0658             INPLI:  CALL    INPUTT    ;CONSOLE CHAR
5887 FE20                       CPI     ' '       ;CONTROL?
5889 DAA858                     JC      INPLC     ;YES
588C FE7F                       CPI     DEL       ;DELETE
588E CAC058                     JZ      INPLB     ;YES
5891 FE5B                       CPI     'Z'+1     ;UPPER CASE?
5893 DA9858                     JC      INPL3     ;YES
5896 E65F                       ANI     5FH       ;MAKE UPPER
5898 77                 INPL3:  MOV     M,A       ;INTO BUFFER
5899 3E20                       MVI     A,32      ;BUFFER SIZE
589B B9                         CMP     C         ;FULL?
589C CA8458                     JZ      INPLI     ;YES, LOOP
589F 7E                         MOV     A,M       ;GET CHAR
58A0 23                         INX     H         ;INCR POINTER
58A1 0C                         INR     C         ;AND COUNT
58A2 CD1B58             INPLE:  CALL    OUTT      ;SHOW CHAR
58A5 C38458                     JMP     INPLI     ;NEXT CHAR
                        ;
                        ; PROCESS CONTROL CHARACTER
                        ;
58A8 FE08               INPLC:  CPI     CTRH      ;^H?
58AA CAC058                     JZ      INPLB     ;YES
58AD FE0D                       CPI     CR        ;RETURN?
58AF C28458                     JNZ     INPLI     ;NO, IGNORE
                        ;
                        ; END OF INPUT LINE
                        ;
58B2 79                         MOV     A,C       ;COUNT
58B3 32A557                     STA     IBUFC     ;SAVE
                        ;
                        ; CARRIAGE-RETURN, LINE-FEED ROUTINE
                        ;
58B6 3E0D               CRLF:   MVI     A,CR
58B8 CD1B58                     CALL    OUTT      ;SEND CR
58BB 3E0A                       MVI     A,LF
58BD C31B58                     JMP     OUTT      ;SEND LF
                        ;
                        ; DELETE PRIOR CHARACTER IF ANY
                        ;
58C0 79                 INPLB:  MOV     A,C       ;CHAR COUNT
58C1 B7                         ORA     A         ;ZERO?
58C2 CA8458                     JZ      INPLI     ;YES
58C5 2B                         DCX     H         ;BACK POINTER
58C6 0D                         DCR     C         ;AND COUNT
58C7 3E08                       MVI     A,BACKUP ;CHARACTER
58C9 C3A258                     JMP     INPLE     ;SEND
                        ;
                        ; GET A CHARACTER FROM CONSOLE BUFFER
                        ; SET CARRY IF EMPTY
```

```
                    ;
58CC E5             GETCH:  PUSH    H           ;SAVE REGS
58CD 2AA357                 LHLD    IBUFP       ;GET POINTER
58D0 3AA557                 LDA     IBUFC       ;AND COUNT
58D3 D601                   SUI     1           ;DECR WITH CARRY
58D5 DAE058                 JC      GETC4       ;NO MORE CHAR
58D8 32A557                 STA     IBUFC       ;SAVE NEW COUNT
58DB 7E                     MOV     A,M         ;GET CHARACTER
58DC 23                     INX     H           ;INCR POINTER
58DD 22A357                 SHLD    IBUFP       ;AND SAVE
58E0 E1             GETC4:  POP     H           ;RESTORE REGS
58E1 C9                     RET
                    ;
                    ; SEND ASCII MESSAGE UNTIL BINARY ZERO
                    ; IS FOUND. POINTER IS D,E
                    ;
58E2 1A             SENDM:  LDAX    D           ;GET BYTE
58E3 B7                     ORA     A           ;ZERO?
58E4 C8                     RZ                  ;YES, DONE
58E5 CD1B58                 CALL    OUTT        ;SEND IT
58E8 13                     INX     D           ;POINTER
58E9 C3E258                 JMP     SENDM       ;NEXT
                    ;
58EC                        END
```

If you don't know the addresses of the console status and data registers and you are using the CP/M operating system, there is another approach you can take. You can use the I/O routines in the CP/M BIOS. The disadvantage of this approach is that CP/M must always be in place whenever the monitor is used. The BIOS entry address is given at memory address 1. The console status, input and output addresses are obtained by adding, respectively, 3, 6, and 9 to this address. The following I/O routines in Listing 6.1B can be substituted for the subroutines in Listing 6.1A starting with the label INPUTT and ending with the label OUT2. If this version is utilized, the addresses in the following sections will not agree with your assembly listings.

Listing 6.1B. Alternate I/O routines using CP/M BIOS.

```
                    ; CONSOLE INPUT ROUTINE USING CP/M BIOS
                    ;
                    INPUTT:
5806 E5             INPUT2: PUSH    H           ;SAVE REGISTERS
5807 D5                     PUSH    D
5808 C5                     PUSH    B
5809 211558                 LXI     H,IN5       ;RETURN ADDRESS
580C E5                     PUSH    H           ;PUT ON STACK
580D 2A0100                 LHLD    1           ;BIOS WARM START
5810 110600                 LXI     D,6         ;OFFSET TO INPUT
5813 19                     DAD     D           ;ADD IN
5814 E9                     PCHL                ;CALL BIOS
5815 C1             IN5:    POP     B           ;RESTORE REGISTERS
5816 D1                     POP     D
5817 E1                     POP     H
5818 FE18                   CPI     CTRX        ;ABORT?
581A CA0058                 JZ      START       ;YES
581D C9                     RET
```

```
                    ; GET CONSOLE-INPUT STATUS USING CP/M
                    ;
581E E5             INSTAT: PUSH    H       ;SAVE REGISTERS
581F D5                     PUSH    D
5820 C5                     PUSH    B
5821 212D58                 LXI     H,ST5   ;RETURN ADDRESS
5824 E5                     PUSH    H       ;PUT ON STACK
5825 2A0100                 LHLD    1       ;BIOS ENTRY
5828 110300                 LXI     D,3     ;OFFSET TO STATUS
582B 19                     DAD     D       ;ADD TO ADDR
582C E9                     PCHL            ;CALL BIOS
582D C1             ST5:    POP     B       ;RESTORE REGISTERS
582E D1                     POP     D
582F E1                     POP     H
5830 B7                     ORA     A
5831 C9                     RET
                    ;
                    ; CONSOLE OUTPUT ROUTINE USING CP/M BIOS
                    ;
5832 F5             OUTT:   PUSH    PSW     ;SAVE BYTE
5833 CD1E58         OUT2:   CALL    INSTAT  ;INPUT?
5836 CA4C58                 JZ      OUT4    ;NO
5839 CD0658                 CALL    INPUT2  ;GET INPUT
583C FE13                   CPI     CTRS    ;FREEZE?
583E C23358                 JNZ     OUT2    ;NO
5841 CD0658         OUT3:   CALL    INPUTT  ;INPUT?
5844 FE11                   CPI     CTRQ    ;RESUME?
5846 C24158                 JNZ     OUT3    ;NO
5849 C33358                 JMP     OUT2
                    ;
584C F1             OUT4:   POP     PSW     ;GET BYTE
584D E5                     PUSH    H       ;SAVE REGISTERS
584E D5                     PUSH    D
584F C5                     PUSH    B
5850 4F                     MOV     C,A     ;MOVE BYTE
5851 F5                     PUSH    PSW
5852 215E58                 LXI     H,OUT5  ;RETURN ADDRESS
5855 E5                     PUSH    H       ;PUT ON STACK
5856 2A0100                 LHLD    1       ;BIOS ENTRY
5859 110900                 LXI     D,9     ;OFFSET TO OUTPUT
585C 19                     DAD     D       ;ADD TOGETHER
585D E9                     PCHL            ;CALL BIOS
585E F1             OUT5:   POP     PSW     ;RESTORE REGISTERS
585F C1                     POP     B
5860 D1                     POP     D
5861 E1                     POP     H
5862 C9                     RET
```

Some of the constants such as PORTN will not be used at this time. However, their inclusion now will simplify things later. There are four occurrences of the dummy instruction

```
JZ      WARM
```

following the label WARM. Each is followed by an instruction that will be needed later. These latter instructions are preceded by a semicolon so that they will be treated as comments by the assembler.

There are some other matters that may need to be considered. One has to do with the sense of the input and output ready flags. There are three conditional jump instructions based on console-ready flags that display a logical 1 (active high) when ready. If your flags are inverted, that is, they present a logic zero when ready, then the three JZ commands must be changed to JNZ commands. These lines, indicated by three stars in the listing below, should be changed to

```
INPUTT: CALL    INSTAT  ;CHECK STATUS
        JNZ     INPUTT  ;NOT READY ***
        . . .
OUT2:   CALL    INSTAT  ;INPUT?
        JNZ     OUT4    ;NO   ***
        . . .
OUT4:   IN      CSTAT   ;CHECK STATUS
        ANI     OMSK
        JNZ     OUT2    ;NOT READY ***
```

The routine that corrects keyboard errors is programmed for a video console. If you have a console printer instead, change the backspace character to a slash.

```
BACKUP  EQU     '/'     ;CORRECTION
```

This will print a slash when an error is corrected. Otherwise the printer will back up during error correction, overstriking the old character with the new. You may also need to add some nulls after each carriage return. The problem here will be evidenced by missing characters at the beginning of each line. The solution is to place additional instructions in the subroutine called CRLF. Replace the last statement in this routine with the following.

```
CALL    OUTT    ;SEND LINE FEED
XRA     A       ;GET A ZERO
CALL    OUTT    ;SEND NULL
CALL    OUTT    ;AND ANOTHER
. . .
. . .   (one line for each null)
JMP     OUTT    ;LAST NULL
```

The rest of the program can be copied directly as it is. The abort command is a control-X. Initially, the abort address of HOME will be needed to leave the new monitor and return to your regular system. We will change this in version 3 when we will add a routine for branching to any memory address.

If you have a TRS-80 Model I, you won't have a control key. Therefore, you will have to change several of the commands shown in the listing. The original commands follow.

```
CTRH      (Control-H)
TAB       (Control-I)
CTRQ      (Control-Q)
CTRS      (Control-S)
CTRX      (Control-X)
DEL       (DEL/RUB)
ESC       (Escape)
```

After you have finished typing the program, exit from the editor and assemble the source program with the assembler. The command line might be

```
B>A:ASM  MON1
```

or

```
B>A:MAC  MON1
```

for the Digital Research assemblers. These two assemblers will produce two files.

```
MON1.ASM        (assembly listing)
MON1.HEX        (hex code)
```

In addition, MAC will produce a symbol table

```
MON1.SYM        (symbol table)
```

Inspect the hex code given in the assembly listing to see that it matches the corresponding instructions given in this chapter. These 8080 listings have all been generated with the Digital Research assembler MAC. This assembler displays the hex code for 16-bit operands in the usual reverse order. The low-order byte appears first followed by the high-order byte. Thus:

```
CD1B58   means  CALL  581B  and
C38458   means  JMP   5884
```

By contrast, the assembly listing produced by the Microsoft assembler reverses the usual order of the two bytes. The high-order byte is given first; this is followed by the low-order byte. In this case, the listing

```
CD 581B   means  CALL  581B  and
C3 5884   means  JMP   5884
```

The next step is to load the hex program into memory using the debugger. The CP/M command would be

```
B>A:DDT MON1.HEX   or
B>A:SID MON1.HEX
```

Now branch to the beginning of the monitor using the debugger G command.

```
G5800
```

The first thing that the monitor will do is display the version number on the first line and a prompt symbol of > underneath it.

Try out this first version by typing a series of letters and numbers. Each character that is typed should appear (echo) on the console. Try the correction keys. Typing either a control-H (backspace) or the RUB/DEL key should back up the cursor on a video terminal. Type a carriage return. The prompt symbol should appear at the beginning of the next line. If all of the features are working properly, type a control-X to return to your regular system. If something appears to be wrong, carefully compare your assembly listing with the one given in Listing 6.1A. *Don't proceed to the next version until the current one is working.*

VERSION 2: A MEMORY DISPLAY

A provision for examining the contents of memory will now be added. This routine is called a *memory dump*, or dump for short; it displays the contents of memory in both hex and ASCII notation. The dump feature is initiated with a command of D followed by the address limits in hexadecimal. For example, the statement

```
>D100 18F
```

will dump memory from address 100 to 18F hex. The first address (100 in this case) must immediately follow the letter D. A space is typed and then the second address (18F in this case) is entered. Leading zeros are unnecessary.

Each line will display 16 memory locations. The hexadecimal address of the first location will appear at the beginning of the line. Then the hexadecimal representation of the contents will follow, two characters per byte. These are arranged in four groups of four bytes. The ASCII representations of the data will be given at the end of the line if printable. Otherwise, a period is given. A dump of the first line of the monitor might look like this.

```
  >D5800 580F   (your command)

5800 C35C58C3 6558CD16 58CA0658 DB11E67F .\X.eX..X..X....
```

Use your system editor to make the necessary alterations and additions to version 1. First, change the version number at the beginning of the program.

```
VERS     EQU     '2'     ;VERSION NUMBER
```

Next, locate the instruction

```
JZ      DUMP
```

that follows the label WARM. Remove the semicolon at the beginning of this line. Also delete the line just before it that jumps to WARM. The region should now look like this.

```
CALL    GETCH
CPI     'D'     ;DUMP?
JZ      DUMP
. . .   . . .
```

The remaining instructions, shown in Listing 6.2, will be placed at the end of version 1, just preceding the END statement. It might be easier to delete the END statement, type in the new code, and then add a new END statement. The END statement is usually optional, anyway. One of the subroutines (READHL) will translate the dump limits from ASCII-encoded hexadecimal into binary. One routine gets both the start and the stop address (using READHL) then checks to see that the second address is larger than the first. If the second address is smaller than the first, then the task will be aborted. Subroutine OUTHEX will convert the binary data already in memory into ASCII-coded hex for output to the console. Subroutine TSTOP is used to determine when to terminate the dump process. Finally, an error routine (ERROR) will be needed in case an invalid character is entered by the user.

Listing 6.2. Memory display

```
                ; DUMP MEMORY IN HEXADECIMAL AND ASCII
                ;
58EC CD2D59     DUMP:   CALL    RDHLDE  ;RANGE
58EF CD8359     DUMP2:  CALL    CRHL    ;NEW LINE
58F2 4E         DUMP3:  MOV     C,M     ;GET BYTE
58F3 CD9359             CALL    OUTHX   ;PRINT
58F6 23                 INX     H       ;POINTER
58F7 7D                 MOV     A,L
58F8 E60F               ANI     0FH     ;LINE END?
58FA CA0559             JZ      DUMP4   ;YES, ASCII
58FD E603               ANI     3       ;SPACE
58FF CC8E59             CZ      OUTSP   ; 4 BYTES
5902 C3F258             JMP     DUMP3   ;NEXT HEX
5905 CD8E59     DUMP4:  CALL    OUTSP
5908 D5                 PUSH    D
5909 11F0FF             LXI     D,-10H  ;RESET LINE
590C 19                 DAD     D
590D D1                 POP     D
590E CD1D59     DUMP5:  CALL    PASCI   ;ASCII DUMP
5911 CDA759             CALL    TSTOP   ;DONE?
5914 7D                 MOV     A,L     ;NO
5915 E60F               ANI     0FH     ;LINE END?
5917 C20E59             JNZ     DUMP5   ;NO
591A C3EF58             JMP     DUMP2
```

```
                    ;
                    ; DISPLAY MEMORY BYTE IN ASCII IF
                    ; POSSIBLE, OTHERWISE GIVE DECIMAL PNT
                    ;
591D 7E             PASCI:  MOV     A,M     ;GET BYTE
591E FE7F                   CPI     DEL     ;HIGH BIT ON?
5920 D22859                 JNC     PASC2   ;YES
5923 FE20                   CPI     ' '     ;CONTROL CHAR?
5925 D22A59                 JNC     PASC3   ;NO
5928 3E2E           PASC2:  MVI     A,'.'   ;CHANGE TO DOT
592A C31B58         PASC3:  JMP     OUTT    ;SEND
                    ;
                    ; GET H,L AND D,E FROM CONSOLE
                    ; CHECK THAT D,E IS LARGER
                    ;
592D CD3859         RDHLDE: CALL    HHLDE
5930 7B             RDHLD2: MOV     A,E
5931 95                     SUB     L       ;E - L
5932 7A                     MOV     A,D
5933 9C                     SBB     H       ;D - H
5934 DA7B59                 JC      ERROR   ;H,L BIGGER
5937 C9                     RET
                    ;
                    ; INPUT H,L AND D,E. SEE THAT
                    ; 2 ADDRESSES ARE ENTERED
                    ;
5938 CD4459         HHLDE:  CALL    READHL  ;H,L
593B DA7B59                 JC      ERROR   ;ONLY 1 ADDR
593E EB                     XCHG            ;SAVE IN D,E
593F CD4459                 CALL    READHL  ;D,E
5942 EB                     XCHG            ;PUT BACK
5943 C9                     RET
                    ;
                    ; INPUT H,L FROM CONSOLE
                    ;
5944 D5             READHL: PUSH    D
5945 C5                     PUSH    B       ;SAVE REGS
5946 210000                 LXI     H,0     ;CLEAR
5949 CDCC58         RDHL2:  CALL    GETCH   ;GET CHAR
594C DA6859                 JC      RDHL5   ;LINE END
594F CD6B59                 CALL    NIB     ;TO BINARY
5952 DA5E59                 JC      RDHL4   ;NOT HEX
5955 29                     DAD     H       ;TIMES 2
5956 29                     DAD     H       ;TIMES 4
5957 29                     DAD     H       ;TIMES 8
5958 29                     DAD     H       ;TIMES 16
5959 B5                     ORA     L       ;ADD NEW CHAR
595A 6F                     MOV     L,A
595B C34959                 JMP     RDHL2   ;NEXT
                    ;
                    ; CHECK FOR BLANK AT END
                    ;
595E FEF7           RDHL4:  CPI     APOS    ;APOSTROPHE
5960 CA6859                 JZ      RDHL5   ;ASCII INPUT
5963 FEF0                   CPI     (' '-'0') AND 0FFH
5965 C27B59                 JNZ     ERROR   ;NO
5968 C1             RDHL5:  POP     B
5969 D1                     POP     D       ;RESTORE
596A C9                     RET
```

```
                    ;
                    ; CONVERT ASCII CHARACTERS TO BINARY
                    ;
596B D630           NIB:    SUI     '0'     ;ASCII BIAS
596D D8                     RC              ; < 0
596E FE17                   CPI     'F'-'0'+1
5970 3F                     CMC             ;INVERT
5971 D8                     RC              ;ERROR, > F
5972 FE0A                   CPI     10
5974 3F                     CMC             ;INVERT
5975 D0                     RNC             ;NUMBER 0-9
5976 D607                   SUI     'A'-'9'-1
5978 FE0A                   CPI     10      ;SKIP : TO
597A C9                     RET             ;LETTER A-F
                    ;
                    ; PRINT ? ON IMPROPER INPUT
                    ;
597B 3E3F           ERROR:  MVI     A,'?'
597D CD1B58                 CALL    OUTT
5980 C30058                 JMP     START   ;TRY AGAIN
                    ;
                    ; START NEW LINE, GIVE ADDRESS
                    ;
5983 CDB658         CRHL:   CALL    CRLF    ;NEW LINE
                    ;
                    ; PRINT H,L IN HEX
                    ;
5986 4C             OUTHL:  MOV     C,H
5987 CD9359                 CALL    OUTHX   ;H
598A 4D             OUTLL:  MOV     C,L
                    ;
                    ; OUTPUT HEX BYTE FROM C AND A SPACE
                    ;
598B CD9359         OUTHEX: CALL    OUTHX
                    ;
                    ; OUTPUT A SPACE
                    ;
598E 3E20           OUTSP:  MVI     A,' '
5990 C31B58                 JMP     OUTT
                    ;
                    ; OUTPUT A HEX BYTE FROM C
                    ; BINARY TO ASCII HEX CONVERSION
                    ;
5993 79             OUTHX:  MOV     A,C
5994 1F                     RAR             ;ROTATE
5995 1F                     RAR             ; FOUR
5996 1F                     RAR             ; BITS TO
5997 1F                     RAR             ; RIGHT
5998 CD9C59                 CALL    HEX1    ;UPPER CHAR
599B 79                     MOV     A,C     ;LOWER CHAR
599C E60F           HEX1:   ANI     0FH     ;TAKE 4 BITS
599E C690                   ADI     90H
59A0 27                     DAA             ;DAA TRICK
59A1 CE40                   ACI     40H
59A3 27                     DAA
59A4 C31B58                 JMP     OUTT
                    ;
                    ; CHECK FOR END, H,L MINUS D,E
                    ; INCREMENT H,L
```

```
                   ;
59A7 23            TSTOP:  INX     H
59A8 7B                    MOV     A,E
59A9 95                    SUB     L          ; E - L
59AA 7A                    MOV     A,D
59AB 9C                    SBB     H          ; D - H
59AC D0                    RNC                ;NOT DONE
59AD E1                    POP     H          ;RAISE STACK
59AE C9                    RET
                   ;
59B1                       END
```

Type up the new instructions, then, after you leave the editor, rename the new file. The CP/M command will be

```
REN  MON2.ASM=MON1.ASM
```

Rename the backup file to its original name.

```
REN  MON1.ASM=MON1.BAK
```

Assemble version 2 and load it into memory. Start it up by branching to the address of START. Again, the version number should be printed, and the prompt symbol should appear. Test the new feature by dumping a portion of the monitor.

```
>D5800 585F
```

Be sure to type a carriage return at the end of the line. Input errors can be corrected by typing a backspace or DEL. Check to see that the hex code displayed on the screen matches the assembly listing code. Most of the ASCII representation will be meaningless. But the section from 5842 to 585A hex will read

```
Ver 2
```

Now test the scroll-freeze commands. Dump a large section of memory.

```
>D0 1000
```

Type a control-S as the data are being displayed on the console. The console screen should freeze. Now type a control-Q. The screen should again resume displaying the data. The commands of Control-S and control-Q will alternately freeze and resume the scrolling.

Try the routine that checks for proper dump limits by typing a larger address first, then a smaller address.

```
D300 200
```

As a result of this improper input, a question mark should be printed. Then the prompt will appear on a new line. If everything is all right, return to your regular system by entering a control-X.

If version 2 does not perform satisfactorily, compare the hex code in your assembly listing with the values given in Listing 6.2 for the new code. Correct any errors, reassemble the program, and try it again.

VERSION 3: A CALL AND GO ROUTINE

Now that both hex-to-binary and binary-to-hex routines are available, we can easily include new features. A CALL routine and a GO routine will be added in version 3. These routines will allow you to branch to any address in memory. The GO command will be useful for testing subroutines. For this latter command, the monitor warm-start address (WARM) is on the stack when the call is made. A subroutine can be called with the C command. The execution of an RET instruction at the end of the subroutine will cause a return to the monitor.

First, change the version number to 3. Then find the instructions corresponding to the C and G commands after the label WARM. Remove the semicolons from the beginning of the lines that branch to CALLS and GO. Delete the prior lines that jump to WARM. The program should now look like

```
CPI     'C'     ;CALL?
JZ      CALLS
CPI     'G'     ;GO?
JZ      GO
```

The remaining lines of code (and some comments) are placed at the end of the source program just prior to the END statement. They are given in Listing 6.3.

Listing 6.3. A CALL and a GO routine.

```
                ; ROUTINE TO GO ANYWHERE IN MEMORY
                ; ADDRESS OF WARM IS ON STACK, SO A
                ; SIMPLE RET WILL RETURN TO THIS MONITOR
                ;
59AF E1         GO:     POP     H       ;RAISE STACK
59B0 CD4459     CALLS:  CALL    READHL  ;GET ADDRESS
59B3 E9                 PCHL            ;GO THERE
                ;
59B4                    END
```

Another important change should be made at this time. Since we can now branch to any place in memory with the GO command, we can change the abort command, control-X. Redefine HOME near the beginning of the source program so that an abort command of control-X will restart the monitor.

```
HOME      EQU      ORGIN    ;ABORT ADDRESS
```

This line was originally entered as a comment. Remove the semicolon at the beginning of the line and delete the previous line.

Assemble the new version and load it into memory. Branch to the monitor and try the dump routine as before. Try the CALL feature by calling the monitor itself.

```
>C5800
```

The cold-start message should appear. Now use the GO routine to return to your main system. If the GO address is zero, then no argument need follow the G command.

```
>G
```

VERSION 4: A MEMORY-LOAD ROUTINE

In version 2 we added a routine that could be used to inspect any memory location. A routine which can be used to change memory will now be added. Change the version number to 4. Locate the instruction

```
;        JZ       LOAD
```

following WARM. Remove the semicolon at the beginning of the line. Delete the original JZ WARM on the prior line. The program should now look like

```
CPI      'L'
JZ       LOAD
```

Add the load routines shown in Listing 6.4 to the end of the source program.

Listing 6.4. A memory-load routine.

```
                    ; LOAD HEX OR ASCII CHAR INTO MEMORY
                    ; FROM CONSOLE. CHECK TO SEE IF
                    ; THE DATA ACTUALLY GOT THERE
                    ; APOSTROPHE PRECEEDS ASCII CHAR
                    ; CARRIAGE RETURN PASSES OVER LOCATION
                    ;
59B4 CD4459         LOAD:   CALL    READHL   ;ADDRESS
59B7 CD8659         LOAD2:  CALL    OUTHL    ;PRINT IT
59BA CD1D59                 CALL    PASCI    ;ASCII
59BD CD8E59                 CALL    OUTSP
59C0 4E                     MOV     C,M      ;ORIG BYTE
59C1 CD8B59                 CALL    OUTHEX   ;HEX
59C4 E5                     PUSH    H        ;SAVE PNTR
59C5 CD7C58                 CALL    INPL2    ;INPUT
59C8 CD4459                 CALL    READHL   ; BYTE
59CB 45                     MOV     B,L      ; TO B
```

```
59CC E1                 POP     H
59CD FEF7               CPI     APOS
59CF CADE59             JZ      LOAD6      ;ASCII INPUT
59D2 79                 MOV     A,C        ;HOW MANY?
59D3 B7                 ORA     A          ;NONE?
59D4 CADA59             JZ      LOAD3      ;YES
59D7 CDE559     LOAD4:  CALL    CHEKM      ;INTO MEMORY
59DA 23         LOAD3:  INX     H          ;POINTER
59DB C3B759             JMP     LOAD2
                ;
                ; LOAD ASCII CHARACTER
                ;
59DE CDCC58     LOAD6:  CALL    GETCH
59E1 47                 MOV     B,A
59E2 C3D759             JMP     LOAD4
                ;
                ; COPY BYTE FROM B TO MEMORY
                ; AND SEE THAT IT GOT THERE
                ;
59E5 70         CHEKM:  MOV     M,B        ;PUT IN MEM
59E6 7E                 MOV     A,M        ;GET BACK
59E7 B8                 CMP     B          ;SAME?
59E8 C8                 RZ                 ;YES
59E9 C37B59             JMP     ERROR      ;BAD
                ;
59EC                    END
```

Assemble version 4 and compare the assembly listing of the new part to Listing 6.4. Load the new program and branch to the beginning. Recheck the dump command by examining the new code for the load routine

```
>D59B4 59EB
```

Now try the load command. Great care must be taken when typing the load address. This command will actually change the contents of memory, including the monitor itself.

Type the letter L, the hexadecimal address, and a carriage return. The response will be the address that was typed and the current contents of that memory location. The data are represented two ways: in ASCII and in hex. If the ASCII value is not a printable character, it is rendered as a period.

The displayed location can now be changed by typing the new value and a carriage return. The data can be entered in several ways. It can be in the form of one or two hex characters. If more than two characters are entered, only the last two are actually used. This allows you to correct an error by continuing to type. Errors can also be corrected with the backspace or the DEL/RUB key. A single ASCII character can be entered into memory by preceding it with an apostrophe.

As each new value and a carriage return is typed, the next address and the present data value will appear. In this way, a machine-language routine can be entered from the console. Of course, using an assembler is a more efficient way to generate a long program. But our load routine will be useful for making simple changes or for writing short routines.

The load command is terminated by typing a control-X (if you redefined HOME as ORGIN back in version 3). It is also terminated if you enter a nonhex character. Control then returns to the monitor. If the load command is used to revise existing code, another feature is useful. A carriage return is given without entering any data. The memory pointer then skips over the current location and the corresponding value is not changed.

After each revised byte is entered into memory, the monitor checks to see that the new value is correct. If an attempt is made to write into protected, nonexistent, or defective memory, the load process is terminated and a question mark is printed.

Try the load routine by entering the following five bytes into a convenient location such as 4000 hex.

```
3E  7 D3 XX C9
```

This sequence corresponds to the assembly language program

```
3E07    MVI     A,7
D3XX    OUT     XX
C9      RET
```

The value of XX is the console-data address (CDATA in the source program). Check the code with the dump command.

```
D4000 4004
```

Now use the CALL command to execute the routine

```
C4000
```

The console bell should sound and control will return to the command level of our monitor.

VERSION 5: USEFUL ENTRY POINTS

Changes to the first four versions were made for the most part by adding new instructions to the end of the existing program. For versions 5, 6, and 7, we are going to start the process over to some extent by inserting some new instructions in the middle of the existing program.

At the beginning of the monitor there are two jump instructions.

```
JMP     COLD
JMP     WARM
```

Entry points such as these are sometimes called *vectors*. The first jump to COLD is the initial, cold-start entry point into the monitor. Stack initialization and printing of the sign-on message occur at this time. But other housekeeping chores, such as interface initialization, could be performed in this section. The second vector causes a jump to WARM, a restart entry point that does not alter the stack pointer.

We will now insert some additional vectors after these first two. The additional jumps will provide fixed entry points to useful subroutines in the monitor. These routines can then be easily called by other programs outside the monitor. Since these jump instructions are all at the beginning of the monitor, their addresses won't change when the monitor is altered. Furthermore, new vectors can be added to the end of the group without affecting those already present.

Place the five jump instructions shown in Listing 6.5 at the beginning of the monitor just after the first two (START and RESTRT).

Listing 6.5. Some useful entry points.

```
                 ; VECTORS TO USEFUL ROUTINES
                 ;
5806 C32A58      COUT:     JMP      OUTT      ;OUTPUT CHAR
5809 C31558      CIN:      JMP      INPUTT    ;INPUT CHAR
580C C3D058      INLN:     JMP      INPLN     ;INPUT LINE
580F C32559      GCHAR:    JMP      GETCH     ;GET CHAR
5812 C3EC59      OUTH:     JMP      OUTHX     ;BIN TO HEX
                 ;
```

Reassemble the monitor, load it into memory, and try the DUMP, LOAD, and GO routines again to be sure that they still work. Now, when separate, external routines are written, they need not contain subroutines for console input, output, conversion of binary to hex, and so on.

A character can be displayed on the console by calling COUT with the character in the accumulator. A single console character is obtained by calling CIN. The byte is returned in the accumulator.

An entire line of characters can be easily obtained by calling the line-input entry INLN. As each character is typed, it is automatically printed on the console. The error-correction commands are available at this time. The backspace and DEL/RUB keys can be used to delete the previously typed character. A line is normally terminated with a carriage return. After the console-input buffer has been filled by a call to INLN, the GCHAR address can be called.

A character is returned in the accumulator for each call to GCHAR. When the input buffer has been exhausted, the carry flag is set. Typing a control-X will abort a routine and return control to the monitor. Therefore, it is not necessary to include an abort routine in separate, external programs.

The fifth new entry point will perform a conversion from binary to ASCII-coded hexadecimal. This will allow display of individual memory locations or any of the CPU registers. The byte to be converted is placed in the C register and the address of OUTH is called. The accumulator is also used by the conversion routine in this case, so it may be necessary to save the accumulator's original contents on the stack by using a PUSH instruction.

Use the monitor to write the following short routine. This program will demontrate the new vectors.

```
3E07     MVI     A,7
C30658   JMP     COUT
```

This program, which is similar to the one written in the last section, can be placed almost anywhere in memory. This time, however, there is no need to worry about the output device address since the monitor takes care of this. Branch to the routine by giving the monitor command of C and the address

```
>C4000
```

The monitor output routine will ring the console bell, then cause a return to the address WARM.

The monitor now contains a bare minimum of features. The DUMP, LOAD, GO, and CALL routines can be used to write and inspect simple routines. The several vectors located at the beginning of the monitor, allow easy access to useful subroutines within. These vectors will greatly simplify the task of writing and debugging simple routines.

At this point, you may wish to go on to Chapter 8 and try some of the routines discussed there. Otherwise, continue in this chapter as we add more features to the monitor. The new features will include memory fill and zero, memory search, ASCII load and dump, input from and output through any port, a memory test, byte replacement, and memory comparison.

VERSION 6: AUTOMATIC MEMORY SIZE

A routine that will automatically find the top of usable memory will be added for version 6. The routine is executed each time a cold or warm start is performed. The first byte of memory in each page of 256 bytes of memory is checked, starting with page zero. The byte in memory is moved to the accumulator, complemented, then written back to the same memory location. The result is compared to the accumulator to see if it is the same. If so, then the byte in the accumulator is complemented back to the original byte and it is written back into memory. This effectively restores the original byte.

Each page of memory is checked in this way until the monitor stack area is encountered or until defective, missing, or protected memory is found. The hexadecimal value of this top page is printed just preceding the monitor greater-than prompt (>). For example, if the monitor starts at 5800 hex and the stack is located at 57A0, then the prompt will appear as

```
57>
```

This routine provides a regular, continuous check on the memory size. It does not check all of the memory, but only the first byte of each 256 bytes of the page. Nevertheless, this check may point up potential problems. A more complete memory test program will be added in version 15.

Listing 6.6. Automatic memory check.

```
                     ;
                     ; FIND TOP OF USABLE MEMORY.
                     ; CHECK FIRST BYTE OF EACH PAGE OF MEMORY
                     ; STARTING AT ADDRESS ZERO.  STOP AT STACK
                     ; OR MISSING/DEFECTIVE/PROTECTED MEMORY.
                     ; DISPLAY HIGH BYTE OF MEMORY TOP.

5865 210000                  LXI     H,0       ;PAGE ZERO
5868 0657                    MVI     B,STACK SHR 8
586A 7E              NPAGE:  MOV     A,M       ;GET BYTE
586B 2F                      CMA               ;COMPLEMENT
586C 77                      MOV     M,A       ;PUT IT BACK
586D BE                      CMP     M         ;SAME?
586E C27858                  JNZ     MSIZE     ;NO, MEM TOP
5871 2F                      CMA               ;ORIG BYTE
5872 77                      MOV     M,A       ;RESTORE IT
5873 24                      INR     H         ;NEXT PAGE
5874 05                      DCR     B
5875 C26A58                  JNZ     NPAGE     ;KEEP GOING
5878 4C              MSIZE:  MOV     C,H       ;MEM TOP
5879 CD0F59                  CALL    CRLF      ;NEW LINE
587C CDEC59                  CALL    OUTHX     ;PRINT MEM SIZE
587F CDD058                  CALL    INPLN     ;CONSOLE LINE
5882 CD2559                  CALL    GETCH     ;FIRST CHAR
                     ;
```

Insert the new instructions shown in Listing 6.6 right after the PUSH H instruction that follows the label WARM.

```
WARM:   LXI     H,WARM   ;RET TO
        PUSH    H        ; HERE

        (add new code here)
```

If your assembler does not have the shift-right operation SHR, then just code the high half of the stack address. The second line in Listing 6.6 might look like this instead.

```
MVI     B,57H
```

Assemble the new version. Check the assembly listing to see that the new additions are correct. Then try out version 6.

The symbol table at this point follows.

```
symbols:

00F7 APOS      0008 BACKUP    59B0 CALLS     0011 CDATA
0011 CDATAO    59E5 CHEKM     584A COLD      000D CR
5983 CRHL      58B6 CRLF      0010 CSTAT     0010 CSTATO
0008 CTRH      0011 CTRQ      0013 CTRS      0018 CTRX
007F DEL       58EC DUMP      58EF DUMP2     58F2 DUMP3
5905 DUMP4     590E DUMP5     597B ERROR     001B ESC
58E0 GETC4     58CC GETCH     59AF GO        599C HEX1
5938 HHLDE     0000 HOME      57A5 IBUFC     57A6 IBUFF
57A3 IBUFP     00DB INC       0001 INMSK     587C. INPL2
5898 INPL3     58C0 INPLB     58A8 INPLC     58A2 INPLE
5884 INPLI     5877 INPLN     580C INPUT2    5806 INPUTT
5816 INSTAT    000A LF        59B4 LOAD      59B7 LOAD2
59DA LOAD3     59D7 LOAD4     59DE LOAD6     596B NIB
0002 OMSK      5800 ORGIN     581C OUT2      582A OUT3
5835 OUT4      00D3 OUTC      598B OUTHEX    5986 OUTHL
5993 OUTHX     598A OUTLL     598E OUTSP     581B OUTT
5928 PASC2     592A PASC3     591D PASCI     57A0 PORTN
5949 RDHL2     595E RDHL4     5968 RDHL5     5930 RDHLD2
592D RDHLDE    5944 READHL    5803 RESTRT    00C9 RETC
58E2 SENDM     5840 SIGNON    57A0 STACK     5800 START
0009 TAB       0018 TOP       59A7 TSTOP     0031 VERS
5853 WARM
```

VERSION 7: COMMAND-BRANCH TABLE

Before incorporating additional features into the monitor, we should make a fundamental change in the *command processor routine.* This routine interprets the initial character of the command line. The routine looks for commands beginning with the letter D, C, G, or L. Five bytes of instruction are needed for each one of these commands. For example, the LOAD routine uses the instructions

```
CPI     'L'
JZ      LOAD
```

Since there are 26 letters of the alphabet, there will eventually be 26 times 5, or 130 bytes needed if 26 different commands are incorporated. This approach is satisfactory for a short table, but there is a better approach when there are many entries.

An alternate method is to use a command branch table. This method only requires two bytes per table entry plus 23 bytes of decoding instructions. The disadvantage of this method is that all 26 table entries will have to be allocated, even if only a few are needed. Thus, there may be a lot of unallocated table entries.

Delete the 13 lines of program immediately following the command of CALL GETCH, just after the label MSIZE.

```
CPI     'D'     ;DUMP <delete 13 lines>   <====!
. . .                                          !
. . .                                          !
JMP     WARM    ;TRY AGAIN                <====!
```

The new code that will be added is given in Listing 6.7. Notice that there are 26 table entries. Each line corresponds to one letter of the alphabet. At this time, most of the entries refer to the error routine called ERROR. This is because we have not yet incorporated many features. As new features are added to the monitor, these error references will be replaced by the desired subroutine names.

Listing 6.7. Command-branch table.

```
                    ; MAIN COMMAND PROCESSOR
                    ;
5885 D641                   SUI     'A'       ;CONVERT OFFSET
5887 DAD459                 JC      ERROR     ; < A
588A FE1A                   CPI     'Z'-'A'+1
588C D2D459                 JNC     ERROR     ; > Z
588F 87                     ADD     A         ;DOUBLE
5890 219C58                 LXI     H,TABLE   ;START
5893 1600                   MVI     D,0
5895 5F                     MOV     E,A       ;OFFSET
5896 19                     DAD     D         ;ADD TO TABLE
5897 5E                     MOV     E,M       ;LOW BYTE
5898 23                     INX     H
5899 56                     MOV     D,M       ;HIGH BYTE
589A EB                     XCHG              ;INTO H,L
589B E9                     PCHL              ;GO THERE
                    ;
                    ; COMMAND TABLE
                    ;
589C D459           TABLE:  DW      ERROR     ;A, ASCII
589E D459                   DW      ERROR     ;B
58A0 095A                   DW      CALLS     ;C, CALL SUBR
58A2 4559                   DW      DUMP      ;D, DUMP
58A4 D459                   DW      ERROR     ;E
58A6 D459                   DW      ERROR     ;F, FILL
58A8 085A                   DW      GO        ;G, GO
58AA D459                   DW      ERROR     ;H, HEX MATH
58AC D459                   DW      ERROR     ;I, PORT INPUT
58AE D459                   DW      ERROR     ;J, MEMORY TEST
58B0 D459                   DW      ERROR     ;K
58B2 0D5A                   DW      LOAD      ;L, LOAD
58B4 D459                   DW      ERROR     ;M, MOVE
58B6 D459                   DW      ERROR     ;N
58B8 D459                   DW      ERROR     ;O, PORT OUTPUT
58BA D459                   DW      ERROR     ;P
58BC D459                   DW      ERROR     ;Q
58BE D459                   DW      ERROR     ;R, REPLACE
58C0 D459                   DW      ERROR     ;S, SEARCH
58C2 D459                   DW      ERROR     ;T
58C4 D459                   DW      ERROR     ;U
58C6 D459                   DW      ERROR     ;V, VERIFY MEM
58C8 D459                   DW      ERROR     ;W
58CA D459                   DW      ERROR     ;X, STACK POINTER
58CC D459                   DW      ERROR     ;Y
58CE D459                   DW      ERROR     ;Z, ZERO
```

Generate the new version, assemble it, and compare the resulting assembly listing to the one given in Listing 6.7. Try out version 7. It should behave exactly like version 6. It will be a bit longer than version 6 at this stage, but it will not grow as rapidly as we add new features. In the remainder of this chapter, we will add the new subroutines to the end of source program. The label for the subroutine will be placed in the appropriate place of the command branch table.

VERSION 8: DISPLAY THE STACK POINTER

By adding seven bytes of new code, we will be able to examine our monitor's stack pointer. This will alert us to a possible problem with the monitor itself. We may find, for example, that as we use the monitor, the stack tends to grow up or down in memory, rather than remain in the same place. This is undesirable and indicates that we are not properly lowering or raising the stack somewhere in the program. For example, subroutine TSTOP increments the pointer then checks to see if the current task should be terminated. If so, the stack is raised with a POP instruction. Then a return instruction skips one level of subroutines, so that control returns to the address of WARM.

Change the version number to 8. Also change the entry in the command table that corresponds to the command of X. This is the third from the last entry. Delete the word ERROR and replace it with the word REGS.

```
DW        REGS    ;X, STK POINTER
```

Then go to the end of the source program. We will make a minor change in subroutine CHECKM, the last subroutine in the monitor. Then the new instructions will be added. Delete the END statement and the instruction just prior to the END statement.

```
JMP       ERROR   ;BAD
```

Then add the new instructions as shown in Listing 6.8.

Listing 6.8. Display the stack pointer.

```
5A42 F1          ERRP:   POP     PSW     ;RAISE STACK
5A43 3E42        ERRB:   MVI     A,'B'   ;BAD
5A45 CD2A58      ERR2:   CALL    OUTT
5A48 CDE759              CALL    OUTSP
5A4B C3DF59              JMP     OUTHL   ;POINTER
                 ;
                 ; DISPLAY STACK POINTER REGISTER
                 ;
5A4E 210000      REGS:   LXI     H,0
5A51 39                  DAD     SP
5A52 C3DF59              JMP     OUTHL
```

Reassemble the monitor and try it out. First give the X command (with no argument). Make a note of the value given for the stack pointer. Now, try other monitor features such as the dump and load commands. After each of these commands, give the X command to see that the stack pointer remains in the same place.

Try the separate routine that rings the console bell. This routine, which was written for version 4, may have to be rewritten if it was destroyed by the assembler or the editor. Again, check that the stack pointer is still in the same place. When you are convinced that everything is all right, continue to the next version.

VERSION 9: ZERO AND FILL ROUTINES

In version 4, we added a routine that could be used to change individual memory locations, one at a time. We will now add a routine which will allow us to fill a portion of memory with a constant value. A separate command for zeroing memory is also added for convenience, even though this operation could be performed with the FILL command.

Change the version number to 9, then alter the two command table entries that correspond to the F (fill) and Z (zero) commands.

```
DW      FILL    ;F, MEMORY
. . .
DW      ZERO    ;Z, MEMORY
```

Add the new code shown in Listing 6.9 to the end of the program.

Listing 6.9. Zero and fill routines.

```
                ; ZERO A PORTION OF MEMORY
                ; THE MONITOR AND STACK ARE
                ; PROTECTED
                ;
5A55 CD8659     ZERO:   CALL    RDHLDE  ;RANGE
5A58 0600               MVI     B,0
5A5A C3665A             JMP     FILL2
                ;
                ; FILL A PORTION OF MEMORY
                ;
5A5D CD7C5A     FILL:   CALL    HLDEBC  ;RANGE, BYTE
5A60 FEF7               CPI     APOS    ;APOSTROPHE?
5A62 CA755A             JZ      FILL4   ;YES ASCII
5A65 41                 MOV     B,C
5A66 7C         FILL2:  MOV     A,H     ;FILL BYTE
5A67 FE57               CPI     STACK SHR 8 ;TOO FAR?
5A69 D2D459             JNC     ERROR   ;YES
5A6C CD3E5A     FILL3:  CALL    CHEKM   ;PUT, CHECK
5A6F CD005A             CALL    TSTOP   ;DONE?
5A72 C3665A             JMP     FILL2   ;NEXT
                ;
```

```
5A75 CD2559    FILL4:  CALL    GETCH   ;ASCII CHAR
5A78 47                MOV     B,A
5A79 C36C5A            JMP     FILL3
               ;
               ; GET H,L D,E AND B,C
               ;
5A7C CD8A5A    HLDEBC: CALL    HLDECK  ;RANGE
5A7F DAD459            JC      ERROR   ;NO BYTE
5A82 E5                PUSH    H
5A83 CD9D59            CALL    READHL  ;3RD INPUT
5A86 44                MOV     B,H     ;MOVE TO
5A87 4D                MOV     C,L     ; B,C
5A88 E1                POP     H
5A89 C9                RET
               ;
               ; GET 2 ADDRESSES, CHECK THAT
               ; ADDITIONAL DATA IS INCLUDED
               ;
5A8A CD9159    HLDECK: CALL    HHLDE   ;2 ADDR
5A8D DAD459            JC      ERROR   ;THAT'S ALL
5A90 C38959            JMP     RDHLD2  ;CHECK
```

Assemble the program, load it into memory, and try it out. First, display a portion of memory.

```
>D4000 404F
```

Then, zero out a part of this region.

```
>Z4000 403F
```

Display the region again to be sure that the zero routine is working. Now fill a portion of the previously zeroed memory with A5 hex bytes.

```
>F4001 401E A5
```

Again, dump this region of memory to ensure that the fill routine is working.

```
>D4000 404F
```

Finally, check the ASCII fill command by filling with a $ symbol.

```
>F4020 402F '$
```

As with the load command, ASCII input is preceded by an apostrophe.

VERSION 10: A BLOCK-MOVE ROUTINE

The next routine to be added will allow us to move a block data from one memory location to another. This is actually a duplication routine, since the original memory block will remain unchanged. As each byte is moved to the new location, a check is made to ensure that it actually got there.

First, change the version number to 10. Then add the new lines to the end of the source program. Change the branch table entry corresponding to the command of M.

```
DW      MOVE    ;M, MEMORY
```

Insert the instructions given in Listing 6.10 to the end of the source program. Assemble the monitor and try it out.

Listing 6.10. A block-move routine.

```
                  ; MOVE A BLOCK OF MEMORY H,L-D,E TO B,C
                  ;
5A93 CD7C5A       MOVE:   CALL    HLDEBC  ;3 ADDR
5A96 CDA05A       MOVDN:  CALL    MOVIN   ;MOVE/CHECK
5A99 CD005A               CALL    TSTOP   ;DONE?
5A9C 03                   INX     B       ;NO
5A9D C3965A               JMP     MOVDN
                  ;
5AA0 7E           MOVIN:  MOV     A,M     ;BYTE
5AA1 02                   STAX    B       ;NEW LOCATION
5AA2 0A                   LDAX    B       ;CHECK
5AA3 BE                   CMP     M       ;IS IT THERE?
5AA4 C8                   RZ              ;YES
5AA5 60                   MOV     H,B     ;ERROR
5AA6 69                   MOV     L,C     ;INTO H,L
5AA7 C3425A               JMP     ERRP    ;SHOW BAD
```

The move command requires three addresses. These are the start and stop address of the source block and the start address of the destination block. For example,

```
>M5800 58FF 4000
```

will move the first page of the monitor (5800 to 58FF hex) down to the address range 4000 to 40FF hex.

The move routine is designed to move data downward. Thus the first byte of a block can be deleted by moving the remainder of the program downward by one byte. The command

```
>M103 1000 100
```

moves the memory block in the address range 103 through 1000 down three bytes to the memory range 100 through FFD hex. On the other hand, a block move in the upward direction must be done carefully.

If the new block does not overlap the old, then there is no problem. But if there is an overlap, then the upward move will destroy some of the data. One possible solution to this problem is to first move the block downward until it is clear of the new upper block. Then move the block up to the desired location. Another possibility is to move the upper half of the block first, then move the lower half.

The best solution is to have a more sophisticated move routine. This routine should first determine whether the move is to be upward or downward. If the movement is downward, then the move commences with the lower part of the block (as with the present program). But if the move is upward, then it should begin with the upper end of the block. The memory pointers should now move downward in memory. With this approach, the original data will be unaltered. This additional feature is more easily coded with the Z-80 block-move routines than with the 8080 instructions.

We have not incorporated the upward-move feature at this time. In developing a system monitor, there must be a tradeoff between features and space. A minimum of idiot-proofing is necessary. But, if we want to have a monitor that will fit into 1K bytes of ROM, we will have to make some compromises.

Notice that this move routine moves a byte from the source location into the destination location. It then reads the byte back from the new location to see that it actually got there. If an attempt is made to move data into read-only memory, protected memory, or defective memory, the process will be terminated. The address of the location will be printed following the letter B (for "bad").

If we want to retain this memory-checking feature, we will not be able to use the Z-80 block-move routines. The problem is that the Z-80 routines perform an automatic pointer increment after each byte is moved. If a memory check is desired, then the destination pointer will have to be backed up after each byte is moved. This will allow the newly moved byte to be checked. Finally, the pointer will have to be incremented again.

VERSION 11: A SEARCH ROUTINE

Sometimes it is necessary to find a particular data byte or address in memory. Or perhaps all occurrences of a data byte or an address within a memory block are needed. For version 11, we will add a hex search routine. Change the version number and the branch table entry for the letter S.

```
DW        SEARCH  ;S, MEMORY
```

Add the new instructions as given in Listing 6.11.

Listing 6.11. Search for 1 or 2 bytes.

```
                    ; SEARCH FOR 1 OR 2 BYTES OVER THE
                    ; RANGE H,L D,E. BYTES ARE IN B,C
                    ; B HAS CARRIAGE RETURN IF ONLY ONE BYTE
                    ; PUT SPACE BTWEEN BYTES IF TWO
                    ; FORMAT: START STOP BYTE1 BYTE2
                    ;
5AAA CD7C5A         SEARCH: CALL    HLDEBC   ;RANGE, 1ST BYTE
5AAD 060D           SEAR2:  MVI     B,CR     ;SET FOR 1 BYTE
```

```
5AAF DAB85A          JC      SEAR3    ;ONLY ONE
5AB2 E5              PUSH    H
5AB3 CD9D59          CALL    READHL   ;2ND BYTE
5AB6 45              MOV     B,L      ;INTO C
5AB7 E1              POP     H
5AB8 7E     SEAR3:   MOV     A,M      ;GET BYTE
5AB9 B9              CMP     C        ;MATCH?
5ABA C2CF5A          JNZ     SEAR4    ;NO
5ABD 23              INX     H        ;YES
5ABE 78              MOV     A,B      ;ONLY 1?
5ABF FE0D            CPI     CR
5AC1 CAC95A          JZ      SEAR5    ;YES
            ;
            ; FOUND FIRST MATCH, CHECK FOR SECOND
            ;
5AC4 7E              MOV     A,M      ;NEXT BYTE
5AC5 B8              CMP     B        ;MATCH?
5AC6 C2CF5A          JNZ     SEAR4    ;NO
            ;
5AC9 2B     SEAR5:   DCX     H        ;A MATCH
5ACA C5              PUSH    B
5ACB CDDC59          CALL    CRHL     ;SHOW ADDR
5ACE C1              POP     B
5ACF CD005A SEAR4:   CALL    TSTOP    ;DONE?
5AD2 C3B85A          JMP     SEAR3    ;NO
```

Our new feature will display the address of every occurrence of one or two chosen bytes. For example, the command

```
>S100 4FF OD
```

will print the address of each occurrence of a carriage return (OD hex) over the memory block 100 to 4FF hex. The alternate command

```
>SO FFFF 3E 10
```

includes two search bytes. This command will look for the byte 3E followed by the byte 10 over the entire 64K-byte memory range. These two bytes might represent the 8080 instruction MVI A,10 or perhaps the address 103E hex.

Notice that if two search bytes are given in the command, they must be separated by a space. If the command is incorrectly given without the space between the bytes, the search will only include the second byte. For example, the command

```
>SO FFFF 3E10
```

will be interpreted as a search for the byte 10 hex. This occurs because only the last two characters of the field are used.

VERSION 12: ASCII LOAD, SEARCH, AND DISPLAY

At this time, the monitor is hex oriented, but it is capable of limited ASCII operations. For example, the DUMP routine gives both the hex and the ASCII representation of the data. The load and fill commands will accept ASCII characters when preceded by an apostrophe. In version 12, we will add three new ASCII commands: ASCII load, ASCII dump, and ASCII search. A continuous series of ASCII characters (a string), including a carriage return, line feed, tab, and so on, can be entered directly into memory. A straight ASCII dump will render an ASCII portion of memory in its natural form. And we will be able to search the memory for one or two ASCII characters. If the command line begins with the letter A, a branch will occur to a second command processor. The letter following the A will cause a jump to the desired task of dump, load, or search.

Change the version number to 12 and the command table entry for the letter A.

```
DW      ASCII   ;A, DUMP, LOAD
```

Type the code from Listing 6.12; assemble the new version and start it up.

Listing 6.12. ASCII load, search, and display.

```
                  ; ASCII SUB-COMMAND PROCESSOR
                  ;
5AD5 CD2559       ASCII:  CALL    GETCH    ;NEXT CHAR
5AD8 FE44                 CPI     'D'      ;DISPLAY
5ADA CA045B               JZ      ADUMP
5ADD FE53                 CPI     'S'      ;SEARCH
5ADF CA2C5B               JZ      ASCS
5AE2 FE4C                 CPI     'L'      ;LOAD
5AE4 C2D459               JNZ     ERROR
                  ;
                  ; LOAD ASCII CHARACTERS INTO MEMORY
                  ; QUIT ON CONTROL-X
                  ;
5AE7 CD9D59               CALL    READHL   ;ADDRESS
5AEA CDDF59               CALL    OUTHL    ;PRINT IT
5AED CD1558       ALOD2:  CALL    INPUTT   ;NEXT CHAR
5AF0 CD2A58               CALL    OUTT     ;PRINT IT
5AF3 47                   MOV     B,A      ;SAVE
5AF4 CD3E5A               CALL    CHEKM    ;INTO MEMORY
5AF7 23                   INX     H        ;POINTER
5AF8 7D                   MOV     A,L
5AF9 E67F                 ANI     7FH      ;LINE END?
5AFB C2ED5A               JNZ     ALOD2    ;NO
5AFE CDDC59               CALL    CRHL     ;NEW LINE
5B01 C3ED5A               JMP     ALOD2
                  ;
                  ; DISPLAY MEMORY IN STRAIGHT ASCII.
                  ; KEEP CARRIAGE RETURN, LINE FEED, CHANGE
                  ; TAB TO SPACE, REMOVE OTHER CONTROL CHAR.
                  ;
```

```
5B04 CD8659    ADUMP:  CALL    RDHLDE   ;RANGE
5B07 7E        ADMP2:  MOV     A,M      ;GET BYTE
5B08 FE7F              CPI     DEL      ;HIGH BIT ON?
5B0A D2265B            JNC     ADMP4    ;YES
5B0D FE20              CPI     ' '      ;CONTROL?
5B0F D2235B            JNC     ADMP3    ;NO
5B12 FE0D              CPI     CR       ;CARR RET?
5B14 CA235B            JZ      ADMP3    ;YES, OK
5B17 FE0A              CPI     LF       ;LINE FEED?
5B19 CA235B            JZ      ADMP3    ;YES, OK
5B1C FE09              CPI     TAB
5B1E C2265B            JNZ     ADMP4    ;SKIP OTHER
5B21 3E20              MVI     A,' '    ;SPACE FOR TAB
5B23 CD2A58    ADMP3:  CALL    OUTT     ;SEND
5B26 CD005A    ADMP4:  CALL    TSTOP    ;DONE?
5B29 C3075B            JMP     ADMP2    ;NO
               ;
               ; SEARCH FOR 1 OR 2 ASCII CHARACTERS
               ; NO SPACE BETWEEN ASCII CHARS
               ; FORMAT: START STOP 1 OR 2 ASCII CHAR
               ;
5B2C CD8659    ASCS:   CALL    RDHLDE   ;RANGE
5B2F CD2559            CALL    GETCH    ;FIRST CHAR
5B32 4F                MOV     C,A
5B33 CD2559            CALL    GETCH    ;2ND OR CARR RET
5B36 DAAD5A            JC      SEAR2    ;ONLY ONE CHAR
5B39 47                MOV     B,A      ;2ND
5B3A C3B85A            JMP     SEAR3
```

Dump a section of memory with the regular hex dump command. Then enter a line of ASCII characters using the new ASCII load command.

```
>AL4000  <carriage return>
4000 This is a test of the new <cr><lf>
ASCII load routine. <cr><lf>
```

All of these characters will be deposited directly into memory, including the carriage returns and line feeds. Type a control-X to abort the task. Inspect the new addition first with the hex dump

```
>D4000 404F
```

then inspect it with the new ASCII dump:

```
>AD4000 404F
```

Notice the difference. The ASCII dump renders the data as it was originally typed.

A carriage-return line-feed pair will cause a real carriage-return line-feed pair to be sent to the console. Tab characters are not expanded but are rendered as blanks (in line with our goal of reducing the monitor size). All other control characters are ignored.

VERSION 13: INPUT AND OUTPUT TO ANY PORT

The load routines added in versions 4 and 12 allow us to change individual memory locations. And the dump routines added in versions 2 and 12 allow us to inspect individual memory locations. For version 13, we will add a routine to read any I/O port and another to send a byte to any I/O port. This feature will allow us to initialize and test I/O ports.

The 8080 and Z-80 microprocessors can address 256 separate, 8-bit input/output ports. These ports are used for communicating with the console, list, and tape devices. In addition, if there is a front panel, the switches are usually assigned to a separate data port. Also, some disk-controller boards use several I/O ports for communication with the CPU.

It is more difficult to implement these I/O features on an 8080 CPU than on a Z-80. The reason is that the 8080 I/O instructions require the port address to be placed in memory immediately following the IN or OUT command

```
DB 10     IN     10H
D3 11     OUT    11H
```

By comparison, the Z-80 can execute I/O instructions with the device address located in the C register. Nevertheless, we will implement the input and output instructions, at this time, using only 8080 code.

The plan is to write the IN or OUT instruction in memory, write the port number in the next byte, then write a RET instruction in the third position. A call to the address of PORTN will then produce the desired effect. The routine that writes these bytes in the stack area is called PUTIO. Since we are developing a monitor that can be placed in ROM, we will have to perform the actual I/O instructions outside of the regular monitor code area. Three bytes of memory just above the stack were previously set aside for this purpose. They start with the address PORTN.

A fourth routine is also needed. Subroutine BITS is used to convert the binary data read from the selected port into ASCII-coded binary characters. An IN command then prints on the console the port data in both hex and binary. For example, the command

```
>IFF
```

will give the front-panel switch setting in both hex and binary notation.

```
F8 11110000
```

The BITS routine can be coded more efficiently if a Z-80 CPU is available. This is because the Z-80 can shift data in the general-purpose registers, as well as in the accumulator. This is discussed in Chapter 7.

Change the version number to 13 and alter the branch table entries for the letters I and O.

```
DW        IPORT    ;I, PORT INPUT
. . .
DW        OPORT    ;O, PORT OUTPUT
```

Add the new routines to the end of the source code. Assemble version 13 and try it out.

Listing 6.13. Input and output to any port.

```
                    ; INPUT FROM ANY PORT (8080 VERSION)
                    ;
5B3D CD9D59         IPORT:  CALL    READHL   ;PORT
5B40 4D                     MOV     C,L      ;PORT TO C
5B41 3EDB                   MVI     A,INC    ;IN CODE
5B43 CD635B                 CALL    PUTIO    ;SETUP INPUT
5B46 6F                     MOV     L,A
5B47 CDE359                 CALL    OUTLL    ;HEX VALUE
                    ;
                    ; PRINT L REGISTER IN BINARY (8080 VER)
                    ;
5B4A 0608           BITS:   MVI     B,8      ;8 BITS
5B4C 7D             BIT2:   MOV     A,L
5B4D 87                     ADD     A        ;SHIFT LEFT
5B4E 6F                     MOV     L,A
5B4F 3E18                   MVI     A,'0'/2  ;HALF OF 0
5B51 8F                     ADC     A        ;DOUBLE+CARRY
5B52 CD2A58                 CALL    OUTT     ;PRINT BIT
5B55 05                     DCR     B
5B56 C24C5B                 JNZ     BIT2     ;8 TIMES
5B59 C9                     RET
                    ;
                    ; OUTPUT BYTE FROM PORT (8080 VERSION)
                    ; FORMAT IS: O,PORT,BYTE
                    ;
5B5A CD9D59         OPORT:  CALL    READHL   ;PORT
5B5D 4D                     MOV     C,L
5B5E CD9D59                 CALL    READHL   ;DATA
5B61 3ED3                   MVI     A,OUTC   ;OUT OPCODE
                    ;
                    ; EMULATE Z80 INP AND OUTP FOR 8080
                    ;
5B63 32A057         PUTIO:  STA     PORTN    ;IN OR OUT CODE
5B66 79                     MOV     A,C      ;PORT NUMBER
5B67 32A157                 STA     PORTN+1
5B6A 3EC9                   MVI     A,RETC   ;RET OPCODE
5B6C 32A257                 STA     PORTN+2
5B6F 7D                     MOV     A,L      ;OUTPUT BYTE
5B70 C3A057                 JMP     PORTN    ;EXECUTE
```

If you have a set of front panel switches, give the command

```
>IFF
```

and see if the bit pattern matches the actual switch setting. Next, try to ring your console bell by sending a binary 7.

```
>O11 7
```

The value of 11 should be changed to your console data port address if it is different.

Modern serial and parallel ports need to be initialized before use. These initialization routines could be placed in the monitor cold-start routines. Initialization can also be performed with the new monitor output command. A Motorola 6850 serial port can be initialized for one stop bit with the two commands

```
>O10 3          <reset>
>O10 15         <set>
```

where 10 is the address of the status/control port.

VERSION 14: HEXADECIMAL ARITHMETIC

A routine for obtaining the sum and difference of two hexadecimal numbers will now be added. Change the version number to 14. Change the branch table corresponding to the entry H.

```
DW        HMATH    ;H, HEX MATH
```

Place the remaining new lines at the end as usual.

```
Listing 6.14. Hecadecimal addition and subtraction.

                  ; HEXADECIMAL MATH, SUM AND DIFFERENCE
                  ;
5B73 CD9159       HMATH:  CALL    HHLDE    ;TWO NUMBERS
5B76 E5                   PUSH    H        ;SAVE H,L
5B77 19                   DAD     D        ;SUM
5B78 CDDF59               CALL    OUTHL    ;PRINT IT
5B7B E1                   POP     H
5B7C 7D                   MOV     A,L
5B7D 93                   SUB     E        ;LOW BYTES
5B7E 6F                   MOV     L,A
5B7F 7C                   MOV     A,H
5B80 9A                   SBB     D
5B81 67                   MOV     H,A      ;HIGH BYTES
5B82 C3DF59               JMP     OUTHL    ;DIFFERENCE
```

The new feature is executed by typing the letter H and the hex numbers. The response is the sum and the difference.

```
>H8000 4000
C000 4000
```

VERSION 15: MEMORY-TEST PROGRAM

Back in version 6, we installed an automatic memory-size routine. This addition performs a memory check of sorts by testing the first byte of each page. In version 15, we will add a more complete memory-test program. Change the version number and the branch table entry for the letter J (justification):

```
DW       JUST    ;J, MEMORY TEST
```

Then, type in the new lines as shown in Listing 6.15.

Listing 6.15. A memory-test program.

```
                   ;
                   ; MEMORY TEST
                   ; THAT DOESN'T ALTER CURRENT BYTE
                   ; INPUT RANGE OF ADDRESSES, ABORT WITH ^X
                   ;
5B85 CD8659        JUST:   CALL    RDHLDE  ;RANGE
5B88 E5                    PUSH    H       ;SAVE START ADDR
5B89 7E            JUST2:  MOV     A,M     ;GET BYTE
5B8A 2F                    CMA             ;COMPLEMENT IT
5B8B 77                    MOV     M,A     ;PUT IT BACK
5B8C BE                    CMP     M       ;DID IT GO?
5B8D C2A55B                JNZ     JERR    ;NO
5B90 2F                    CMA             ;ORIGINAL BYTE
5B91 77                    MOV     M,A     ;PUT IT BACK
5B92 7D            JUST3:  MOV     A,L     ;PASS
5B93 93                    SUB     E       ; COMPLETED?
5B94 7C                    MOV     A,H
5B95 9A                    SBB     D
5B96 23                    INX     H
5B97 DA895B                JC      JUST2   ;NO
                   ;
                   ; AFTER EACH PASS,
                   ; SEE IF ABORT WANTED
                   ;
5B9A CD2558                CALL    INSTAT  ;INPUT?
5B9D C41558                CNZ     INPUTT  ;YES, GET IT
5BA0 E1                    POP     H       ;START ADDR
5BA1 E5                    PUSH    H       ;SAVE AGAIN
5BA2 C3895B                JMP     JUST2   ;NEXT PASS
                   ;
                   ; FOUND MEMORY ERROR, PRINT POINTER AND
                   ; BIT MAP: 0=GOOD, 1=BAD BIT
                   ;
5BA5 F5            JERR:   PUSH    PSW     ;SAVE COMPLEMENT
5BA6 CDDC59                CALL    CRHL    ;PRINT POINTER
5BA9 F1                    POP     PSW
5BAA AE                    XRA     M       ;SET BAD BITS
5BAB E5                    PUSH    H       ;SAVE POINTER
5BAC 6F                    MOV     L,A     ;BIT MAP TO L
5BAD CD4A5B                CALL    BITS    ;PRINT BINARY
5BB0 E1                    POP     H
5BB1 C3925B                JMP     JUST3   ;CONTINUE
```

Assemble the program, load it into memory, and try it out. The memory range from zero to 58FF hex is tested with the command

```
>JO 5800
```

This is a continuing test. The given range is tested over and over until aborted with a control-X command. This memory-test program is not very sophisticated. The routine will not find unusual problems in flakey, dynamic memories. It will, however, locate those regions with no memory, protected memory, and grossly defective memory. The address of each bad location is printed in hex, then the bit pattern follows. ASCII ones are shown for the bad bits and ASCII zeros are given for the good bits.

The test program gets the original memory byte, complements it, and puts it back. It then complements it a second time and restores the original byte. Thus, the original memory is left intact. The only caution here is that the stack area should not be tested.

Much more sophisticated memory test programs are needed for difficult memory errors. Of course, such programs will require a lot of memory, and so would not fit into a compact system monitor. One feature of such a program is to provide a delay between the time the test byte is placed into memory and the time that the byte is checked. One disadvantage of a more powerful memory-test program is that it does not protect the original memory contents.

VERSION 16: REPLACE ONE BYTE WITH ANOTHER

In version 11 we added a memory-search routine. This feature gives us the ability to find every occurrence of a particular byte. A companion feature added in version 16 allows us to change every occurrence of a particular byte to a different byte. Change the version number and the branch table corresponding to the letter R.

```
DW        REPL     ;R, REPLACE
```

Add the new lines shown in listing 6.16 to the end of the program.

Listing 6.16. Replace one hex byte with another.

```
                    ; REPLACE HEX BYTE WITH ANOTHER
                    ; OVER GIVEN RANGE
                    ; FORMAT IS: START, STOP, ORIG, NEW
                    ;
5BB4 CD7C5A         REPL:   CALL    HLDEBC  ;RANGE, 1ST BYTE
5BB7 DAD459                 JC      ERROR   ;NO 2ND
5BBA 41                     MOV     B,C     ;1ST TO B
5BBB E5                     PUSH    H
```

```
5BBC CD9D59           CALL     READHL    ;2ND BYTE
5BBF 4D               MOV      C,L       ;INTO C
5BC0 E1               POP      H
5BC1 7E      REPL2:   MOV      A,M       ;FETCH BYTE
5BC2 B8               CMP      B         ;A MATCH?
5BC3 C2CC5B           JNZ      REPL3     ;NO
5BC6 71               MOV      M,C       ;SUBSTITUTE
5BC7 79               MOV      A,C
5BC8 BE               CMP      M         ;SAME?
5BC9 C2435A           JNZ      ERRB      ;NO, BAD
5BCC CD005A  REPL3:   CALL     TSTOP     ;DONE?
5BCF C3C15B           JMP      REPL2
```

Assemble version 16 and try it out. Move three lines of the monitor's code to a lower place using the M command.

```
>M5800 582F 4000
```

Dump these three lines of memory with the D command.

```
>D4000 402F
```

Change every occurrence of the byte C3 found in those lines to a 40 hex using the command

```
>R4000 402F C3 40
```

Notice that a space must separate the two bytes C3 and 40. Now, dump this portion of memory with the command

```
>D4000 402F
```

The new byte is an ASCII "at" sign (@), therefore it will show up clearly on the ASCII portion of the dump.

The replace routine can be useful for relocating a short executable program. Suppose that a routine is programmed for execution at 3000 hex. It can be moved to 4000 hex with the block-move command

```
>M3000 3FFF 4000
```

However, the program will not run at the new location if there are absolute jumps present. The high byte of each jump address will have to be changed from 30 to 40 in this case. The search routine can be used to find all occurrences of 30 hex in the program.

```
>S4000 4FFF 30
```

Then the replace command can be given to convert each 30 hex into a 40 hex.

```
>R4000 4FFF 30 40
```

Another use for the replace command is to convert an assembly language source file from one format to another. For example, the CP/M format requires a line feed to follow a carriage return. But another assembler may generate lines in which only the carriage return is placed at the end of each line. In this case, the original file can be loaded into memory. Then, all of the carriage returns (OD hex) can be replaced with an ASCII character such as a # symbol (23 hex).

```
>R100 38FF OD 23
```

After the file is altered with the monitor, it can be saved on a disk. The final step can be performed with the system editor. The global replace command of this editor can be used to replace every occurrence of the # sign with a carriage-return/line-feed pair. With the Word-Master editor, the command would be

```
*MR#$^N$0TT
```

The first step required the monitor because the system editors cannot be directed to globally change a carriage return to something else. The carriage-return/line-feed pair must be treated as a unit.

VERSION 17: COMPARE TWO BLOCKS OF MEMORY

This last addition to our system monitor will fill out the size to just under 1K bytes. The new routine will allow us to compare two blocks of memory. If discrepancies are found, the address and the contents of the appropriate location in both blocks will be shown. Change the version number and the branch table corresponding to the letter R.

```
DW      VERM    ;V
```

Add the new lines shown in Listing 6.17.

Listing 6.17. Compare two blocks of memory.

```
                    ; GIVE RANGE OF 1ST BLOCK
                    ; AND START OF SECOND
                    ;
5BD2 CD7C5A         VERM:   CALL    HLDEBC  ;3 ADDRESSES
5BD5 0A             VERM2:  LDAX    B       ;FETCH BYTE
5BD6 BE                     CMP     M       ;SAME AS OTHER?
5BD7 CAF35B                 JZ      VERM3   ;YES
5BDA E5                     PUSH    H       ;DIFFERENT
5BDB C5                     PUSH    B
5BDC CDDC59                 CALL    CRHL    ;PRINT 1ST POINTER
5BDF 4E                     MOV     C,M     ;FIRST BYTE
5BE0 CDE459                 CALL    OUTHEX  ;PRINT IT
5BE3 3E3A                   MVI     A,':'
```

```
5BE5 CD2A58           CALL   OUTT
5BE8 E1               POP    H        ;B,C TO H,L
5BE9 CDDF59           CALL   OUTHL    ;SECOND POINTER
5BEC 4E               MOV    C,M      ;2ND BYTE
5BED CDEC59           CALL   OUTHX    ;PRINT IT
5BF0 4D               MOV    C,L      ;RESTORE C
5BF1 44               MOV    B,H      ;AND B
5BF2 E1               POP    H        ;AND H,L
5BF3 CD005A   VERM3:  CALL   TSTOP    ;DONE?
5BF6 03               INX    B        ;2ND POINTER
5BF7 C33D55B          JMP    VERM2
```

The symbol table should now look like this.

```
5B07 ADMP2     5B23 ADMP3     5B26 ADMP4     5B04 ADUMP
5AED ALOD2     00F7 APOS      5AD5 ASCII     5B2C ASCS
0008 BACKUP    5B4C BIT2      5B4A BITS      5A09 CALLS
0011 CDATA     0011 CDATAO    5A3E CHEKM     5809 CIN
5858 COLD      5806 COUT      000D CR        59DC CRHL
590F CRLF      0010 CSTAT     0010 CSTATO    0008 CTRH
0011 CTRQ      0013 CTRS      0018 CTRX      007F DEL
5945 DUMP      5948 DUMP2     594B DUMP3     595E DUMP4
5967 DUMP5     5A45 ERR2      5A43 ERRB      59D4 ERROR
5A42 ERRP      001B ESC       5A5D FILL      5A66 FILL2
5A6C FILL3     5A75 FILL4     580F GCHAR     5939 GETC4
5925 GETCH     5A08 GO        59F5 HEX1      5991 HHLDE
5A7C HLDEBC    5A8A HLDECK    5B73 HMATH     57A5 IBUFC
57A6 IBUFF     57A3 IBUFP     00DB INC       580C INLN
0001 INMSK     58D5 INPL2     58F1 INPL3     5919 INPLB
5901 INPLC     58FB INPLE     58DD INPLI     58D0 INPLN
581B INPUT2    5815 INPUTT    5825 INSTAT    5B3D IPORT
5BA5 JERR      5B85 JUST      5B89 JUST2     5B92 JUST3
000A LF        5A0D LOAD      5A10 LOAD2     5A33 LOAD3
5A30 LOAD4     5A37 LOAD6     5A96 MOVDN     5A93 MOVE
5AA0 MOVIN     5878 MSIZE     59C4 NIB       586A NPAGE
0002 OMSK      5B5A OPORT     5800 ORGIN     582B OUT2
5839 OUT3      5844 OUT4      00D3 OUTC      5812 OUTH
59E4 OUTHEX    59DF OUTHL     59EC OUTHX     59E3 OUTLL
59E7 OUTSP     582A OUTT      5981 PASC2     5983 PASC3
5976 PASCI     57A0 PORTN     5B63 PUTIO     59A2 RDHL2
59B7 RDHL4     59C1 RDHL5     5989 RDHLD2    5986 RDHLDE
599D READHL    5A4E REGS      5BB4 REPL      5BC1 REPL2
5BCC REPL3     5803 RESTRT    00C9 RETC      5AAD SEAR2
5AB8 SEAR3     5ACF SEAR4     5AC9 SEAR5     5AAA SEARCH
593B SENDM     584F SIGNON    57A0 STACK     5800 START
0009 TAB       589C TABLE     0018 TOP       5A00 TSTOP
5BD2 VERM      5BD5 VERM2     5BF3 VERM3     3731 VERS
5861 WARM      5A55 ZERO
```

Try the new addition by first moving a copy of the monitor down to a lower memory location.

```
>M5800 5BFF 4800
```

Then verify that the two copies are the same.

```
>V5800 5BFF 4800
```

Of course this step is not necessary, since there is a verification step included in the block-move routine. Change one byte in the new location so that there will be a difference.

```
>L4820
4820 . XX 0        <zero location 4820>
. . .      ↑X      <quit>
```

Then, give the verification command again.

```
>V5800 5BFF 4800
```

Because you changed one byte of the copy, there should be an indication of error.

This compare routine completes the 1K 8080 system monitor. We have incorporated many useful features into a minimum of space. We have carefully distinguished program code from data code so that the monitor can be placed into ROM or PROM.

AUTOMATIC EXECUTION OF THE MONITOR

If you program the monitor into ROM, it will be ready to use each time the computer is turned on. On the other hand, you may want to copy it from disk into memory each time it is needed. We have been loading the monitor with the system debugger each time it is needed. But it is easier to include a short loader program at the beginning of the monitor. Then you can execute the monitor just by typing its name.

A suitable loader program is given in Listing 6.18. Type the program into your editor. There are two locations that need to be matched to your monitor; these are the addresses of START and FINAL. START must correspond to the first address of your monitor. The address FINAL is the last address of the monitor.

Listing 6.18. Loader program to move the monitor.

```
5800 =          START   EQU     5800H   ;MONITOR START
5BFF =          FINAL   EQU     5BFFH   ;MONITOR END
A920 =          OFFST   EQU     120-START ;LOAD OFFSET
                ;
0100            ORG     100H            ;START HERE
                ;
0100 210058             LXI     H,START ;NEW START
0103 012001             LXI     B,120H  ;OLD START
0106 11FF5B             LXI     D,FINAL
                ;
0109 0A         LOOP:   LDAX    B       ;GET A BYTE
010A 77                 MOV     M,A     ;TO NEW PLACE
010B BE                 CMP     M       ;DID IT GO?
```

```
010C C20000          JNZ     0        ;NO, QUIT
010F 23              INX     H        ;INCREMENT
0110 03              INX     B        ; POINTERS
0111 7B              MOV     A,E      ;DONE?
0112 95              SUB     L
0113 7A              MOV     A,D
0114 9C              SBB     H
0115 D20901          JNC     LOOP     ;KEEP GOING
0118 C30058          JMP     START    ;DONE
              ;
011B                 END
```

Assemble the loader program, then load it into memory with the debugger SID or DDT.

```
A>DDT  MOVE.HEX
```

Next, place a copy of the monitor into memory starting at address 120 hex. If the monitor is already in memory, a copy can be generated with the monitor itself. DDT or SID can also be used for this task. The command is

```
M5800 5BFF 120
```

If the monitor resides on disk as a hex file, it can be loaded with the debugger after you calculate the offset. The offset is necessary since hex files are normally loaded at the operating address, but we want to put it somewhere else.

The required offset should be given in the assembly listing of the loader program as the value of the equate OFFST. If your assembler doesn't print such values, then use the debugger to calculate the value.

```
H120   5800   <starting value of monitor>
5920   A920
<sum>  <difference>
```

Give the commands

```
IMON17.HEX
R<offset>
```

so that the monitor will be loaded starting at address 120 hex.

Return to the CP/M system

```
G0      <go to zero>
```

and save the combination

```
A>SAVE  5  MONITOR.COM
```

From now on, all you have to do is type the command

```
A>MONITOR
```

and the monitor will automatically start up.

What actually happens is that the combination of the monitor and the loader program is first copied into memory at 100 hex. The move program relocates the monitor from address 120 hex to its proper place. Then control is transferred to the monitor. As each byte is moved to the new location, it is checked to see that it actually got there. If not, the process is terminated and control returns to CP/M.

This short loader can be placed on the front of any program that must be relocated. Only the first two instructions may have to be changed to reflect the proper starting and ending addresses.

In the next chapter, we will convert our monitor to Z-80 code. The Z-80 version will be smaller so that we can incorporate a few additional features and still be able to fit the program into 1K of ROM. The features in the next chapter can be incorporated in the 8080 version, but they will take so much space that the monitor will no longer fit into 1K bytes.

CHAPTER SEVEN

A Z-80 System Monitor

The system monitor developed in Chapter 6 contains many features. Since the size is less than 1,024 bytes, it will easily fit into a 1K PROM, such as the 2708 EPROM. It can then be ready for use as soon as the computer is turned on. But, in this case, it may be necessary to include a routine to initialize the peripheral ports, such as those that handle the console and printer. In addition, you might want to send output to a printer as well as to the video console. If these two features are added to the monitor, the size will increase beyond 1K bytes and it will not fit into a single 1K PROM.

One way to add these new features without increasing the monitor's size is to remove some of the original routines. Another way, if you have a Z-80 CPU, is to convert some of the instructions to the more compact Z-80 equivalent operations. The latter approach will be followed in this chapter. Listing 7.1 gives the final version with all changes discussed in this chapter. The symbol table at the end can be used to find the routines of interest.

Listing 7.1 The Z-80 version of the system monitor.

```
                        TITLE   Z-80 SYSTEM MONITOR
                        ;
                        ; (Date goes here)
                        ;
                        ; FOUR SECTIONS HAVE BEEN REMOVED:
                        ;  VERS EQU       ...      (1 LINE)
                        ;  SIGNON:        ...      (4 LINES)
                        ;  LXI    D,SIGNON         (2 LINES)
                        ;  SENDM:         ...      (6 LINES)
                        ;
                        ; ONE SECTION HAS BEEN ADDED:
                        ;  LIST OUPUT ROUTINES
                        ;
0018                    TOP     EQU     24        ;MEMORY TOP, K BYTES
5800                    ORGIN   EQU     (TOP-2)*1024  ;PROGRAM START
                        ;
```

```
0000                ASEG                ;ABSOLUTE CODE
                    .Z80
                    ORG     ORGIN
                    ;
57A0                STACK   EQU     ORGIN-60H
0010                CSTAT   EQU     10H     ;CONSOLE STATUS
0011                CDATA   EQU     CSTAT+1 ;CONSOLE DATA
0001                INMSK   EQU     1       ;INPUT MASK
0002                OMSK    EQU     2       ;OUTPUT MASK
0012                LSTAT   EQU     12H     ;LIST STATUS (18)
0013                LDATA   EQU     LSTAT+1 ;LIST DATA   (18)
0002                LOMSK   EQU     2       ;OUTPUT MASK (18)
0004                NNULS   EQU     4       ;LIST NULLS  (18)
                    ;
57A0                PORTN   EQU     STACK   ;CONS=0,LIST=1
57A3                IBUFP   EQU     STACK+3 ;BUFFER POINTER
57A5                IBUFC   EQU     IBUFP+2 ;BUFFER COUNT
57A6                IBUFF   EQU     IBUFP+3 ;INPUT BUFFER
                    ;
0008                CTRH    EQU     8       ;^H BACKSPACE
0009                TAB     EQU     9       ;^I
0010                CTRP    EQU     16      ;^P (18)
0011                CTRQ    EQU     17      ;^Q
0013                CTRS    EQU     19      ;^S
0018                CTRX    EQU     24      ;^X, ABORT
0008                BACKUP  EQU     CTRH    ;BACKUP CHAR
007F                DEL     EQU     127     ;RUBOUT
00F7                APOS    EQU     (39-'0') AND OFFH
000D                CR      EQU     13      ;CARRIAGE RET
000A                LF      EQU     10      ;LINE FEED
                    ;
                    START:
5800 C3 587E                JP      COLD    ;COLD START
5803 C3 5891        RESTRT: JP      WARM    ;WARM START
                    ;
                    ; VECTORS TO USEFUL ROUTINES
                    ;
5806 C3 5835        COUT:   JP      OUTT    ;OUTPUT CHAR
5809 C3 5815        CIN:    JP      INPUTT  ;INPUT CHAR
580C C3 58FD        INLN:   JP      INPLN   ;INPUT LINE
580F C3 5949        GCHAR:  JP      GETCH   ;GET CHAR
5812 C3 59F8        OUTH:   JP      OUTHX   ;BIN TO HEX
                    ;
                    ; CONSOLE INPUT ROUTINE
                    ; CHECK FOR CONTROL-P, LIST TOGGLE
                    ;
5815 CD 5827        INPUTT: CALL    INSTAT   ;CHECK STATUS
5818 28 FB                  JR      Z,INPUTT ;NOT READY
581A DB 11          INPUT2: IN      A,(CDATA) ;GET BYTE
581C E6 7F                  AND     DEL
581E FE 18                  CP      CTRX    ;ABORT?
5820 28 DE                  JR      Z,START ;YES
5822 FE 10                  CP      CTRP    ;^P?
5824 28 06                  JR      Z,SETLST ;LIST
5826 C9                     RET
                    ;
                    ; GET CONSOLE-INPUT STATUS
                    ;
```

```
5827 DB 10      INSTAT: IN      A,(CSTAT)
5829 E6 01              AND     INMSK
582B C9                 RET
                ;
                ; TOGGLE LIST OUTPUT WITH CONTROL-P
                ;
582C 3A 57A0    SETLST: LD      A,(PORTN) ;CHECK FLAG
582F 2F                 CPL               ;INVERT
5830 32 57A0            LD      (PORTN),A ;SAVE
5833 18 E0              JR      INPUTT    ;NEXT BYTE
                ;
                ; CONSOLE OUTPUT ROUTINE
                ;
5835 F5         OUTT:   PUSH    AF
5836 3A 57A0            LD      A,(PORTN) ;WHERE?
5839 B7                 OR      A       ;ZERO?
583A 20 1F              JR      NZ,LOUT ;LIST OUTPUT
583C CD 5827    OUT2:   CALL    INSTAT  ;INPUT?
583F 28 10              JR      Z,OUT4  ;NO
5841 CD 581A            CALL    INPUT2  ;GET INPUT
5844 FE 13              CP      CTRS    ;FREEZE?
5846 20 F4              JR      NZ,OUT2 ;NO
5848 CD 5815    OUT3:   CALL    INPUTT  ;INPUT?
584B FE 11              CP      CTRQ    ;RESUME?
584D 20 F9              JR      NZ,OUT3 ;NO
584F 18 EB              JR      OUT2
                ;
5851 DB 10      OUT4:   IN      A,(CSTAT) ;GET STATUS
5853 E6 02              AND     OMSK
5855 28 E5              JR      Z,OUT2    ;NOT READY
5857 F1                 POP     AF
5858 D3 11              OUT     (CDATA),A ;SEND DATA
585A C9                 RET
                ;
                ; LIST OUTPUT ROUTINE
                ; SEND TO CONSOLE TOO
                ;
585B CD 5827    LOUT:   CALL    INSTAT    ;INPUT?
585E C4 581A            CALL    NZ,INPUT2 ;YES, GET IT
                ;
5861 DB 12              IN      A,(LSTAT) ;CHECK STATUS
5863 E6 02              AND     OMSK
5865 28 F4              JR      Z,LOUT    ;NOT READY
5867 F1                 POP     AF
5868 D3 13              OUT     (LDATA),A ;SEND DATA
586A D3 11              OUT     (CDATA),A ;CONSOLE TOO
586C E6 7F              AND     7FH       ;MASK PARITY
                ;
                ; ADD TIME DELAY AFTER CARRIAGE RETURN
                ;
586E FE 0D              CP      CR      ;CARRIAGE RET?
5870 C0                 RET     NZ      ;NO
5871 D5                 PUSH    DE      ;USE D,E
5872 16 78              LD      D,30 * NNULS
5874 1E FA      OUTCR:  LD      E,250
5876 1D         OUTCR2: DEC     E
5877 20 FD              JR      NZ,OUTCR2 ;INNER LOOP
5879 15                 DEC     D
```

```
587A 20 F8              JR      NZ,OUTCR   ;OUTER LOOP
587C D1                 POP     DE         ;RESTORE
587D C9                 RET
                ;
                ; CONTINUATION OF COLD START
                ;
587E 31 57A0    COLD:   LD      SP,STACK
                ;
                ; INITIALIZE I/O PORTS
                ;
5881 3E 03              LD      A,3
5883 D3 10              OUT     (CSTAT),A ;RESET
5885 D3 12              OUT     (LSTAT),A
5887 3E 15              LD      A,15H
5889 D3 10              OUT     (CSTAT),A ;SET
588B D3 12              OUT     (LSTAT),A
588D AF                 XOR     A     ;GET A ZERO
588E 32 57A0            LD      (PORTN),A ;RESET
                ;
                ; WARM-START ENTRY
                ;
5891 21 5891    WARM:   LD      HL,WARM ;RET TO
5894 E5                 PUSH    HL      ; HERE
                ;
                ; FIND TOP OF USABLE MEMORY.
                ; CHECK FIRST BYTE OF EACH PAGE OF MEMORY
                ; STARTING AT ADDRESS ZERO.  STOP AT STACK
                ; OR MISSING/DEFECTIVE/PROTECTED MEMORY.
                ; DISPLAY HIGH BYTE OF MEMORY TOP.
                ;
5895 21 0000            LD      HL,0      ;PAGE ZERO
5898 06 57              LD      B,HIGH STACK ;STOP HERE
589A 7E         NPAGE:  LD      A,(HL)    ;GET BYTE
589B 2F                 CPL               ;COMPLEMENT
589C 77                 LD      (HL),A    ;PUT IT BACK
589D BE                 CP      (HL)      ;SAME?
589E 20 05              JR      NZ,MSIZE ;NO, MEM TOP
58A0 2F                 CPL               ;ORIG BYTE
58A1 77                 LD      (HL),A    ;RESTORE IT
58A2 24                 INC     H         ;NEXT PAGE
58A3 10 F5              DJNZ    NPAGE     ;KEEP GOING
58A5 4C         MSIZE:  LD      C,H       ;MEM TOP
58A6 CD 5935            CALL    CRLF      ;NEW LINE
58A9 CD 59F8            CALL    OUTHX     ;PRINT MEM SIZE
58AC CD 58FD            CALL    INPLN     ;CONSOLE LINE
58AF CD 5949            CALL    GETCH     ;FIRST CHAR
                ;
                ; MAIN COMMAND PROCESSOR
                ;
58B2 D6 41              SUB     'A'       ;CONVERT OFFSET
58B4 DA 59E0            JP      C,ERROR ; < A
58B7 FE 1A              CP      'Z'-'A'+1
58B9 D2 59E0            JP      NC,ERROR ; > Z
58BC 87                 ADD     A,A       ;DOUBLE
58BD 21 58C9            LD      HL,TABLE ;START
58C0 16 00              LD      D,0
58C2 5F                 LD      E,A       ;OFFSET
58C3 19                 ADD     HL,DE     ;ADD TO TABLE
```

```
58C4 5E                 LD      E,(HL)   ;LOW BYTE
58C5 23                 INC     HL
58C6 56                 LD      D,(HL)   ;HIGH BYTE
58C7 EB                 EX      DE,HL    ;INTO H,L
58C8 E9                 JP      (HL)     ;GO THERE
                ;
                ; COMMAND TABLE
                ;
58C9 5AD3       TABLE:  DW      ASCII    ;A, DUMP,LOAD
58CB 59E0               DW      ERROR    ;B
58CD 5A15               DW      CALLS    ;C, SUBROUTINE
58CF 595E               DW      DUMP     ;D, DUMP
58D1 59E0               DW      ERROR    ;E
58D3 5A64               DW      FILL     ;F, MEMORY
58D5 5A14               DW      GO       ;G, GO
58D7 5B51               DW      HMATH    ;H, HEX MATH
58D9 5B31               DW      IPORT    ;I, PORT INPUT
58DB 5B60               DW      JUST     ;J, MEMORY TEST
58DD 59E0               DW      ERROR    ;K
58DF 5A19               DW      LOAD     ;L, LOAD
58E1 5A97               DW      MOVE     ;M, MEMORY
58E3 59E0               DW      ERROR    ;N
58E5 5B47               DW      OPORT    ;O, PORT OUTPUT
58E7 59E0               DW      ERROR    ;P
58E9 59E0               DW      ERROR    ;Q
58EB 5B8C               DW      REPL     ;R, REPLACE
58ED 5AAD               DW      SEARCH   ;S, MEMORY
58EF 59E0               DW      ERROR    ;T
58F1 59E0               DW      ERROR    ;U
58F3 5BA8               DW      VERM     ;V, VERIFY MEM
58F5 59E0               DW      ERROR    ;W
58F7 5A56               DW      REGS     ;X, STK PNTR
58F9 59E0               DW      ERROR    ;Y
58FB 5A5D               DW      ZERO     ;Z, MEMORY
                ;
                ; INPUT A LINE FROM CONSOLE AND PUT IT
                ; INTO THE BUFFER. CARRIAGE RETURN ENDS
                ; THE LINE. RUBOUT OR ^H CORRECTS LAST
                ; LAST ENTRY. CONTROL-X RESTARTS LINE.
                ; OTHER CONTROL CHARACTERS ARE IGNORED
                ;
58FD 3E 3E      INPLN:  LD      A,'>'    ;PROMPT
58FF CD 5835            CALL    OUTT
5902 21 57A6    INPL2:  LD      HL,IBUFF ;BUFFER ADDR
5905 22 57A3            LD      (IBUFP),HL ;SAVE POINTER
5908 0E 00              LD      C,0      ;COUNT
590A CD 5815    INPLI:  CALL    INPUTT   ;CONSOLE CHAR
590D FE 20              CP      ' '      ;CONTROL?
590F 38 18              JR      C,INPLC  ;YES
5911 FE 7F              CP      DEL      ;DELETE
5913 28 2A              JR      Z,INPLB  ;YES
5915 FE 5B              CP      'Z'+1    ;UPPER CASE?
5917 38 02              JR      C,INPL3  ;YES
5919 E6 5F              AND     5FH      ;MAKE UPPER
591B 77         INPL3:  LD      (HL),A   ;INTO BUFFER
591C 3E 20              LD      A,32     ;BUFFER SIZE
591E B9                 CP      C        ;FULL?
591F 28 E9              JR      Z,INPLI  ;YES, LOOP
```

```
5921 7E                 LD      A,(HL)   ;GET CHAR
5922 23                 INC     HL       ;INCR POINTER
5923 0C                 INC     C        ;AND COUNT
5924 CD 5835    INPLE:  CALL    OUTT     ;SHOW CHAR
5927 18 E1              JR      INPLI    ;NEXT CHAR
                ;
                ; PROCESS CONTROL CHARACTER
                ;
5929 FE 08      INPLC:  CP      CTRH     ;^H?
592B 28 12              JR      Z,INPLB  ;YES
592D FE 0D              CP      CR       ;RETURN?
592F 20 D9              JR      NZ,INPLI ;NO, IGNORE
                ;
                ; END OF INPUT LINE
                ;
5931 79                 LD      A,C      ;COUNT
5932 32 57A5            LD      (IBUFC),A ;SAVE
                ;
                ; CARRIAGE-RETURN, LINE-FEED ROUTINE
                ;
5935 3E 0D      CRLF:   LD      A,CR
5937 CD 5835            CALL    OUTT     ;SEND CR
593A 3E 0A              LD      A,LF
593C C3 5835            JP      OUTT     ;SEND LF
                ;
                ; DELETE PRIOR CHARACTER IF ANY
                ;
593F 79         INPLB:  LD      A,C      ;CHAR COUNT
5940 B7                 OR      A        ;ZERO?
5941 28 C7              JR      Z,INPLI  ;YES
5943 2B                 DEC     HL       ;BACK POINTER
5944 0D                 DEC     C        ;AND COUNT
5945 3E 08              LD      A,BACKUP ;CHARACTER
5947 18 DB              JR      INPLE    ;SEND
                ;
                ; GET A CHARACTER FROM CONSOLE BUFFER
                ; SET CARRY IF EMPTY
                ;
5949 E5         GETCH:  PUSH    HL       ;SAVE REGS
594A 2A 57A3            LD      HL,(IBUFP) ;GET POINTER
594D 3A 57A5            LD      A,(IBUFC) ;AND COUNT
5950 D6 01              SUB     1        ;DECR WITH CARRY
5952 38 08              JR      C,GETC4  ;NO MORE CHAR
5954 32 57A5            LD      (IBUFC),A ;SAVE NEW COUNT
5957 7E                 LD      A,(HL)   ;GET CHARACTER
5958 23                 INC     HL       ;INCR POINTER
5959 22 57A3            LD      (IBUFP),HL ;AND SAVE
595C E1         GETC4:  POP     HL       ;RESTORE REGS
595D C9                 RET
                ;
                ; DUMP MEMORY IN HEXADECIMAL AND ASCII
                ;
595E CD 5999    DUMP:   CALL    RDHLDE   ;RANGE
5961 CD 59E8    DUMP2:  CALL    CRHL     ;NEW LINE
5964 4E         DUMP3:  LD      C,(HL)   ;GET BYTE
5965 CD 59F8            CALL    OUTHX    ;PRINT
5968 23                 INC     HL       ;POINTER
5969 7D                 LD      A,L
```

```
596A E6 0F              AND     0FH      ;LINE END?
596C 28 07              JR      Z,DUMP4  ;YES, ASCII
596E E6 03              AND     3        ;SPACE
5970 CC 59F3            CALL    Z,OUTSP  ; 4 BYTES
5973 18 EF              JR      DUMP3    ;NEXT HEX
5975 CD 59F3   DUMP4:   CALL    OUTSP
5978 D5                 PUSH    DE
5979 11 FFF0            LD      DE,-10H  ;RESET LINE
597C 19                 ADD     HL,DE
597D D1                 POP     DE
597E CD 598B   DUMP5:   CALL    PASCI    ;ASCII DUMP
5981 CD 5A0C            CALL    TSTOP    ;DONE?
5984 7D                 LD      A,L      ;NO
5985 E6 0F              AND     0FH      ;LINE END?
5987 20 F5              JR      NZ,DUMP5 ;NO
5989 18 D6              JR      DUMP2
                ;
                ; DISPLAY MEMORY BYTE IN ASCII IF
                ; POSSIBLE, OTHERWISE GIVE DECIMAL PNT
                ;
598B 7E        PASCI:   LD      A,(HL)   ;GET BYTE
598C FE 7F              CP      DEL      ;HIGH BIT ON?
598E 30 04              JR      NC,PASC2 ;YES
5990 FE 20              CP      ' '      ;CONTROL CHAR?
5992 30 02              JR      NC,PASC3 ;NO
5994 3E 2E     PASC2:   LD      A,'.'    ;CHANGE TO DOT
5996 C3 5835   PASC3:   JP      OUTT     ;SEND
                ;
                ; GET H,L AND D,E FROM CONSOLE
                ; CHECK THAT D,E IS LARGER
5999 CD 59A3   RDHLDE:  CALL    HHLDE
599C 7B        RDHLD2:  LD      A,E
599D 95                 SUB     L        ;E - L
599E 7A                 LD      A,D
599F 9C                 SBC     A,H      ;D - H
59A0 38 3E              JR      C,ERROR  ;H,L BIGGER
59A2 C9                 RET
                ;
                ; INPUT H,L AND D,E
                ;
59A3 CD 59AE   HHLDE:   CALL    READHL   ;H,L
59A6 38 38              JR      C,ERROR  ;ONLY 1 ADDR
59A8 EB                 EX      DE,HL    ;SAVE IN D,E
59A9 CD 59AE            CALL    READHL   ;D,E
59AC EB                 EX      DE,HL    ;PUT BACK
59AD C9                 RET
                ;
                ; INPUT H,L FROM CONSOLE
                ;
59AE D5        READHL:  PUSH    DE
59AF C5                 PUSH    BC       ;SAVE REGS
59B0 21 0000            LD      HL,0     ;CLEAR
59B3 CD 5949   RDHL2:   CALL    GETCH    ;GET CHAR
59B6 38 15              JR      C,RDHL5  ;LINE END
59B8 CD 59D0            CALL    NIB      ;TO BINARY
59BB 38 08              JR      C,RDHL4  ;NOT HEX
59BD 29                 ADD     HL,HL    ;SHIFT LEFT
59BE 29                 ADD     HL,HL    ; FOUR
```

```
59BF 29              ADD    HL,HL    ; BYTES
59C0 29              ADD    HL,HL
59C1 B5              OR     L        ;ADD NEW CHAR
59C2 6F              LD     L,A
59C3 18 EE           JR     RDHL2    ;NEXT
                ;
                ; CHECK FOR COMMA OR BLANK AT END
                ;
59C5 FE F7      RDHL4:  CP     APOS     ;APOSTROPHE
59C7 28 04              JR     Z,RDHL5  ;ASCII INPUT
59C9 FE F0              CP     (' '-'0') AND 0FFH
59CB 20 13              JR     NZ,ERROR ;NOT BLANK
59CD C1         RDHL5:  POP    BC
59CE D1                 POP    DE       ;RESTORE
59CF C9                 RET
                ;
                ; CONVERT ASCII CHARACTERS TO BINARY
                ;
59D0 D6 30      NIB:    SUB    '0'      ;ASCII BIAS
59D2 D8                 RET    C        ; < 0
59D3 FE 17              CP     'F'-'0'+1
59D5 3F                 CCF             ;INVERT
59D6 D8                 RET    C        ;ERROR, > F
59D7 FE 0A              CP     10
59D9 3F                 CCF             ;INVERT
59DA D0                 RET    NC       ;NUMBER 0-9
59DB D6 07              SUB    'A'-'9'-1
59DD FE 0A              CP     10       ;REMOVE :-
59DF C9                 RET             ;LETTER A-F
                ;
                ; PRINT ? ON IMPROPER INPUT
                ;
59E0 3E 3F      ERROR:  LD     A,'?'
59E2 CD 5835            CALL   OUTT
59E5 C3 5800            JP     START    ;TRY AGAIN
                ;
                ; START NEW LINE, GIVE ADDRESS
                ;
59E8 CD 5935    CRHL:   CALL   CRLF     ;NEW LINE
                ;
                ; PRINT H,L IN HEX
                ;
59EB 4C         OUTHL:  LD     C,H
59EC CD 59F8            CALL   OUTHX    ;H
59EF 4D         OUTLL:  LD     C,L
                ;
                ; OUTPUT HEX BYTE FROM C AND A SPACE
                ;
59F0 CD 59F8    OUTHEX: CALL   OUTHX
                ;
                ; OUTPUT A SPACE
                ;
59F3 3E 20      OUTSP:  LD     A,' '
59F5 C3 5835            JP     OUTT
                ;
                ; OUTPUT A HEX BYTE FROM C
                ; BINARY TO ASCII HEX CONVERSION
                ;
```

```
59F8 79          OUTHX:  LD      A,C
59F9 1F                  RRA              ;ROTATE
59FA 1F                  RRA              ; FOUR
59FB 1F                  RRA              ; BITS TO
59FC 1F                  RRA              ; RIGHT
59FD CD 5A01             CALL    HEX1     ;UPPER CHAR
5A00 79                  LD      A,C      ;LOWER CHAR
5A01 E6 0F       HEX1:   AND     0FH      ;TAKE 4 BITS
5A03 C6 90               ADD     A,90H
5A05 27                  DAA              ;DAA TRICK
5A06 CE 40               ADC     A,40H
5A08 27                  DAA
5A09 C3 5835             JP      OUTT
                 ;
                 ; CHECK FOR END, H,L MINUS D,E
                 ; INCREMENT H,L
                 ;
5A0C 23          TSTOP:  INC     HL
5A0D 7B                  LD      A,E
5A0E 95                  SUB     L        ; E - L
5A0F 7A                  LD      A,D
5A10 9C                  SBC     A,H      ; D - H
5A11 D0                  RET     NC       ;NOT DONE
5A12 E1                  POP     HL       ;RAISE STACK
5A13 C9                  RET
                 ;
                 ; ROUTINE TO GO ANYWHERE IN MEMORY
                 ; FOR CALL ENTRY, ADDRESS OF WARM
                 ; IS ON STACK, SO A SIMPLE RET
                 ; WILL RETURN TO THIS MONITOR
                 ;
5A14 E1          GO:     POP     HL       ;RAISE STACK
5A15 CD 59AE     CALLS:  CALL    READHL   ;GET ADDRESS
5A18 E9                  JP      (HL)     ;GO THERE
                 ;
                 ; LOAD HEX OR ASCII CHAR INTO MEMORY
                 ; FROM CONSOLE. CHECK TO SEE IF
                 ; THE DATA ACTUALLY GOT THERE
                 ; APOSTROPHE PRECEEDS ASCII CHAR
                 ; CARRIAGE RET  PASSES OVER LOCATION
                 ;
5A19 CD 59AE     LOAD:   CALL    READHL   ;ADDRESS
5A1C CD 59EB     LOAD2:  CALL    OUTHL    ;PRINT IT
5A1F CD 598B             CALL    PASCI    ;ASCII
5A22 CD 59F3             CALL    OUTSP
5A25 4E                  LD      C,(HL)   ;ORIG BYTE
5A26 CD 59F0             CALL    OUTHEX   ;HEX
5A29 E5                  PUSH    HL       ;SAVE PNTR
5A2A CD 5902             CALL    INPL2    ;INPUT
5A2D CD 59AE             CALL    READHL   ; BYTE
5A30 45                  LD      B,L      ; TO B
5A31 E1                  POP     HL
5A32 FE F7               CP      APOS
5A34 28 0A               JR      Z,LOAD6  ;ASCII INPUT
5A36 79                  LD      A,C      ;HOW MANY?
5A37 B7                  OR      A        ;NONE?
5A38 28 03               JR      Z,LOAD3  ;YES
```

```
5A3A CD 5A46     LOAD4:  CALL     CHEKM     ;INTO MEMORY
5A3D 23          LOAD3:  INC      HL        ;POINTER
5A3E 18 DC               JR       LOAD2
                 ;
                 ; LOAD ASCII CHARACTER
                 ;
5A40 CD 5949     LOAD6:  CALL     GETCH
5A43 47                  LD       B,A
5A44 18 F4               JR       LOAD4
                 ;
                 ; COPY BYTE FROM B TO MEMORY
                 ; AND SEE THAT IT GOT THERE
                 ;
5A46 70          CHEKM:  LD       (HL),B    ;PUT IN MEM
5A47 7E                  LD       A,(HL)    ;GET BACK
5A48 B8                  CP       B         ;SAME?
5A49 C8                  RET      Z         ;OK
5A4A F1          ERRP:   POP      AF        ;RAISE STACK
5A4B 3E 42       ERRB:   LD       A,'B'     ;BAD
5A4D CD 5835             CALL     OUTT
5A50 CD 59F3             CALL     OUTSP
5A53 C3 59EB             JP       OUTHL     ;POINTER
                 ;
                 ; DISPLAY STACK POINTER REGISTER
                 ;
5A56 21 0000     REGS:   LD       HL,0
5A59 39                  ADD      HL,SP
5A5A C3 59EB             JP       OUTHL
                 ;
                 ; ZERO A PORTION OF MEMORY
                 ;
5A5D CD 5999     ZERO:   CALL     RDHLDE    ;RANGE
5A60 06 00               LD       B,0
5A62 18 08               JR       FILL2
                 ;
                 ; FILL A PORTION OF MEMORY
                 ;
5A64 CD 5A80     FILL:   CALL     HLDEBC    ;RANGE, BYTE
5A67 FE F7               CP       APOS      ;APOSTROPHE?
5A69 28 0F               JR       Z,FILL4   ;YES, ASCII
5A6B 41                  LD       B,C
5A6C 7C          FILL2:  LD       A,H       ;FILL BYTE
5A6D FE 57               CP       HIGH STACK ;TOO FAR?
5A6F D2 59E0             JP       NC,ERROR   ;YES
5A72 CD 5A46     FILL3:  CALL     CHEKM     ;PUT, CHECK
5A75 CD 5A0C             CALL     TSTOP     ;DONE?
5A78 18 F2               JR       FILL2     ;NEXT
                 ;
5A7A CD 5949     FILL4:  CALL     GETCH     ;ASCII CHAR
5A7D 47                  LD       B,A
5A7E 18 F2               JR       FILL3
                 ;
                 ; GET H,L D,E AND B,C
                 ;
5A80 CD 5A8E     HLDEBC: CALL     HLDECK    ;RANGE
5A83 DA 59E0             JP       C,ERROR   ;NO BYTE
5A86 E5                  PUSH     HL
```

```
5A87 CD 59AE            CALL    READHL   ;3RD INPUT
5A8A 44                 LD      B,H      ;MOVE TO
5A8B 4D                 LD      C,L      ; B,C
5A8C E1                 POP     HL
5A8D C9                 RET
                ;
                ; GET 2 ADDRESSES, CHECK THAT
                ; ADDITIONAL DATA IS INCLUDED
                ;
5A8E CD 59A3    HLDECK: CALL    HHLDE    ;2 ADDR
5A91 DA 59E0            JP      C,ERROR  ;THAT'S ALL
5A94 C3 599C            JP      RDHLD2   ;CHECK
                ;
                ; MOVE A BLOCK OF MEMORY H,L-D,E TO B,C
                ;
5A97 CD 5A80    MOVE:   CALL    HLDEBC   ;3 ADDR
5A9A CD 5AA3    MOVDN:  CALL    MOVIN    ;MOVE/CHECK
5A9D CD 5A0C            CALL    TSTOP    ;DONE?
5AA0 03                 INC     BC       ;NO
5AA1 18 F7              JR      MOVDN
                ;
5AA3 7E         MOVIN:  LD      A,(HL)   ;BYTE
5AA4 02                 LD      (BC),A   ;NEW LOCATION
5AA5 0A                 LD      A,(BC)   ;CHECK
5AA6 BE                 CP      (HL)     ;IS IT THERE?
5AA7 C8                 RET     Z        ;YES
5AA8 60                 LD      H,B      ;ERROR
5AA9 69                 LD      L,C      ;INTO H,L
5AAA C3 5A4A            JP      ERRP     ;SHOW BAD
                ;
                ; SEARCH FOR 1 OR 2 BYTES OVER THE
                ; RANGE H,L D,E. BYTES ARE IN B,C
                ; B HAS CARRIAGE RETURN IF ONLY ONE BYTE
                ; PUT SPACE BTWEEN BYTES IF TWO
                ; FORMAT: START STOP BYTE1 BYTE2
                ;
5AAD CD 5A80    SEARCH: CALL    HLDEBC   ;RANGE, 1ST BYTE
5AB0 06 0D      SEAR2:  LD      B,CR     ;SET FOR 1 BYTE
5AB2 38 06              JR      C,SEAR3  ;ONLY ONE
5AB4 E5                 PUSH    HL
5AB5 CD 59AE            CALL    READHL   ;2ND BYTE
5AB8 45                 LD      B,L      ;INTO C
5AB9 E1                 POP     HL
5ABA 7E         SEAR3:  LD      A,(HL)   ;GET BYTE
5ABB B9                 CP      C        ;MATCH?
5ABC 20 10              JR      NZ,SEAR4 ;NO
5ABE 23                 INC     HL       ;YES
5ABF 78                 LD      A,B      ;ONLY 1?
5AC0 FE 0D              CP      CR
5AC2 28 04              JR      Z,SEAR5  ;YES
                ;
                ; FOUND FIRST MATCH, CHECK FOR SECOND
                ;
5AC4 7E                 LD      A,(HL)   ;NEXT BYTE
5AC5 B8                 CP      B        ;MATCH?
5AC6 20 06              JR      NZ,SEAR4 ;NO
                ;
```

```
5AC8 2B          SEAR5:  DEC     HL        ;A MATCH
5AC9 C5                  PUSH    BC
5ACA CD 59E8             CALL    CRHL      ;SHOW ADDR
5ACD C1                  POP     BC
5ACE CD 5A0C     SEAR4:  CALL    TSTOP     ;DONE?
5AD1 18 E7               JR      SEAR3     ;NO
                 ;
                 ; ASCII SUB-COMMAND PROCESSOR
                 ;
5AD3 CD 5949     ASCII:  CALL    GETCH     ;NEXT CHAR
5AD6 FE 44               CP      'D'       ;DISPLAY
5AD8 28 24               JR      Z,ADUMP
5ADA FE 53               CP      'S'       ;SEARCH
5ADC 28 42               JR      Z,ASCS
5ADE FE 4C               CP      'L'       ;LOAD
5AE0 C2 59E0             JP      NZ,ERROR
                 ;
                 ; LOAD ASCII CHARACTERS INTO MEMORY
                 ; QUIT ON CONTROL-X
                 ;
5AE3 CD 59AE             CALL    READHL    ;ADDRESS
5AE6 CD 59EB             CALL    OUTHL     ;PRINT IT
5AE9 CD 5815     ALOD2:  CALL    INPUTT    ;NEXT CHAR
5AEC CD 5835             CALL    OUTT      ;PRINT IT
5AEF 47                  LD      B,A       ;SAVE
5AF0 CD 5A46             CALL    CHEKM     ;INTO MEMORY
5AF3 23                  INC     HL        ;POINTER
5AF4 7D                  LD      A,L
5AF5 E6 7F               AND     7FH       ;LINE END?
5AF7 20 F0               JR      NZ,ALOD2 ;NO
5AF9 CD 59E8             CALL    CRHL      ;NEW LINE
5AFC 18 EB               JR      ALOD2
                 ;
                 ; DISPLAY MEMORY IN STRAIGHT ASCII.
                 ; KEEP CARRIAGE RETURN, LINE FEED, CHANGE
                 ; TAB TO SPACE, REMOVE OTHER CONTROL CHAR.
                 ;
5AFE CD 5999     ADUMP:  CALL    RDHLDE    ;RANGE
5B01 7E          ADMP2:  LD      A,(HL)    ;GET BYTE
5B02 FE 7F               CP      DEL       ;HIGH BIT ON?
5B04 30 15               JR      NC,ADMP4 ;YES
5B06 FE 20               CP      ' '       ;CONTROL?
5B08 30 0E               JR      NC,ADMP3 ;NO
5B0A FE 0D               CP      CR        ;CARR RET?
5B0C 28 0A               JR      Z,ADMP3 ;YES, OK
5B0E FE 0A               CP      LF        ;LINE FEED?
5B10 28 06               JR      Z,ADMP3 ;YES, OK
5B12 FE 09               CP      TAB
5B14 20 05               JR      NZ,ADMP4 ;SKIP OTHER
5B16 3E 20               LD      A,' '     ;SPACE FOR TAB
5B18 CD 5835     ADMP3:  CALL    OUTT      ;SEND
5B1B CD 5A0C     ADMP4:  CALL    TSTOP     ;DONE?
5B1E 18 E1               JR      ADMP2     ;NO
                 ;
                 ; SEARCH FOR 1 OR 2 ASCII CHARACTERS
                 ; NO SPACE BETWEEN ASCII CHARS
                 ; FORMAT: START STOP 1 OR 2 ASCII CHAR
                 ;
```

```
5B20 CD 5999    ASCS:   CALL    RDHLDE   ;RANGE
5B23 CD 5949            CALL    GETCH    ;FIRST CHAR
5B26 4F                 LD      C,A
5B27 CD 5949            CALL    GETCH    ;2ND OR CARR RET
5B2A DA 5AB0            JP      C,SEAR2  ;ONLY ONE CHAR
5B2D 47                 LD      B,A      ;2ND
5B2E C3 5ABA            JP      SEAR3
                ;
                ; INPUT FROM ANY PORT (Z-80 VERSION)
                ;
5B31 CD 59AE    IPORT:  CALL    READHL   ;PORT
5B34 4D                 LD      C,L      ;PORT TO C
5B35 ED 68              IN      L,(C)    ;INPUT
5B37 CD 59EF            CALL    OUTLL    ;HEX VALUE
                ;
                ; PRINT L REGISTER IN BINARY (Z-80 VER)
                ;
5B3A 06 08      BITS:   LD      B,8      ;8 BITS
5B3C CB 25      BIT2:   SLA     L        ;SHIFT L LEFT
5B3E 3E 18              LD      A,'0'/2  ;HALF OF 0
5B40 8F                 ADC     A,A      ;DOUBLE+CARRY
5B41 CD 5835            CALL    OUTT     ;PRINT BIT
5B44 10 F6              DJNZ    BIT2     ;8 TIMES
5B46 C9                 RET
                ;
                ; OUTPUT BYTE FROM PORT (Z-80 VERSION)
                ; FORMAT IS: O,PORT,BYTE
                ;
5B47 CD 59AE    OPORT:  CALL    READHL   ;PORT
5B4A 4D                 LD      C,L
5B4B CD 59AE            CALL    READHL   ;DATA
5B4E ED 69              OUT     (C),L    ;OUTPUT
5B50 C9                 RET
                ;
                ; HEXADECIMAL MATH, SUM AND DIFFERENCE
                ;
5B51 CD 59A3    HMATH:  CALL    HHLDE    ;TWO NUMBERS
5B54 E5                 PUSH    HL       ;SAVE H,L
5B55 19                 ADD     HL,DE    ;SUM
5B56 CD 59EB            CALL    OUTHL    ;PRINT IT
5B59 E1                 POP     HL
5B5A B7                 OR      A        ;CLEAR CARRY
5B5B ED 52              SBC     HL,DE
5B5D C3 59EB            JP      OUTHL    ;DIFFERENCE
                ;
                ; MEMORY TEST THAT DOESN'T ALTER CURRENT BYTE
                ; INPUT RANGE OF ADDRESSES, ABORT WITH ^X
                ;
5B60 CD 5999    JUST:   CALL    RDHLDE   ;RANGE
5B63 E5                 PUSH    HL       ;SAVE START ADDR
5B64 7E         JUST2:  LD      A,(HL)   ;GET BYTE
5B65 2F                 CPL              ;COMPLEMENT IT
5B66 77                 LD      (HL),A   ;PUT IT BACK
5B67 BE                 CP      (HL)     ;DID IT GO?
5B68 C2 5B7E            JP      NZ,JERR  ;NO
5B6B 2F                 CPL              ;ORIGINAL BYTE
5B6C 77                 LD      (HL),A   ;PUT IT BACK
```

```
5B6D 7D         JUST3:  LD      A,L     ;PASS
5B6E 93                 SUB     E       ; COMPLETED?
5B6F 7C                 LD      A,H
5B70 9A                 SBC     A,D
5B71 23                 INC     HL
5B72 38 F0              JR      C,JUST2 ;NO
                ;
                ; AFTER EACH PASS,
                ; SEE IF ABORT WANTED
                ;
5B74 CD 5827            CALL    INSTAT  ;INPUT?
5B77 C4 5815            CALL    NZ,INPUTT ;YES, GET IT
5B7A E1                 POP     HL      ;START ADDR
5B7B E5                 PUSH    HL      ;SAVE AGAIN
5B7C 18 E6              JR      JUST2   ;NEXT PASS
                ;
                ; FOUND MEMORY ERROR, PRINT POINTER AND
                ; BIT MAP: 0=GOOD, 1=BAD BIT
                ;
5B7E F5         JERR:   PUSH    AF      ;SAVE COMPLEMENT
5B7F CD 59E8            CALL    CRHL    ;PRINT POINTER
5B82 F1                 POP     AF
5B83 AE                 XOR     (HL)    ;SET BAD BITS
5B84 E5                 PUSH    HL      ;SAVE POINTER
5B85 6F                 LD      L,A     ;BIT MAP TO L
5B86 CD 5B3A            CALL    BITS    ;PRINT BINARY
5B89 E1                 POP     HL
5B8A 18 E1              JR      JUST3   ;CONTINUE
                ;
                ; REPLACE HEX BYTE WITH ANOTHER
                ; FORMAT IS: START, STOP, ORIG, NEW
                ;
5B8C CD 5A80    REPL:   CALL    HLDEBC  ;RANGE, 1ST BYTE
5B8F DA 59E0            JP      C,ERROR ;NO 2ND
5B92 41                 LD      B,C     ;1ST TO B
5B93 E5                 PUSH    HL
5B94 CD 59AE            CALL    READHL  ;2ND BYTE
5B97 4D                 LD      C,L     ;INTO C
5B98 E1                 POP     HL
5B99 7E         REPL2:  LD      A,(HL)  ;FETCH BYTE
5B9A B8                 CP      B       ;A MATCH?
5B9B 20 06              JR      NZ,REPL3 ;NO
5B9D 71                 LD      (HL),C  ;SUBSTITUTE
5B9E 79                 LD      A,C
5B9F BE                 CP      (HL)    ;SAME?
5BA0 C2 5A4B            JP      NZ,ERRB ;NO, BAD
5BA3 CD 5A0C    REPL3:  CALL    TSTOP   ;DONE?
5BA6 18 F1              JR      REPL2
                ;
                ; GIVE RANGE OF 1ST BLOCK AND START OF SECOND
                ;
5BA8 CD 5A80    VERM:   CALL    HLDEBC  ;3 ADDRESSES
5BAB 0A         VERM2:  LD      A,(BC)  ;FETCH BYTE
5BAC BE                 CP      (HL)    ;SAME AS OTHER?
5BAD 28 19              JR      Z,VERM3 ;YES
5BAF E5                 PUSH    HL      ;DIFFERENT
5BB0 C5                 PUSH    BC
```

```
5BB1 CD 59E8              CALL    CRHL      ;PRINT 1ST POINTER
5BB4 4E                   LD      C,(HL)    ;FIRST BYTE
5BB5 CD 59F0              CALL    OUTHEX    ;PRINT IT
5BB8 3E 3A                LD      A,':'
5BBA CD 5835              CALL    OUTT
5BBD E1                   POP     HL        ;B,C TO H,L
5BBE CD 59EB              CALL    OUTHL     ;SECOND POINTER
5BC1 4E                   LD      C,(HL)    ;2ND BYTE
5BC2 CD 59F8              CALL    OUTHX     ;PRINT IT
5BC5 4D                   LD      C,L       ;RESTORE C
5BC6 44                   LD      B,H       ;AND B
5BC7 E1                   POP     HL        ;AND H,L
5BC8 CD 5A0C      VERM3:  CALL    TSTOP     ;DONE?
5BCB 03                   INC     BC        ;2ND POINTER
5BCC 18 DD                JR      VERM2
                  ;
                          END     START
```

Symbols:

```
ADMP2    5B01    ADMP3    5B18    ADMP4    5B1B    ADUMP    5AFE
ALOD2    5AE9    APOS     FFF7    ASCII    5AD3    ASCS     5B20
BACKUP   0008    BIT2     5B3C    BITS     5B3A    CALLS    5A15
CDATA    0011    CHEKM    5A46    CIN      5809    COLD     587E
COUT     5806    CR       000D    CRHL     59E8    CRLF     5935
CSTAT    0010    CTRH     0008    CTRP     0010    CTRQ     0011
CTRS     0013    CTRX     0018    DEL      007F    DUMP     595E
DUMP2    5961    DUMP3    5964    DUMP4    5975    DUMP5    597E
ERRB     5A4B    ERROR    59E0    ERRP     5A4A    FILL     5A64
FILL2    5A6C    FILL3    5A72    FILL4    5A7A    GCHAR    580F
GETC4    595C    GETCH    5949    GO       5A14    HEX1     5A01
HHLDE    59A3    HLDEBC   5A80    HLDECK   5A8E    HMATH    5B51
IBUFC    57A5    IBUFF    57A6    IBUFP    57A3    INLN     580C
INMSK    0001    INPL2    5902    INPL3    591B    INPLB    593F
INPLC    5929    INPLE    5924    INPLI    590A    INPLN    58FD
INPUT2   581A    INPUTT   5815    INSTAT   5827    IPORT    5B31
JERR     5B7E    JUST     5B60    JUST2    5B64    JUST3    5B6D
LDATA    0013    LF       000A    LOAD     5A19    LOAD2    5A1C
LOAD3    5A3D    LOAD4    5A3A    LOAD6    5A40    LOMSK    0002
LOUT     585B    LSTAT    0012    MOVDN    5A9A    MOVE     5A97
MOVIN    5AA3    MSIZE    58A5    NIB      59D0    NNULS    0004
NPAGE    589A    OMSK     0002    OPORT    5B47    ORGIN    5800
OUT2     583C    OUT3     5848    OUT4     5851    OUTCR    5874
OUTCR2   5876    OUTH     5812    OUTHEX   59F0    OUTHL    59EB
OUTHX    59F8    OUTLL    59EF    OUTSP    59F3    OUTT     5835
PASC2    5994    PASC3    5996    PASCI    598B    PORTN    57A0
RDHL2    59B3    RDHL4    59C5    RDHL5    59CD    RDHLD2   599C
RDHLDE   5999    READHL   59AE    REGS     5A56    REPL     5B8C
REPL2    5B99    REPL3    5BA3    RESTRT   5803    SEAR2    5AB0
SEAR3    5ABA    SEAR4    5ACE    SEAR5    5AC8    SEARCH   5AAD
SETLST   582C    STACK    57A0    START    5800    TAB      0009
TABLE    58C9    TOP      0018    TSTOP    5A0C    VERM     5BA8
VERM2    5BAB    VERM3    5BC8    WARM     5891    ZERO     5A5D
```

CONVERSION OF THE MONITOR TO Z-80 MNEMONICS

If you used 8080 mnemonics to program the monitor in Chapter 6, you can now convert it to Z-80 mnemonics. The form of the mnemonics depends on the type of assembler you have. The Microsoft assembler accepts both the Intel 8080 and the Zilog Z-80 mnemonics. Since most other assemblers use only one or the other, you may need a second assembler.

The Digital Research assembler MAC requires the 8080 mnemonics, but it can generate Z-80 code with an accompanying macro library. The Xitan assembler utilizes 8080 mnemonics for the common set of 8080-type instructions and Zilog-like instructions for the others.

First, make a working copy of the monitor using PIP, a CP/M utility routine.

```
PIP   MONZ.ASM=MON17.ASM[V]
```

If you are using MAC or the Xitan assembler, skip to the next section. Otherwise, use the system editor to make the necessary changes to the new file. The conversion can be easily performed with the global substitute command of the Word-Master or the CP/M editor. For example, the 8080 mnemonic

```
MOV    A,M
```

can be changed to the equivalent Z-80 mnemonic

```
LD     A,(HL)
```

with the command

```
*SMOV<tab>A,M$LD<tab>A,(HL)$0TT
```

The $ symbols indicate that the escape key is pressed. The "tab" refers to the ASCII tab key, a control-I. You may find the cross-reference list for 8080 and Z-80 mnemonics, given in Appendix G, helpful in the conversion process.

After changing the monitor to Z-80 mnemonics, assemble it and carefully check the assembly listing to see that the hex code is correct. The Z-80 version at this time should generate the same hex code as the 8080 version. A further check can be made with the monitor's V command. Load the binary code into memory with an offset. A command of

```
DDT
IMONZ.HEX
RF000
```

will load the new version 4K bytes below the regular monitor position. Branch to the monitor prepared in the last chapter. Then compare its code to the new version using the verify command. If there is a discrepancy, find the error and correct it. When you are convinced that the Z-80 version produces the same code as the 8080 version, you can begin the alterations to reduce the monitor's size.

REDUCING THE MONITOR SIZE

In this section you will reduce the monitor size by converting many of the 3-byte absolute jump instructions into 2-byte relative jump instructions. This change will make room for additional features. There are five types of jumps to be changed.

```
absolute          relative         condition
 jump              jump
JP    X           JR    X          unconditional
JP    Z,X         JR    Z,X        zero
JP    NZ,X        JR    NZ,X       not zero
JP    C,X         JR    C,X        carry
JP    NC,X        JR    NC,X       not carry
```

Not all of the absolute jumps can be converted in this way since the relative jumps are limited to a distance of about 126 bytes.

Another way to obtain more space is to move some of the subroutines to more advantageous locations. This will allow a few more absolute jumps to be converted into relative jumps. For example, several routines contain a jump to the routine ERROR. These can be placed together in a group. Then the ERROR routine can be moved into the middle of the group.

Another change will free up three more bytes. Notice that subroutine OUTSP ends with the instruction

```
JP    OUTT
```

If this subroutine were located directly ahead of subroutine OUTT, then the jump instruction would not be necessary. Actually, this type of change has already been used extensively in our monitor. Subroutines CRHL, OUTHL, OUTHEX, and OUTSP are all directly related. They initially could have been programmed (using Z-80 mnemonics) as

```
CRHL:   CALL    CRLF
        CALL    OUTHL
        RET
;
OUTHL:  LD      C,H
        CALL    OUTHX
OUTLL:  LD      C,L
        CALL    OUTHEX
        RET
;
OUTHEX: CALL    OUTHX
        CALL    OUTSP
        RET
;
OUTSP:  LD      A,' '
        CALL    OUTT
        RET
```

The CALL/RET combination at the end of each routine can be replaced by a JP instruction. Then, since the calling program is located directly above the called program, the jump instruction becomes unnecessary. Thus, four bytes are saved in each of the first three routines. Furthermore, if this entire block of four subroutines were located just prior to subroutine OUTT, we could eliminate the final JP OUTT instruction and save three more bytes.

While this kind of subroutine rearrangement can be used to make the overall program smaller, there is a penalty. The readability is reduced. We have traded comprehension for space. This may not, in general, be a worthwhile tradeoff for assembly-language programming. Such programs are more difficult to understand than those written in a high-level language such as Pascal or BASIC. Furthermore, assembly-language programs are typically much shorter than they would be if written in a higher-level language. But if packing a maximum number of features into a 1K PROM is your goal, then this technique may be worth it.

GETTING MORE FREE SPACE

The two instructions

```
DEC     B
JP      NZ,X
```

which generate four bytes of code appear in two places. Replace them with the 2-byte instruction

```
DJNZ    X
```

One location is just prior to the label MSIZE (address 58A5 in Listing 7.1) and the other is in subroutine BITS (5B3A). This change will free four more bytes.

The 16-bit subtraction routine in HMATH (5B51) has been improved. The sequence of instructions

```
LD      A,L
SUB     A,E
LD      L,A
LD      A,H
SBC     A,D
LD      H,A
```

is replaced by the shorter, double-precision subtraction:

```
OR      A       ;reset carry
SBC     HL,DE   ;subtract
```

Three more bytes are freed by this change.

Since the Z-80 contains a set of instructions for direct rotation of data in the general CPU registers, we can simplify subroutine BITS. In the 8080 version, the three instructions

```
MOV     A,L
ADD     A
MOV     L,A
```

are used to move the data from a general register to the accumulator, perform the shift, then move it back. The 2-byte, Z-80 arithmetic shift left instruction

```
SLA     L
```

performs the shift directly in the L register.

The 8080 can output a byte only from the accumulator, and can input a byte only to the accumulator. Furthermore, the address of the peripheral must be located in memory immediately following the first byte of the input or output instruction.

The port-input routine IPORT and the port-output routine OPORT in the system monitor utilized subroutine PUTIO. This routine writes the desired IN or OUT instruction in memory, the requested port address and then a return instruction. There is a Z-80 instruction that can perform I/O from any register. The address of the peripheral is located in register C in this case. Since the port address does not have to be located in memory, subroutine PUTIO can be eliminated. The resulting Z-80 code is 19 bytes shorter than the 8080 version. See Listing 7.1 for the new versions of IPORT (5B31) and OPORT (5B47).

Since you are nearly finished with the development of the monitor program, you can gain some more space by removing the routines that print the version number. There are four areas involved. First, delete the line near the beginning that identifies the version number.

```
VERS    EQU     '17'
```

Second, remove four lines starting with the label SIGNON. Third, delete the two lines starting on the line after the label COLD.

```
LD      DE,SIGNON
CALL    SENDM
```

Fourth, remove the entire subroutine SENDM, but keep a copy of it in case you want to incorporate it in another program.

PERIPHERAL PORT INITIALIZATION

There are two schools of thought on peripheral port initialization. One approach is to initialize ports only on a cold start or a warm start. The other way is to initialize a port each time it is used. The method you use depends on the integrity of your system.

The approach taken in this chapter initializes ports only on a cold start. The instructions are placed just after the label COLD. In anticipation of adding a printer-output routine, we include the initialization for two separate peripherals.

Ports which need initialization utilize a control register for this purpose. The address of the control register is the same as the status register. A CPU IN instruction reads the status register, while a CPU OUT instruction to the same address writes into the control register. A typical initialization procedure requires two OUT instructions. The first is used to reset the port; the second is used to set the desired options. The values shown in the listing correspond to a Motorola 6850 ACIA serial port set for eight data bits, one stop bit, and no interrupts.

```
LD      A,3
OUT     (CSTAT),A       ; RESET
LD      A,15H
OUT     (CSTAT),A       ; SET FEATURES
```

PRINTER OUTPUT ROUTINES

Up to this point, we have been writing programs for output to a console video screen. We output an ASCII backspace character for error correction so that the cursor will actually back up on the screen. We also included a pair of scroll commands: control-S to freeze the display and control-Q to resume the scrolling.

Sometimes, however, we want computer output we can look at after the computer has been shut off. A printer or list device is what we need for this purpose. We will not want to use the printer as a main console, though, because it is too slow.

For sophisticated operating systems like CP/M, the software for the list device is wholly separate. For example, we can divert a disk file to the printer and none of the system commands will appear on the listing.

Our approach will be a little different. The video console will always display all output whether the printer is on or not. Of course, when the printer is engaged, the console speed will be reduced to that of the printer. We will both enable and disable the printer with a control-P command, just as in CP/M. We refer to the control-P command as a list toggle: the same command turns it on or off. The output includes the echoing of the commands typed in from the console keyboard.

Both the input and output routines will have to be changed if you want to incorporate the printer routines. In addition, two new subroutines will be added. First, add two new lines to the input routine; they will look for a control-P from the console keyboard. If a control-P is found, the program will branch to a new subroutine called SETLST. The two new lines appear in subroutine INPUTT (5815).

```
CP      CTRP     ;^P
JR      Z,SETLST ;LIST
```

Subroutine SETLST (582C), containing 4 lines of code, is added just after subroutine INSTAT.

```
SETLST: LD      A,(PORTN) ;CHECK FLAG
        CPL               ;INVERT
        LD      (PORTN),A ;SAVE
        JR      INPUTT    ;NEXT BYTE
```

This routine complements the printer flag (PORTN) when a control-P is typed. The output routine uses this flag to determine whether to send output to the printer. Notice that in Chapter 6, the identifier PORTN was used to set up the port number for the I and O commands. This feature is not needed for the Z-80 version, so we can use the location for the printer flag instead.

The third new section is placed in the output routine OUTT (5835).

```
LD      A,(PORTN) ;WHERE?
OR      A         ;ZERO?
JR      NZ,LOUT   ;LIST OUTPUT
```

This part checks the flag PORTN to see if output is to be sent to the printer.

The fourth routine is LOUT (585B); it follows OUT4. This routine sends output to both the console and the printer. It first checks the status port for the printer. When the output bit indicates ready, a byte is sent to the printer. Since the console video screen operates so much faster than the printer, there is no need to check the console-ready flag. The byte is therefore also sent directly to the console by the next instruction. The output appears simultaneously at both devices.

DELAY AFTER A CARRIAGE RETURN

Video screens operate with electron beams that move very fast. Mechanical printers, on the other hand, are much slower. For some printers, the time it takes to execute a carriage return is so great that the first few characters of the next line may be lost. The solution is to have the computer do something else for a little while after it sends a carriage return.

One method of slowing down the computer is to arrange for it to send binary zeros, called nulls, after each carriage return or carriage-return/line-feed pair. One routine for accomplishing this is as follows.

```
CRLF:   LD      A,CR    ;CARRIAGE RET
        CALL    OUTT    ;SEND
        LD      A,LF    ;LINE FEED
        CALL    OUTT    ;SEND
        XOR     A       ;GET A NULL
        CALL    OUTT    ;SEND IT
        CALL    OUTT    ;A SECOND ONE
        CALL    OUTT    ;A THIRD
        JP      OUTT    ;THE FOURTH
```

But this approach may cause trouble if the printer circuits attempt to interpret the null characters.

A different approach is taken with the list-output routine, LOUT, shown in Listing 7.1. After each carriage return is sent, the computer starts executing a double loop. The inner loop is executed 250 times. The outer loop is set according to the equivalent number of nulls that are needed. No nulls are actually sent, though, in this case.

The disadvantage of this method is that the resultant delay time is a function of the computer speed. A Z-80 running at 4 MHz would require approximately twice the number of loops as would a 2-MHz Z-80. Thus, the loop-initialization values may have to be adjusted to the particular computer.

Be careful to tailor the port-initialization routines to your system or remove them if they are not needed. The time delay in the list-output routine should also be removed if it is not needed. If you are not sure whether a delay is necessary, then leave it in, at least for the first version. Then use the memory load command of the monitor itself to reduce the delay values on the two loops. When you reduce the delay time to too small a value, then you will notice that some of the characters are missing from the beginning of some of the lines.

A sample loop-change session could look like this.

```
>D5870 587F
5870 C0D51678 1EFA1D20 1520. . .
>L5873
5873 x 78 3C
5873 . 1E ↑X (to quit)
```

The first command line is used to display the memory region containing the loop constants. Then the outer loop value of 78 hex is changed to 3C hex which is half the value. As long as you change the timing-loop values with the printer disengaged, no problem should occur. After each change in the timing loops, re-engage the printer with a control-P. Display several lines on the printer by giving the D command. Check to see if any of the first few characters of each line are missing. If everything is all right, then again reduce the loop constant until characters are lost. (Be sure to disengage the printer between each change.)

CHAPTER EIGHT

Number-Base Conversion

This chapter deals with assembly language routines that can be used to convert data from one form to another. The first part deals with the conversion of a sequence of ASCII characters called a *string* into a binary number. The second part reverses the procedure; binary numbers are converted into ASCII strings. The characters in each string represent digits in one of the common bases 2, 8, 10, or 16. The corresponding binary number may be 4 bits, 8 bits, or 16 bits in size.

All of the programs in this chapter are designed to run with the system monitor developed in Chapters 6 and 7. Some of the monitor's input and output facilities are needed. These include the console input buffer which supplies the characters, the binary-to-hexadecimal conversion routine which will print the answer in hexadecimal, and the console output routine needed for the error message.

The monitor error-correction features are available during input. Pressing the DEL (or RUB) key or the backspace (control-H) key will delete the previously typed character and remove it from the console video screen. If the list routines have been incorporated into the monitor, the printer can be turned on by typing a control-P. When you have finished with each routine, you can return to the monitor simply by typing a control-X.

THE ASCII CODE

When a key is pressed on a computer terminal, a unique signal is sent to the computer. There are several, very different ways of electronically encoding this signal. ASCII, which stands for American Standard Code for Information Interchange, is the most commonly used code. Appendix A gives the 128 ASCII characters with the corresponding values expressed in decimal, hexadecimal, octal, and binary. EBCDIC, which is used by IBM, is another coding technique.

The ASCII table can be divided into four parts. Part 1 of the table contains the nonprinting control characters. Part 2 contains most of the special characters such as $, %, and #, and the digits 0–9. The uppercase letters are found in part 3, and the lowercase letters are found in part 4.

Computer terminals typically have a keyboard that looks like a typewriter. There is a shift key to change from lowercase letters to uppercase letters. In addition to the shift key, there will usually be a control key. This key will give the letter keys a third meaning. Thus the user can enter a lowercase letter A, an uppercase letter A (a shift A), or a control-A. The bit patterns are:

```
110 0001  lowercase A
100 0001  uppercase A
000 0001  control-A
```

It can be seen from the pattern that the shift key resets bit 5 while the control key resets both bits 5 and 6.

Some of the commonly used control functions such as the carriage return (control-M), line feed (control-J), the horizontal tab (control-I), and the backspace (control-H) may have their own separate keys.

All console input to the computer will be in the form of ASCII characters. The console will send eight data bits for each character. But the ASCII code contains only seven bits per character. Consequently, the eighth, high-order bit is not needed. The user will need to have routines for converting strings of ASCII characters into the ultimate numbers that will reside in memory. For example, if the operator enters the string

```
3014
```

from the console, the computer would actually receive the bit patterns

```
011 0111   (ASCII 3)
011 0000   (ASCII 0)
011 0001   (ASCII 1)
011 0100   (ASCII 4)
```

The next step is to convert the string into a 16-bit number. The conversion scheme that is chosen depends on whether the string represents a decimal number, an octal number, or a hexadecimal number.

Additionally, a check is made to ensure that each character in the string is within the proper range. For example, octal numbers must contain only the digits zero through 7. The digits 8 and 9, the letters A through Z, and the other characters are not used. Finally, we may need a special character, called a *delimiter*, to indicate the end of a string. We will use a space or a carriage return for this purpose. Thus the string of characters

```
1034 2347
```

will be interpreted as two separate numbers since a space appears in the middle.

The ASCII string may need to be converted into a 4-bit nibble, an 8-bit byte destined for a CPU register, or a 16-bit word meant for a double register. Furthermore, the format may be either *free entry* or *fixed entry.* The choice is a matter of personal taste. With free entry, leading zeros are not needed. The entries

```
0004
004
04
4
```

are all interpreted as the same number. An additional feature is that you can recover from an error by retyping the entry on the same line. Suppose that the 4-digit number 1035 is desired but 1045 was typed by mistake. The correct value can be immediately typed without a space.

```
10351045
```

If two 4-digit numbers are needed, they must be separated by a delimiter.

```
1045 1055
```

With the fixed-entry format, the required number of digits, including leading zeros, must be entered. But since an end-of-string indicator is not needed, two numbers can be run together. The fixed-entry expression

```
10451055
```

will be interpreted as two separate numbers.

CONVERSION OF ASCII-ENCODED BINARY CHARACTERS TO AN 8-BIT BINARY NUMBER IN REGISTER C

One of the simplest base-conversion routines is the ASCII-to-binary program. This program takes a string of ASCII-encoded ones and zeros from the console input buffer and produces an 8-bit binary number in register C. The hexadecimal equivalent of the number is printed on the console. If the operator types the string

```
10101100
```

the keyboard actually transmits the following sequence.

011 0001
011 0000
011 0001
011 0000
011 0001
011 0001
011 0000
011 0000

The conversion routine will take this combination, convert it to the binary number

10101100

and place it into the C register.

Type the routine shown in Listing 8.1 Set the assembly location somewhere below the monitor's stack and include the address of the monitor using the EQU directive. The monitor I/O routines are defined relative to the monitor's address.

Listing 8.1. ASCII-encoded binary to binary in C.

```
                    ; THIS PROGRAM IS DESIGNED TO OPERATE
                    ; WITH THE SYSTEM MONITOR AT 5800 HEX.
                    ;
                    ; JAN 29, 80
                    ;
5000                ORG      5000H
                    ;
5800 =              MONIT    EQU      5800H
5806 =              OUTT     EQU      MONIT+6
580C =              INPLN    EQU      MONIT+0CH
580F =              GETCH    EQU      MONIT+0FH
5812 =              OUTHX    EQU      MONIT+12H
                    ;
5000 3E0D           START:   MVI      A,0DH    ;CARR RET
5002 CD0658                  CALL     OUTT
5005 3E0A                    MVI      A,0AH    ;LINE FEED
5007 CD0658                  CALL     OUTT
500A CD0C58                  CALL     INPLN    ;GET A LINE
500D CD1B50                  CALL     BBIN     ;CONVERT
5010 CD1258                  CALL     OUTHX    ;HEX VALUE
5013 3E20                    MVI      A,' '
5015 CD0658                  CALL     OUTT
                    ;
                    ; THE NEXT INSTRUCTION IS NEEDED WHEN
                    ; THE ROUTINE IN LISTING 8.9 IS APPENDED
                    ;        CALL     BITS     ;BIN TO ASCII
5018 C30050                  JMP      START    ;NEXT VALUE
                    ;
                    ; SUBROUTINE TO CONVERT UP TO 8 ASCII-
                    ; ENCODED BINARY CHARACTERS INTO AN
                    ; 8-BIT BINARY NUMBER IN C
                    ;
```

```
501B E5         BBIN:   PUSH    H            ;SAVE REGS
501C 210000             LXI     H,0          ;CLEAR
501F CD0F58     BBIN2:  CALL    GETCH        ;GET CHAR
5022 DA3A50             JC      BBIN3        ;LINE END
5025 D630               SUI     '0'          ;CONV TO BINARY
5027 DA3550             JC      BBIN4        ; < 0
502A FE02               CPI     2
502C D23D50             JNC     ERROR        ; > 1
502F 29                 DAD     H            ;SHIFT LEFT
5030 B5                 ORA     L            ;ADD NEW CHAR
5031 6F                 MOV     L,A
5032 C31F50             JMP     BBIN2        ;NEXT
                ;
                ; CHECK FOR BLANK AT END
                ;
5035 FEF0       BBIN4:  CPI     (' '-'0') AND 0FFH
5037 C23D50             JNZ     ERROR        ;NOT BLANK
503A 4D         BBIN3:  MOV     C,L          ;8 BITS TO C
503B E1                 POP     H            ;RESTORE
503C C9                 RET
                ;
                ; PRINT ? ON IMPROPER INPUT
                ;
503D C1         ERROR:  POP     B            ;RAISE STACK
503E C1                 POP     B
503F 3E3F               MVI     A,'?'
5041 CD0658             CALL    OUTT
5044 C30050             JMP     START        ;TRY AGAIN
```

Assemble the program and load it into memory; start it up by branching to the beginning of the program, the address of START. The monitor prompt symbol of > will appear on the console. Test the routine by entering the following binary numbers. Be sure to add a carriage return to the end of each line.

```
>0        (you type this)
00        (program responds with this)
>1
01
>10
02        (binary 10 is hexadecimal 2)
>11
03
>101
05        (binary 101 is hexadecimal 5)
>1111
0F
>11110000
F0
>10101010
AA
```

Of course, only ASCII zeros and ones are acceptable binary characters. Leading zeros are not necessary. If more than eight characters are entered, only the last eight are used. A question mark will be printed if a nonbinary character is typed. Typing errors can be corrected with a backspace or DEL keys.

The program consists of three parts. The first and third parts will be common to other conversion programs in this chapter. The first part calls the monitor to obtain data from the console. The last part converts the data to the hexadecimal equivalent and prints it on the console if valid. If the entry is invalid, a question mark is printed.

The conversion routine occupies the middle portion of the program. It works as follows: Since HL is used as a working register, the original contents are first saved on the stack. While this step is not necessary in this case, it may be needed in a real application. The HL register is then zeroed.

As each new character is obtained from the input buffer, it is converted from ASCII to binary by subtracting 30 hex, the value of the ASCII zero. An ASCII zero, which has a value 30 hex, becomes a binary zero. Similarly, an ASCII 1, which has a value of 31 hex, becomes a binary 1.

```
011 0000    ASCII zero
011 0000    subtract ASCII zero
--------
000 0000    binary zero

011 0001    ASCII 1
011 0000    subtract ASCII zero
--------
000 0001    binary 1
```

A check is made at this point to ensure that an invalid character has not been typed. Only three characters are acceptable: An ASCII zero, an ASCII 1, and a space. If the carry flag is set after the subtraction of an ASCII zero, then the input value was neither a zero nor a 1. But it might be a space character. A jump is made to subroutine BBIN4 in this case. This routine determines whether the current character is a space or some other character. A space is the normal end-of-string character (delimiter); other characters are not.

Each character is also checked to see that it is not greater than an ASCII 1. In either case, if any character in the string is found to be other than an ASCII zero or 1, then the subroutine is terminated with a jump to the error routine. At this point, the stack is raised with a POP instruction, and control returns to START at the top of the program.

If the input value is a zero or 1, the procedure continues. The current value in the HL register is multiplied by two, the binary number base. This arithmetic shift left is accomplished by adding the HL register to itself with the double-precision add DAD H. An alternate method would be to place the sum in the accumulator. In this case the multiplication is performed with an ADD A instruction. But then the intermediate sum would have to be saved in another register while the new character was checked.

The new character, which is now a binary zero or 1 in the accumulator, is added to the value in HL. The addition of the 8-bit accumulator to the 16-bit HL register generally requires several steps.

1. Add L to A.
2. Move sum in A to L.
3. Increment H if carry is set.

But in this particular case, the carry flag will never be set. Consequently, a simpler method of addition can be used. The one chosen for this application is to perform a logical OR operation with the L register and the accumulator.

CONVERSION OF ASCII DECIMAL CHARACTERS TO A BINARY NUMBER

We frequently find it useful to input computer data in the form of decimal numbers. We may then need a program to convert ASCII-encoded decimal numbers into binary form. The program given in Listing 8.2 will perform this task for us.

Listing 8.2. ASCII decimal to binary in HL.

```
                        ; THIS PROGRAM IS DESIGNED TO OPERATE
                        ; WITH THE SYSTEM MONITOR AT 5800 HEX
                        ;
                        ; JAN 22, 80
                        ;
5000                    ORG     5000H
                        ;
5800 =                  MONIT   EQU     5800H
5806 =                  OUTT    EQU     MONIT+6
580C =                  INPLN   EQU     MONIT+0CH
580F =                  GETCH   EQU     MONIT+0FH
5812 =                  OUTHX   EQU     MONIT+12H
                        ;
5000 3E0D               START:  MVI     A,0DH   ;CARR RET
5002 CD0658                     CALL    OUTT
5005 3E0A                       MVI     A,0AH   ;LINE FEED
5007 CD0658                     CALL    OUTT
500A CD0C58                     CALL    INPLN   ;GET A LINE
500D CD2050                     CALL    DBIN    ;ASCII TO DEC
5010 4C                         MOV     C,H     ;HIGH HALF
5011 CD1258                     CALL    OUTHX
5014 4D                         MOV     C,L
5015 CD1258                     CALL    OUTHX   ;LOW HALF
5018 3E20                       MVI     A,' '   ;SPACE
501A CD0658                     CALL    OUTT
                        ;
                        ; THE NEXT INSTRUCTION IS NEEDED WHEN
                        ; THE ROUTINE IN LISTING 8.11 IS USED
                        ;       CALL    BIND    ;BIN/DECIMAL
501D C30050                     JMP     START   ;NEXT VALUE
                        ;
                        ; ASCII DECIMAL TO 16-BIT IN H,L
                        ;
5020 D5                 DBIN:   PUSH    D       ;SAVE REGS
5021 210000                     LXI     H,0     ;CLEAR
```

```
5024 CD0F58     DBIN2:  CALL    GETCH    ;GET CHAR
5027 DA4650             JC      DBIN3    ;LINE END
502A D630               SUI     '0'      ;CONV TO BINARY
502C DA4150             JC      DBIN4    ; < 0
502F FE0A               CPI     10
5031 D24850             JNC     ERROR    ; > 10
5034 54                 MOV     D,H      ;COPY H,L
5035 5D                 MOV     E,L      ; INTO D,E
5036 29                 DAD     H        ;TIMES 2
5037 29                 DAD     H        ;TIMES 4
5038 19                 DAD     D        ;TIMES 5
5039 29                 DAD     H        ;TIMES 10
503A 5F                 MOV     E,A      ;NEW BYTE
503B 1600               MVI     D,0
503D 19                 DAD     D        ;ADD NEW BYTE
503E C32450             JMP     DBIN2    ;NEXT
                ;
                ; CHECK FOR BLANK AT END
                ;
5041 FEF0       DBIN4:  CPI     (' '-'0') AND 0FFH
5043 C24850             JNZ     ERROR    ;NOT BLANK
5046 D1         DBIN3:  POP     D        ;RESTORE
5047 C9                 RET
                ;
                ; PRINT ? ON IMPROPER INPUT
                ;
5048 E1         ERROR:  POP     H        ;RESTORE
5049 E1                 POP     H        ;STACK
504A 3E3F               MVI     A,'?'
504C CD0658             CALL    OUTT
504F C30050             JMP     START    ;TRY AGAIN
```

This new program uses our monitor for some of the necessary subroutines, just like the ASCII binary-to-binary program given in the previous section. Assemble the program, load it into memory, and branch to START. Again, the monitor prompt symbol > will appear. Try this routine by entering the following decimal numbers. Remember to type a carriage return at the end of each line.

```
>0        (you type this)
0000      (computer response)
>1
0001
>10       (decimal 10)
0A        (gives 0A hex)
>16
000F
>64
0100
>1024
0400
>65545
FFFF
>65547
0002
```

If the input consists of valid decimal characters, then the response will be the corresponding hexadecimal value. If an invalid character is typed, a question mark is printed as an error message and the program is restarted.

Since this routine generates a 16-bit binary number, the largest possible value is one less than 2 to the power 16. This is equivalent to a decimal number of 65,545. If a larger number than this is entered, the excess over 65,545 is lost. Thus, an input of 65547 will give a value of 0002. Remember that the monitor error-correction features and the control-P list toggle are available.

The algorithm is similar to the one in the previous section. The HL register pair is initially zeroed. The incoming character is converted from ASCII to binary, then checked to see that it is in the range 0–9. The current value is multiplied by 10 (the number base) prior to adding in the new character. The multiplication is accomplished with the double-register add instructions as follows.

```
MOV   D,H (duplicate H in D)
MOV   E,L (duplicate E in L)
DAD   H   (double initial value)
DAD   H   (quadruple it)
DAD   D   (5 times initial value)
DAD   H   (double, making 10 times
           the initial value)
```

The total is first duplicated in the DE register. Two double-precision DAD H operations multiply the original value by 4. Adding in the original value with the DAD D makes it 5. A final DAD H produces the desired multiplication by 10.

If only an 8-bit binary number is needed, then the multiplication can be performed in the accumulator rather than in the HL register. This will free the HL register for some other use, such as a memory pointer. The 8-bit version is given in Listing 8.3

Listing 8.3. ASCII decimal to binary in C

```
                    ; THIS PROGRAM IS DESIGNED TO OPERATE
                    ; WITH THE SYSTEM MONITOR AT 5800 HEX
                    ;
                    ; JAN 12, 80
                    ;
5000                ORG      5000H
                    ;
5800 =              MONIT    EQU      5800H
5806 =              OUTT     EQU      MONIT+6
580C =              INPLN    EQU      MONIT+0CH
580F =              GETCH    EQU      MONIT+0FH
5812 =              OUTHX    EQU      MONIT+12H
                    ;
5000 3E0D           START:   MVI      A,0DH    ;CARR RET
5002 CD0658                  CALL     OUTT
5005 3E0A                    MVI      A,0AH    ;LINE FEED
5007 CD0658                  CALL     OUTT
```

```
500A CD0C58             CALL    INPLN     ;GET A LINE
500D CD1650             CALL    DBIN      ;DEC TO BIN
5010 CD1258             CALL    OUTHX     ;HEX
5013 C30050             JMP     START
                ;
                ; CONVERT ASCII-DECIMAL TO 8-BIT BINARY
                ;
5016 0E00       DBIN:   MVI     C,0       ;CLEAR C
5018 CD0F58     DBIN2:  CALL    GETCH     ;GET CHAR
501B D8                 RC                ;LINE END
501C D630               SUI     '0'       ;CONV TO BINARY
501E DA3250             JC      DBIN4     ; < 0
5021 FE0A               CPI     10
5023 D23850             JNC     ERROR     ; > 10
5026 57                 MOV     D,A       ;SAVE NEW
5027 79                 MOV     A,C       ;SUM
5028 87                 ADD     A         ;TIMES 2
5029 4F                 MOV     C,A       ;SAVE
502A 87                 ADD     A         ;TIMES 4
502B 87                 ADD     A         ;TIMES 8
502C 81                 ADD     C         ;TIMES 10
502D 82                 ADD     D         ;COMBINE
502E 4F                 MOV     C,A       ;SAVE IN C
502F C31850             JMP     DBIN2     ;NEXT
                ;
                ; CHECK FOR BLANK AT END
                ;
5032 FEF0       DBIN4:  CPI     (' '-'0') AND 0FFH
5034 C23850             JNZ     ERROR     ;NOT BLANK
5037 C9                 RET
                ; PRINT ? ON IMPROPER INPUT
                ;
5038 F1         ERROR:  POP     PSW       ;RESTORE
5039 3E3F               MVI     A,'?'
503B CD0658             CALL    OUTT
503E C30050             JMP     START     ;TRY AGAIN
                ;
5041                    END
```

CONVERSION OF ASCII HEXADECIMAL CHARACTERS TO A 16-BIT BINARY NUMBER IN HL

The development of a routine to convert a string of ASCII-encoded hexadecimal characters into a 16-bit binary number will now be considered. This is the routine most frequently used in a system monitor. In fact, this was one of the first routines to be incorporated into our system monitor. Somewhere along the way there will have to be a multiplication by 16, since this is the base of the hexadecimal number. The multiplication can be easily performed by shifting the results left by four bits. Shifting left one bit is equivalent to multiplying by 2. Consequently, shifting by two bits performs a multiplication by 4.

For the binary routine we considered first, only the characters 1 and zero were valid. In the decimal routine that followed, the range of valid input was zero to 9. The hexadecimal routine we will now consider is complicated by the fact that both the digits 0–9 and the letters A–F are valid.

Two separate algorithms will be considered; one produces an 8-bit result, the other gives a 16-bit result. The program given in Listing 8.4 is similar to the previous ones. It will convert a string of ASCII-encoded hex characters into a 16-bit binary number in the H,L register pair. The double-precision add, DAD H, is used four times to perform the multiplication by 16.

Listing 8.4. ASCII hex to binary in HL.

```
                        ; THIS PROGRAM IS DESIGNED TO OPERATE
                        ; WITH THE SYSTEM MONITOR AT 5800 HEX
                        ;
                        ; JAN 18, 80
                        ;
5000                    ORG     5000H
                        ;
5800 =                  MONIT   EQU     5800H
5806 =                  OUTT    EQU     MONIT+6
580C =                  INPLN   EQU     MONIT+0CH
580F =                  GETCH   EQU     MONIT+0FH
5812 =                  OUTHX   EQU     MONIT+12H
                        ;
5000 3E0D               START:  MVI     A,0DH   ;CARR RET
5002 CD0658                     CALL    OUTT
5005 3E0A                       MVI     A,0AH   ;LINE FEED
5007 CD0658                     CALL    OUTT
500A CD0C58                     CALL    INPLN   ;GET A LINE
500D CD2150                     CALL    READHL  ;CONVERT
5010 4C                         MOV     C,H
5011 CD1258                     CALL    OUTHX   ;HIGH HALF
5014 4D                         MOV     C,L
5015 CD1258                     CALL    OUTHX   ;LOW HALF
5018 3E20                       MVI     A,' '   ;SPACE
501A CD0658                     CALL    OUTT
501D 7C                         MOV     A,H     ;HIGH BYTE
                        ;
                        ; THE NEXT INSTRUCTION IS NEEDED
                        ; WHEN LISTING 8.10 IS APPENDED
                        ;       CALL    DEC8    ;H IN DECIMAL
501E C30050                     JMP     START   ;NEXT VALUE
                        ;
                        ; READ UP TO 4 ASCII HEX DIGITS FROM
                        ; CONSOLE AND CONVERT TO 16-BIT
                        ; BINARY NUMBER IN H,L
                        ;
5021 210000             READHL: LXI     H,0     ;CLEAR
5024 CD0F58             RDHL2:  CALL    GETCH   ;GET CHAR
5027 D8                         RC              ;LINE END
5028 CD3E50                     CALL    NIB     ;TO BINARY
502B DA3750                     JC      RDHL4   ;NOT HEX
502E 29                         DAD     H       ;TIMES 2
502F 29                         DAD     H       ;TIMES 4
5030 29                         DAD     H       ;TIMES 8
5031 29                         DAD     H       ;TIMES 16
```

```
5032 B5                ORA     L         ;NEW CHAR
5033 6F                MOV     L,A
5034 C32450            JMP     RDHL2     ;NEXT
                ; CHECK FOR BLANK AT END
                ;
5037 FEF0       RDHL4: CPI     (' '-'0') AND 0FFH
5039 C0                RNZ
503A E1                POP     H         ;RAISE STACK
503B C34C50            JMP     ERROR
                ;
                ; CONVERT ASCII CHARACTERS TO BINARY
                ;
503E D630       NIB:   SUI     '0'       ;ASCII BIAS
5040 D8                RC                ; < 0
5041 FE17              CPI     'F'-'0'+1
5043 3F                CMC               ;INVERT
5044 D8                RC                ;ERROR, > F
5045 FE0A              CPI     10
5047 3F                CMC               ;INVERT
5048 D0                RNC               ;NUMBER 0-9
5049 D607              SUI     'A'-'9'-1
504B C9                RET               ;LETTER A-F
                ;
                ; PRINT ? ON IMPROPER INPUT
                ;
504C 3E3F       ERROR: MVI     A,'?'
504E CD0658            CALL    OUTT
5051 C30050            JMP     START     ;TRY AGAIN
```

Each ASCII character is converted to binary in subroutine NIB. This routine subtracts an ASCII zero, then checks to see that the character is valid. If it was originally in the range of an ASCII zero to ASCII 9, it will now be converted to the binary number 0–9. If a hex character A–F was entered, it will be converted to binary form by the additional subtraction of 7. Of course, nonhex characters will produce the error message of a question mark.

Assemble the program, load it into memory, and start it up. Type the following series of hex numbers.

```
>1
0001
>10
0010
>A
000A
>FFFF
FFFF
>12345
2345      (only last 4 characters used)
```

As with the other programs, leading zeros are not needed. If more than four characters are input, only the last four are used. Return to the monitor with a control-X.

CONVERSION OF TWO ASCII HEXADECIMAL CHARACTERS TO AN 8-BIT BINARY NUMBER IN REGISTER C

In the previous section, HL was used to convert hex characters into a binary number. But, if the HL register pair is needed as a memory pointer, then the accumulator can be used for this conversion. In this case, however, only an 8-bit binary number is produced. Listing 8.5 gives this version. Since the routine uses a fixed format, exactly two characters must be typed. This means that leading zeros must be entered.

Listing 8.5. ASCII hex to 8-bit binary C.

```
                        ; THIS PROGRAM IS DESIGNED TO OPERATE
                        ; WITH THE SYSTEM MONITOR AT 5800 HEX
                        ;
                        ; JAN 22, 80
                        ;
5000                    ORG     5000H
                        ;
5800 =                  MONIT   EQU     5800H
5806 =                  OUTT    EQU     MONIT+6
580C =                  INPLN   EQU     MONIT+0CH
580F =                  GETCH   EQU     MONIT+0FH
                        ;
                        ; REMOVE NEXT LINE WHEN LISTING 8.12 ADDED
                        ;OUTHX  EQU     MONIT+12H
                        ;
5000 3E0D               START:  MVI     A,0DH   ;CARR RET
5002 CD0658                     CALL    OUTT
5005 3E0A                       MVI     A,0AH   ;LINE FEED
5007 CD0658                     CALL    OUTT
500A CD0C58                     CALL    INPLN   ;GET A LINE
500D CD1650                     CALL    RDHEX   ;ASCII TO HEX
5010 CD4350                     CALL    OUTHX   ;BIN TO HEX
5013 C30050                     JMP     START
                        ;
                        ; CONVERT 2 ASCII-HEX CHARACTERS TO
                        ; AN 8-BIT BINARY NUMBER IN C
                        ;
5016 CD2450             RDHEX:  CALL    HEX2    ;LEFT CHAR
5019 87                         ADD     A       ;TIMES 2
501A 87                         ADD     A       ;TIMES 4
501B 87                         ADD     A       ;TIMES 8
501C 87                         ADD     A       ;TIMES 16
501D 4F                         MOV     C,A     ;SAVE
501E CD2450                     CALL    HEX2    ;RIGHT CHAR
5021 B1                         ORA     C       ;COMBINE
5022 4F                         MOV     C,A
5023 C9                         RET
                        ;
5024 CD0F58             HEX2:   CALL    GETCH   ;HEX CHAR
                        ;
                        ; CONVERT ASCII CHARACTERS TO BINARY
                        ;
```

```
5027 D630              SUI     '0'       ;ASCII BIAS
5029 DA3950            JC      ERROR     ; < 0
502C FE17              CPI     'F'-'0'+1
502E D23950            JNC     ERROR     ;ERROR, > F
5031 FE0A              CPI     10
5033 D8                RC                ;NUMBER 0-9
5034 D607              SUI     'A'-'9'-1
5036 FE0A              CPI     10
5038 D0                RNC               ;LETTER A-F
                ;
                ; PRINT ? ON IMPROPER INPUT
                ;
5039 F1         ERROR: POP     PSW       ;RAISE STACK
503A F1                POP     PSW
503B 3E3F              MVI     A,'?'
503D CD0658            CALL    OUTT
5040 C30050            JMP     START     ;TRY AGAIN
```

The multiplication by 16 is performed in the accumulator by using four ADD A instructions. The ADD A instruction is equivalent to an arithmetic shift left. Data is moved from the lower four bits to the upper four, and fills the lower bits with zero.

Assemble the program shown in the listing, load it into memory, and try it out. Remember, for this version, exactly two hex characters must be entered.

```
>01
01
>10
10
>0A
0A
>12
12
```

If everything is all right, return to the monitor with a control-X.

CONVERSION OF ASCII OCTAL CHARACTERS TO A 16-BIT BINARY NUMBER IN REGISTER HL

We did not use octal operations in the system monitor developed in Chapter 6, yet they can be very useful in trying to understand 8080 assembly language instructions. From the 8080 instruction set in Appendix D, it can be seen that the registers are assigned values as shown in the following table.

register	value
0	B
1	C
2	D
3	E
4	H
5	L
6	memory
7	A

Thus the register-move operations are obvious from the octal representation, but not from the hex or decimal form.

```
octal    hex    decimal    operation
 101      41       65      MOV   B,C
 123      53       83      MOV   D,E
 167      77      119      MOV   M,A
```

While octal numbers appear to be better than hexadecimal for expressing the 8080 operation codes, they leave something to be desired as memory pointers. The problem is that 16-bit addresses must be considered as two 8-bit bytes. But octal numbers represent groupings of three bits, and 8 is not evenly divisible by 3. An address of FFFF hex is equivalent to 177777 octal. But if this address is stored in two consecutive bytes, each byte will contain FF hex or 377 octal. This peculiarity of octal has given rise to the expression "crazy octal." A value of FFFF hex is 377:377 crazy octal.

```
hex      octal    crazy octal
  FF       377    000:377
 FFF      7777    017:377
7FFF     77777    177:377
FFFF    177777    377:377
```

Assemble the program, load it into memory, and try it out. The octal numbers input to this routine can be in the range of 0 to 177777. Try various octal numbers.

```
>0
0000
>10
0008
>20
0010
>177
007F
>377
00FF
>400
0100
>123456
A72E
>200000    (too big)
0000
```

Listing 8.6 ASCII octal to binary in HL.

```
                        ; THIS PROGRAM IS DESIGNED TO OPERATE
                        ; WITH THE SYSTEM MONITOR AT 5800 HEX
                        ;
                        ; JAN 22,79
                        ;
5000                    ORG     5000H
                        ;
5800 =                  MONIT   EQU     5800H
5806 =                  OUTT    EQU     MONIT+6
580C =                  INPLN   EQU     MONIT+0CH
580F =                  GETCH   EQU     MONIT+0FH
5812 =                  OUTHX   EQU     MONIT+12H
                        ;
5000 3E0D               START:  MVI     A,0DH   ;CARR RET
5002 CD0658                     CALL    OUTT
5005 3E0A                       MVI     A,0AH   ;LINE FEED
5007 CD0658                     CALL    OUTT
500A CD0C58                     CALL    INPLN   ;GET A LINE
500D CD2050                     CALL    RDOCT   ;OCT TO BIN
5010 4C                         MOV     C,H     ;HIGH HALF
5011 CD1258                     CALL    OUTHX   ;BIN-HEX
5014 4D                         MOV     C,L     ;LOW HALF
5015 CD1258                     CALL    OUTHX   ;BIN-HEX
5018 3E20                       MVI     A,' '   ;SPACE
501A CD0658                     CALL    OUTT
                        ;
                        ; THE NEXT INSTRUCTION IS NEEDED WHEN
                        ; THE ROUTINE IN LISTING 8.13 IS USED
                        ;       CALL    OUTOCT  ;BIN-OCT
501D C30050                     JMP     START   ;NEXT VALUE
                        ;
                        ; INPUT H,L FROM CONSOLE
                        ;
5020 D5                 RDOCT:  PUSH    D       ;SAVE REGS
5021 210000                     LXI     H,0     ;CLEAR
5024 CD0F58             OBIN2:  CALL    GETCH   ;GET CHAR
5027 DA4350                     JC      OBIN3   ;LINE END
502A D630                       SUI     '0'     ;TO BINARY
502C DA3E50                     JC      OBIN4   ; < 0
502F FE08                       CPI     8
5031 D24550                     JNC     ERROR   ; > 8
5034 29                         DAD     H       ;TIMES 2
5035 29                         DAD     H       ;TIMES 4
5036 29                         DAD     H       ;TIMES 8
5037 5F                         MOV     E,A     ;NEW BYTE
5038 1600                       MVI     D,0
503A 19                         DAD     D       ;ADD NEW BYTE
503B C32450                     JMP     OBIN2   ;NEXT
                        ;
                        ; CHECK FOR BLANK AT END
                        ;
503E FEF0               OBIN4:  CPI     (' '-'0') AND 0FFH
5040 C24550                     JNZ     ERROR   ;NOT BLANK
5043 D1                 OBIN3:  POP     D       ;RESTORE
5044 C9                         RET
                        ;
```

```
                        ; PRINT ? ON IMPROPER INPUT
                        ;
5045 E1                 ERROR:  POP     H          ;RAISE STACK
5046 E1                         POP     H
5047 3E3F                       MVI     A,'?'
5049 CD0658                     CALL    OUTT
504C C30050                     JMP     START      ;TRY AGAIN
```

This routine will convert a string of ASCII-encoded octal characters into a 16-bit binary number in the HL register pair. The current value in the HL register pair is multiplied by 8 (the number base) by performing three DAD H instructions. The new byte is converted from ASCII to binary by subtracting an ASCII zero. It is checked at this time to ensure that it is in the proper octal range of 0 to 7. The addition of the new digit to the present value is more complicated than it was for the binary or hex routines. The problem is that there can be a carry out from the low-order byte. For this reason, the new byte is placed into the DE register pair, then combined with the value in HL by using the double-precision DAD D instruction.

CONVERSION OF THREE ASCII OCTAL CHARACTERS TO AN 8-BIT BINARY NUMBER IN REGISTER C

The routine given in Listing 8.7 will convert exactly three ASCII-encoded octal characters into an 8-bit octal number in register C. The routine is programmed for fixed-format input; therefore, exactly three characters must be given. Assemble the program, load it into memory, and start it up. Type in the following octal numbers, including the leading zeros.

```
>000
00
>010
08
>020
10
>177
7F
>377
FF
```

Since the largest 8-bit number is 255 decimal or 377 octal, the first digit must be in the range 0–3. The remaining two digits must be in the range 0–7. A check is made to ensure that the characters are in the proper range. The first ASCII character in the input string is converted to binary by subtracting an ASCII zero. The result is multiplied by 8 with three ADD A instructions and the result is saved in the C register. The next character is converted to binary and added to the first with the ORA C instruction. The new sum is multiplied by 8 again with three ADD A instructions, then saved in the C register. Finally, the third character is converted to binary and added in. The final result is moved to the C register.

Listing 8.7. ASCII octal 8-bit binary in C.

```
                    ; IF THREE DIGITS, LEFT ONE MUST BE <4
                    ;
                    ; THIS PROGRAM IS DESIGNED TO OPERATE
                    ; WITH THE SYSTEM MONITOR AT 5800 HEX
                    ;
                    ; JAN 22, 80
                    ;
5000                ORG     5000H
                    ;
5800 =              MONIT   EQU     5800H
5806 =              OUTT    EQU     MONIT+6
580C =              INPLN   EQU     MONIT+0CH
580F =              GETCH   EQU     MONIT+0FH
5812 =              OUTHX   EQU     MONIT+12H
                    ;
5000 3E0D           START:  MVI     A,0DH   ;CARR RET
5002 CD0658                 CALL    OUTT
5005 3E0A                   MVI     A,0AH   ;LINE FEED
5007 CD0658                 CALL    OUTT
500A CD0C58                 CALL    INPLN   ;GET A LINE
500D CD1B50                 CALL    OCT8    ;OCT/BIN
5010 CD1258                 CALL    OUTHX   ;PRINT HEX
5013 3E20                   MVI     A,' '   ;BLANK
5015 CD0658                 CALL    OUTT
                    ;
                    ; THE NEXT INSTRUCTION IS NEEDED WHEN
                    ; THE ROUTINE IN LISTING 8.14 IS USED
                    ;       CALL    OCT     ;BIN TO ASCII
5018 C30050                 JMP     START   ;NEXT VALUE
                    ;
                    ; CONVERT ASCII-ENCODED OCTAL
                    ; TO 8-BIT BINARY NUMBER IN C
                    ;
501B CD3550         OCT8:   CALL    OCTIN   ;1ST CHAR
501E FE04                   CPI     4       ;FIRST
5020 D24750                 JNC     ERROR   ;TOO LARGE
5023 87                     ADD     A       ;TIMES 2
5024 87                     ADD     A       ;TIMES 4
5025 87                     ADD     A       ;TIMES 8
5026 4F                     MOV     C,A     ;SAVE BYTE
5027 CD3550                 CALL    OCTIN   ;2ND CHAR
502A B1                     ORA     C       ;COMBINE
502B 87                     ADD     A       ;TIMES 2
502C 87                     ADD     A       ;TIMES 4
502D 87                     ADD     A       ;TIMES 8
502E 4F                     MOV     C,A
502F CD3550                 CALL    OCTIN   ;3RD CHAR
5032 B1                     ORA     C       ;COMBINE
5033 4F                     MOV     C,A     ;SAVE IN C
5034 C9                     RET
                    ;
                    ; CONVERT INPUT CHARACTER 0-7 TO BINARY
                    ;
5035 CD0F58         OCTIN:  CALL    GETCH
5038 DA4650                 JC      ERR2    ;NO CHAR
503B D630                   SUI     '0'     ;ASCII BIAS
503D DA4650                 JC      ERR2    ;TOO SMALL
```

```
5040 FE08              CPI     8
5042 D24650            JNC     ERR2     ;TOO LARGE
5045 C9                RET
                  ;
                  ; PRINT ? ON IMPROPER INPUT
                  ;
5046 C1           ERR2:   POP     B        ;RAISE STACK
5047 C1           ERROR:  POP     B
5048 3E3F                 MVI     A,'?'
504A CD0658               CALL    OUTT
504D C30050               JMP     START    ;TRY AGAIN
```

CONVERSION OF TWO ASCII BCD DIGITS TO AN 8-BIT BINARY NUMBER IN REGISTER C

The binary coded decimal (BCD) notation was introduced in Chapter 2. This method of encoding is less efficient than the binary notation. It takes more memory space to encode numbers, and mathematical operations can be considerably slower. The BCD notation, however, has two advantages. One is that conversion from decimal to BCD is simpler than conversion from decimal to binary. The second advantage is freedom from round-off error.

The conversion routine given in Listing 8.8 accepts exactly two ASCII-encoded decimal digits and converts them into an 8-bit BCD number in the C register. The routine can be repeatedly called to convert numbers with more than two characters.

Listing 8.8. ASCII hex to BCD in C.

```
                  ; THIS PROGRAM IS DESIGNED TO OPERATE
                  ; WITH THE SYSTEM MONITOR AT 5800 HEX
                  ;
                  ; JAN 13, 80
                  ;
5000              ORG     5000H
                  ;
5800 =            MONIT   EQU     5800H
5806 =            OUTT    EQU     MONIT+6
580C =            INPLN   EQU     MONIT+0CH
580F =            GETCH   EQU     MONIT+0FH
5812 =            OUTHX   EQU     MONIT+12H
                  ;
5000 3E0D         START:  MVI     A,0DH    ;CARR RET
5002 CD0658               CALL    OUTT
5005 3E0A                 MVI     A,0AH    ;LINE FEED
5007 CD0658               CALL    OUTT
500A CD0C58               CALL    INPLN    ;GET A LINE
500D CD2050       RDHL2:  CALL    HEX2     ;LEFT CHAR
5010 87                   ADD     A        ;TIMES 2
5011 87                   ADD     A        ;TIMES 4
5012 87                   ADD     A        ;TIMES 8
5013 87                   ADD     A        ;TIMES 16
5014 4F                   MOV     C,A      ;SAVE
```

```
5015 CD2050            CALL    HEX2     ;RIGHT CHAR
5018 B1                ORA     C        ;COMBINE
5019 4F                MOV     C,A
501A CD1258            CALL    OUTHX    ;PRINT
501D C30050            JMP     START    ;NEXT
                ;
5020 CD0F58     HEX2:  CALL    GETCH    ;HEX CHAR
                ;
                ; CONVERT ASCII CHARACTERS TO BINARY
                ;
5023 D630       NIB:   SUI     '0'      ;ASCII BIAS
5025 DA2B50            JC      ERROR    ; < 0
5028 FE0A              CPI     10
502A D8                RC               ;NUMBER 0-9
                ;
                ; PRINT ? ON IMPROPER INPUT
                ;
502B 3E3F       ERROR: MVI     A,'?'
502D CD0658            CALL    OUTT
5030 C30050            JMP     START    ;TRY AGAIN
                ;
5033                   END
```

With the BCD notation, there is a 2:1 correspondence between the number of BCD digits and the necessary number of bytes. Or, put another way, two decimal digits are stored in each byte. This arrangement is sometimes called *packed decimal.* The right decimal digit is encoded in the low-order four bits and the left digit is encoded into the high-order four bits.

38	0011 1000
27	0010 0111
59	0101 1001

BCD encoding involves essentially the same steps as does the hex-to-binary routine, except that only the characters 0 through 9 are allowed. The bit patterns corresponding to the hexadecimal numbers A through F are not allowed.

CONVERSION OF AN 8-BIT BINARY NUMBER IN C TO A STRING OF EIGHT ASCII BINARY CHARACTERS

In the first part of this chapter we developed programs to convert strings of ASCII-encoded characters into binary numbers. The programs in the following sections will perform the reverse operation. Binary numbers will be converted into strings of ASCII-encoded characters. Furthermore, we will combine the new routines with those already developed so that they may be more easily tested.

The program shown in Listing 8.9 can be used to convert an 8-bit binary number in register C into eight ASCII-encoded binary characters. The resulting characters are sent to the console in this case, but they could be placed into sequential memory locations instead. This routine is incorporated into the system monitor developed in Chapters 6 and 7.

Listing 8.9 Binary in C to ASCII binary.

```
                    ; THIS PROGRAM IS DESIGNED TO RUN WITH
                    ; THE SYSTEM MONITOR AND WITH THE
                    ; PROGRAM SHOWN IN LISTING 8.1
                    ;
                    ; PRINT AN 8-BIT BINARY NUMBER IN C AS
                    ; 8 ASCII-ENCODED BITS
                    ;
504A 0608           BITS:   MVI     B,8       ;8 BITS
504C 79             BIT2:   MOV     A,C       ;GET BYTE
504D 87                     ADD     A         ;SET CARRY
504E 4F                     MOV     C,A       ;PUT BACK
504F 3E18                   MVI     A,'0'/2   ;HALF OF 0
5051 8F                     ADC     A         ;DOUBLE+CARRY
5052 CD0658                 CALL    OUTT      ;ONE BIT
5055 05                     DCR     B         ;COUNT
5056 C24C50                 JNZ     BIT2      ;8 TIMES
5059 C9                     RET
                    ;
505A                        END
```

Make a duplicate copy of the source program shown in Listing 8.1. This routine was used to convert ASCII-encoded binary characters into an 8-bit binary number in C. Remove the semicolon from the instruction that reads

```
;       CALL    BITS    ;BIN TO ASCII
```

Also delete the END directive if you used one. Add the lines given in Listing 8.9 to the end of the program.

The new routine works in the following way. Register B is initialized with the value of 8, the number of bits to be generated. Register C begins with the original binary byte. The bits of this byte are shifted to the left one at a time into the carry flag. The carry flag is then added to an ASCII zero to produce a 0 or a 1 at the console. For the 8080 version, the byte is moved to the accumulator, shifted left with an add instruction, then returned to register C.

If the current high-order bit is a zero, then the carry flag is reset and a zero is printed. If the current high-order bit is a 1, then the carry flag is set and a 1 is printed. The count in the B register is decremented after each character is printed. When the count reaches zero, the process is terminated.

Notice that the addition of the carry flag to the ASCII zero could have been accomplished with the instructions

```
MVI     A,'0'
ACI     0
```

However, this will require four bytes. The code we will use, which is not so obvious, requires only three bytes.

```
MVI     A,'0'/2
ADC     A
```

If you have a Z-80 CPU, this routine can be simplified and shortened by three bytes. Performing the rotation directly in the C register reduces the program by one byte. Two additional bytes are gained by using the decrement B, jump-not-zero operation. Now the accumulator can be zeroed with an exclusive-or operation and the carry can be added directly to an ASCII zero.

```
BITS:   LD      B,8     ;8 BITS
BIT2:   XOR     A       ;ZERO A
        SLA     L       ;SHIFT L LEFT
        ADC     A,'0'   ;ADD CARRY TO 0
        CALL    OUTT    ;SEND
        DJNZ    BIT2    ;8 TIMES
        RET
```

Assemble the combined program, load it into memory, and start it up. Be sure that the monitor is in place at the address of MONIT. The new binary-to-ASCII binary routine has been added in addition to the original binary to ASCII hex in the system monitor (OUTHX). Consequently, each number will now be rendered in both hex and binary. Type in the following binary numbers.

```
>0                  (you type this)
00 00000000         (both hex and binary are given)
>1
01 00000001
>101
05 00000101
>10101010
AA 10101010
>11110000
F0 11110000
```

Remember that the error-correction features of the monitor are available. If you inadvertently type a control-X, you will end up in the monitor itself. You can return from the monitor to the new program, however, by typing

```
>G5000
```

if this is where you assembled the new routine.

CONVERSION OF AN 8-BIT BINARY NUMBER INTO THREE ASCII DECIMAL CHARACTERS

An 8-bit binary number can be converted into a string of ASCII-encoded decimal characters by repeated subtraction of powers of 10, the number base. The corresponding decimal number will lie in the range of zero to 255. The value of 100 (decimal) is repeatedly subtracted from the original binary number until the result becomes negative. One less than the number of subtractions is the number of hundreds in the decimal form. The result, which in this case can only be 0, 1, or 2, is added to the value of an ASCII zero, then sent to the console.

As an example of the 100s subtraction, consider the number 137.

```
 137
-100  (one subtraction)
  37
-100  (too many)
(negative number)
```

Since there was one subtraction of 100 before the number became negative, the number is in the range 100 to 199.

The value of the last 100 is added back to make the number positive again. Then the value of 10 is repeatedly subtracted from the new value until a negative result is again obtained. The number of tens in the decimal form is one smaller than the number of subtractions. This count is added to an ASCII zero and sent to the console for the middle digit. The value of ten is added back to the remainder. This adjusted remainder is then added to an ASCII zero to produce the units digit. It too is sent to the console.

Continuing with the example:

```
(negative number)
+100  (add back last 100)
  37
 -10  (first subtraction)
  27
 -10  (2nd subtraction)
  17
 -10  (3rd subtraction)
   7
 -10  (too many)
(negative number)
 +10  (add back last 10)
   7  (units)
```

Since there were three subtractions of 10 before the remainder became negative, the middle digit is a 3. Finally, the right digit is 7, the remainder after the last subtraction of 10.

Lisiting 8.10 gives the instructions for the conversion of an 8-bit binary number in register H. Register D initially contains the value of 100 (decimal) that is to be repeatedly subtracted from the number in H. As soon as the result of the subtraction becomes negative, the last 100 is added back, and the value in D is changed to a 10 by subtraction of 90. Register C is used to count the number of subtractions. It is initialized with the value of one less than an ASCII zero to simplify the conversion.

Listing 8.10. Binary in A to ASCII decimal.

```
                  ; THIS PROGRAM IS DESIGNED TO RUN WITH
                  ; THE SYSTEM MONITOR AND WITH THE
                  ; PROGRAM SHOWN IN LISTING 8.4
                  ;
                  ; FEB 27, 80
                  ;
                  ;
                  ; PRINT BINARY NUMBER IN A AS ASCII
                  ; DECIMAL DIGITS. LEADING ZEROS SUPPRESSED
                  ;
5057 D5           DEC8:   PUSH    D
5058 C5                   PUSH    B
5059 1E00                 MVI     E,0        ;LEADING 0 FLAG
505B 1664                 MVI     D,100
505D 0E2F         DEC81:  MVI     C,'0'-1
505F 0C           DEC82:  INR     C
5060 92                   SUB     D          ;100 OR 10
5061 D25F50               JNC     DEC82      ;STILL +
5064 82                   ADD     D          ;ADD BACK
5065 47                   MOV     B,A        ;REMAINDER
5066 79                   MOV     A,C        ;GET 100/10
5067 FE31                 CPI     '1'        ;ZERO?
5069 D27250               JNC     DEC84      ;YES
506C 7B                   MOV     A,E        ;CHECK FLAG
506D B7                   ORA     A          ;RESET?
506E 79                   MOV     A,C        ;RESTORE BYTE
506F CA7750               JZ      DEC85      ;LEADING ZERO
5072 CD0658       DEC84:  CALL    OUTT       ;PRINT IT
5075 1EFF                 MVI     E,0FFH     ;SET 0 FLAG
5077 7A           DEC85:  MOV     A,D
5078 D65A                 SUI     90         ;100 TO 10
507A 57                   MOV     D,A
507B 78                   MOV     A,B        ;REMAINDER
507C D25D50               JNC     DEC81      ;AGAIN
507F C630                 ADI     '0'        ;ASCII BIAS
5081 CD0658               CALL    OUTT
5084 C1                   POP     B
5085 D1                   POP     D
5086 C9                   RET
                  ;
5087                      END
```

There is an additional feature added to this routine: leading-zero suppression. Leading zeros are typically suppressed when a number is expressed in decimal form. On the other hand, leading zeros are commonly left in place for binary, octal, and hexadecimal numbers. Thus, we write

```
  1
  2
 18
197
207
```

for decimal numbers, but

```
00001111  (binary)
     040  (octal)
    0FE3  (hexadecimal)
```

for the others.

One reason for keeping leading zeros is to make it easier to distinguish octal or hex numbers from decimal numbers. With this convention, the number 0137 would be interpreted as an octal or hex value rather than a decimal number. The suppression of leading zeros is not a difficult task, but it does require some additional code. It is not sufficient to merely remove all zeros, for then the number 0307 becomes 37.

One technique is to utilize a zero-suppression flag which is initially reset. Zeros are omitted as long as the flag remains reset. Then the flag is set when the first nonzero character is encountered. Subsequent zeros are not removed since the flag is set.

The necessary zero-suppression code has been included in the routine shown in Listing 8.10. Register E is used for the flag; it is initially reset. Then each character is checked with a CPI '1' instruction to see if it is an ASCII zero. If the value is not a zero, it is printed and the flag is set to FF hex. On the other hand, if the digit is a zero, the flag is checked. If it is found to be reset, the zero is not printed. Only three decimal characters can be produced from the original byte; consequently, just the first two need to be checked. If the value of the byte is zero, a single ASCII zero is printed.

The B, C, D, and E registers are utilized in the operations. Consequently, the original values are initially saved on the stack, then restored at the conclusion of the routine.

Duplicate the program shown in Listing 8.4, the ASCII-hex to binary routine. Keep one of the copies as a backup, then rename the other one HEXDEC.ASM. Remove the semicolon at the beginning of the line

```
;       CALL    DEC8
```

and remove the END directive at the end of the program. Add the lines given in Listing 8.10. Assemble the combination program, load it into memory, and start it up by branching to the address of START. We have coupled the 16-bit hex input program with the 8-bit decimal output program. Consequently, only the high-order byte will be converted to decimal. This is the byte in the H register. Try the combined program with the following hex numbers.

```
>0
0000 0
>1
0001 0
>100
0100 1
>A00
0A00 10   (hex A is decimal 10)
>1000
1000 16   (10 hex is 16 decimal)
>FFFF
FFFF 255   (FF hex is 255 decimal)
```

This binary-to-decimal routine can be very useful at the CP/M systems level. Suppose that you want to save a program that starts at 100 hex and runs to 2736 hex. If the decimal routine is incorporated into a hexadecimal math routine of the system monitor, you have only to type

```
>H2800 100        (command)
2900 2700 39     (response)
```

The response of 39 is the decimal number of 256-byte blocks to be saved. Then you can give the CP/M command

```
A>SAVE 39   FILENAME.EXT
```

CONVERSION OF A 16-BIT BINARY NUMBER INTO FIVE ASCII DECIMAL CHARACTERS

In the previous section, we developed a routine to convert an 8-bit binary number into three ASCII-encoded decimal characters. We will now write a routine for converting a 16-bit binary number in HL into 5 ASCII-encoded decimal characters. This double-precision decimal number will range from zero to 65,535.

Duplicate the decimal-to-16-bit binary program in Listing 8.2. Remove the semicolon from the line

```
;         CALL     DBIN
```

and the END directive if you used one. Add the program shown in Listing 8.11 to the end.

Listing 8.11 Binary in HL to ASCII decimal.

```
                    ; THIS PROGRAM IS DESIGNED TO RUN WITH
                    ; THE SYSTEM MONITOR AND WITH THE
                    ; PROGRAM SHOWN IN LISTING 8.2
                    ;
                    ; FEB 22, 79
                    ;
                    ; PRINT BINARY NUMBER IN H,L AS ASCII
                    ; DECIMAL DIGITS. LEADING ZERO SUPPRESSED.
                    ;
5055 0600           BIND:   MVI     B,0       ;LEADING 0 FLAG
5057 11F0D8                 LXI     D,-10000  ;2'S COMPL
505A CD7550                 CALL    SUBTR     ;10 THOUS
505D 1118FC                 LXI     D,-1000
5060 CD7550                 CALL    SUBTR     ;THOUS
5063 119CFF                 LXI     D,-100
5066 CD7550                 CALL    SUBTR     ;HUNDREDS
5069 11F6FF                 LXI     D,-10
506C CD7550                 CALL    SUBTR     ;TENS
506F 7D                     MOV     A,L
5070 C630                   ADI     '0'       ;ASCII BIAS
5072 C30658                 JMP     OUTT      ;UNITS
                    ;
                    ; SUBTRACT POWER OF TEN AND COUNT
                    ;
5075 0E2F           SUBTR:  MVI     C,'0'-1   ;ASCII COUNT
5077 0C             SUBT2:  INR     C
5078 19                     DAD     D         ;ADD NEG NUMBER
5079 DA7750                 JC      SUBT2     ;KEEP GOING
                    ;
                    ; ONE TOO MANY, ADD ONE BACK
                    ;
507C 7A                     MOV     A,D       ;COMPLEMENT
507D 2F                     CMA               ; D,E
507E 57                     MOV     D,A
507F 7B                     MOV     A,E
5080 2F                     CMA
5081 5F                     MOV     E,A
5082 13                     INX     D         ;AND
5083 19                     DAD     D         ;ADD BACK
5084 79                     MOV     A,C       ;GET COUNT
                    ;
                    ; CHECK FOR ZERO
                    ;
5085 FE31                   CPI     '1'       ;LESS THAN 1?
5087 D29150                 JNC     NZERO     ;NO
508A 78                     MOV     A,B       ;CHECK 0 FLAG
508B B7                     ORA     A         ;SET?
508C 79                     MOV     A,C       ;RESTORE
508D C8                     RZ                ;SKIP LEADING 0
508E C30658                 JMP     OUTT      ;INTERIOR ZERO
                    ;
                    ; SET FLAG FOR NON-ZERO CHARACTER
                    ;
5091 06FF           NZERO:  MVI     B,0FFH    ;SET 0 FLAG
5093 C30658                 JMP     OUTT      ;PRINT IT
                    ;
5096                        END
```

This 16-bit version is similar to the 8-bit version of the previous section. This time we start with the subtraction of the decimal value 10,000 instead of 100. The number of subtractions is counted as before. When the result becomes negative, the last 10,000 is added back. The number of subtractions that was performed is the desired digit for the ten-thousandths position.

Since the 8080 CPU does not incorporate a 16-bit subtraction operation, the subtraction is obtained by adding the corresponding two's complement

```
LXI     D,-10000
. . .
. . .
DAD     D
```

The carry flag will be set for each addition (subtraction) except the last. The carry flag is then reset for the operation corresponding to the result becoming negative. The JC instruction in subroutine SUBTR causes the computer to loop the correct number of times. Suppose, for example, that the binary number in HL corresponds to the decimal number 32,128. This is the equivalent of the hexadecimal value 7D80. The two's complement of 10,000 is D8F0 hex. The sum of these two is

```
 decimal    hexadecimal
 32,128      7D80
-10,000     +D8F0
-------      ----
 22,128      5670
```

Thus the addition of 7D80 and D8F0 hex is equivalent to subtracting 10,000 from 32,128.

The last subtraction that causes the result to become negative has to be undone. This could be accomplished by adding 10,000. However, the addition is accomplished in subroutine SUBTR which is also used to add back the 1,000, 100, and 10 for the equivalent steps of each decade. Therefore, we won't know, in general, which value to add. The solution is to obtain the necessary value from the two's complement of the two's complement for the current value. This two's complement is first obtained by complementing both the D and the E register to produce the one's complement. Then the DE register pair is incremented. Subroutine SUBTR is first called with DE set to −10,000, then to −1,000, −100, and −10. At this point, the units digit is contained in the L register.

This double-precision routine also incorporates instructions for suppressing leading zeros. The approach is similar to the one used in the previous section except that the H register is used for the zero flag.

Assemble the combination program, load it into memory, and start it up. Enter the following decimal numbers. The response will include both the hexadecimal and the decimal values of the input number.

```
>0
0000 0
>1
0001 1
>100
0064 100
>1024
0400 1024
>65535
FFFF 65535
```

At this point you may want to incorporate decimal numbers into the system monitor. When we incorporated an ASCII-input feature into the monitor we used the apostrophe to indicate this fact. It is customary to precede decimal numbers by a number sign. For example, the following input would indicate decimal input.

```
>H#1024 #200
```

CONVERSION OF AN 8-BIT BINARY NUMBER INTO TWO ASCII HEXADECIMAL CHARACTERS

The conversion of an 8-bit binary byte into two ASCII-encoded hexadecimal characters is an interesting exercise. The high-order four bits are represented by one hex character, and the low-order four bits are represented by the other hex character. Since the upper character is printed first, the upper four bits are rotated down to the lower position. These new lower four bits are converted to ASCII to produce the first character. Then the original low-order four bits are converted to ASCII to get the second character.

The nibble conversion appears to be straightforward. The upper four bits are zeroed by performing a masking AND with the value of 0F hex. The lower four bits are then converted to ASCII by the addition of an ASCII 0. If the result is in the range 0–9, then the conversion is complete. Otherwise, a binary 7 is added to the result, converting it to an ASCII-encoded letter of A through F.

To see why this conversion works, look at the ASCII table in Appendix A. The ASCII number 9 has a decimal value of 57. The ASCII letter A has a decimal value of 65. But the hexadecimal value of A follows the value of 9. Therefore, we need to add 7 to any valid hex number larger than 9 to convert it into the appropriate ASCII letter A through F.

The program shown in Listing 8.12 will convert an 8-bit binary number in the C register into two ASCII characters and send them to the console. Duplicate the program in Listing 8.5. Remove the external reference to subroutine OUTHX near the beginning.

```
OUTHX    EQU     MONIT+12H
```

Our new program will perform the same function. Also remove the final END directive if you used one. Copy the lines from Listing 8.12 on the end of the program. Assemble the combined program, load it into memory, and start it up. Our monitor will still have to be in memory since we will need the I/O routines. This current version, Listing 8.12, with the built-in binary-to-hex routine, should respond exactly the same as the earlier version, Listing 8.5. The only difference is that we are converting binary to hex within the program rather than using a routine in the monitor.

Listing 8.12 Binary in C to ASCII hex

```
                 ; THIS PROGRAM IS DESIGNED TO RUN WITH
                 ; THE SYSTEM MONITOR AND WITH THE
                 ; PROGRAM SHOWN IN LISTING 8.5
                 ;
                 ; JAN 18, 80
                 ;
                 ; ROUTINE TO OUTPUT 2 HEX CHARACTERS FROM C
                 ; 8-BIT BINARY TO ASCII HEX CONVERSION
                 ;
5043 79          OUTHX:  MOV     A,C      ;GET BYTE
5044 1F                  RAR              ;ROTATE
5045 1F                  RAR              ; FOUR
5046 1F                  RAR              ; BITS TO
5047 1F                  RAR              ; THE RIGHT
5048 CD4C50              CALL    HEX1     ;UPPER CHAR
504B 79                  MOV     A,C      ;LOWER CHAR
                 ;
504C E60F        HEX1:   ANI     0FH      ;LOW 4 BITS
504E C630                ADI     '0'      ;ASCII ZERO
5050 FE3A                CPI     '9'+1    ;0 TO 9
5052 DA0658              JC      OUTT     ;YES
5055 C607                ADI     7        ;CONVERT
5057 C30658              JMP     OUTT     ;A TO F
                 ;
505A                     END
```

The algorithm used to convert the binary nibble to a hex character clearly demonstrates the technique. But there is a more efficient method for CPUs such as the 8080 and Z-80 that incorporate the decimal adjust accumulator (DAA) instruction. Change the second, third, fourth, and fifth lines of subroutine HEX1 so the subroutine looks like

```
HEX1:   ANI     0FH     (same)
        ADI     90H     (new)
        DAA             (new)
        ACI     40H     (new)
        DAA             (new)
        JMP     OUTT    (same)
```

Conversion of a 16-bit binary value into four hex characters is obtained by calling the 8-bit routine twice. For example, the HL register pair can be printed with the following routine.

```
OUTHL:  MOV     C,H
        CALL    OUTHX
        MOV     C,L
        CALL    OUTHX
```

Conversion of a binary number to a string of ASCII-encoded BCD characters does not require a special routine. It is performed simply by calling the binary-to-hex routine OUTHX.

CONVERSION OF A 16-BIT BINARY NUMBER INTO SIX ASCII OCTAL CHARACTERS

A 16-bit binary number in the HL register pair can be converted into six ASCII-encoded octal characters by repeatedly shifting the double register to the left. We make use of the double-register add instruction DAD H. This instruction adds the register pair HL to itself. The operation is equivalent to a double-precision arithmetic shift left. All 16 bits are shifted left and a zero is moved into the lowest order bit. The carry flag is set if the original highest-order bit was a logical one. The carry bit is reset otherwise.

The routine is shown in Listing 8.13. As the bits of the double register are shifted out the high end, they are converted to the corresponding octal number in the accumulator. The largest 16-bit octal number is 177777; consequently, the first octal character (starting from the left) can only be a zero or a one. The remaining five characters can range from zero through 7. They are obtained from groups of three bits.

Listing 8.13 Binary in HL to ASCII octal.

```
                ; THIS PROGRAM IS DESIGNED TO RUN WITH
                ; THE SYSTEM MONITOR AND WITH THE
                ; PROGRAM SHOWN IN LISTING 8.6
                ;
                ; JAN 18, 80
                ;
                ; ROUTINE TO OUTPUT A 16-BIT BINARY
                ; VALUE IN H,L AS OCTAL CHARACTERS
                ;
5052 E5         OUTOCT: PUSH    H       ;SAVE VALUE
5053 C5                 PUSH    B
5054 0605               MVI     B,5     ;5 CHAR
5056 AF                 XRA     A       ;ZERO
5057 29                 DAD     H       ;HIGH BIT
5058 CE30               ACI     '0'     ;ADDED IN
505A CD0658             CALL    OUTT    ;PRINT IT
505D 3E06       OCT2:   MVI     A,6     ;ROTATE 60Q
505F 29                 DAD     H       ;CARRY
5060 17                 RAL             ;ROTATE TO A
5061 29                 DAD     H       ;AGAIN
5062 17                 RAL             ;TO A
5063 29                 DAD     H       ;3RD TIME
5064 17                 RAL             ;TO A
5065 CD0658             CALL    OUTT    ;PRINT CHAR
5068 05                 DCR     B       ;COUNT
5069 C25D50             JNZ     OCT2    ;5 TIMES
506C C1                 POP     B
506D E1                 POP     H
506E C9                 RET
                ;
506F                    END
```

The first step shown in Listing 8.13 saves the HL and BC registers on the stack. This operation may not always be necessary. The XRA operation is used to zero the accumulator. It must be performed before the DAD H instruction since it resets the carry flag. The DAD instruction will set the carry flag if the high-order bit was a 1, or reset the flag if the high-order bit was a zero. The carry flag is then added to an ASCII zero and sent to the console.

The remaining five digits are generated in a loop starting at label OCT2. Register B is preloaded with the value of 5 to count the remaining digits. We need to transfer the current three high-order bits from the H register into the accumulator. This is accomplished by three DAD H instructions. After each one, the carry flag will reflect the state of the most recent high-order bit of H. After each DAD instruction, we perform a rotate accumulator instruction. This moves the carry bit into the low-order bit of A. After three such operations, the three low-order bits of the accumulator will contain the proper octal bit pattern for the appropriate character. But they are in binary form and we need to convert them to ASCII for printing.

The bit pattern for the ASCII zero is 011 0000. Notice that it contains zeros in the lower four bit positions. At the beginning of the OCT2 loop we preloaded the accumulator with a binary 6 which has a bit pattern of 000 0110. At the conclusion of the OCT2 loop, the three rotations will convert this binary 6 into a 60 octal which is equivalent to an ASCII zero. This effectively converts the three lower-order bits, shifted in from the carry flag, into ASCII-encoded octal.

The Z-80 version can be a little shorter if the instruction

```
DJNZ    OCT2
```

is used in place of

```
DCR     B
JNZ     OCT2
```

CONVERSION OF AN 8-BIT BINARY NUMBER INTO THREE ASCII OCTAL CHARACTERS

In the previous section we derived a subroutine for the conversion of a 16-bit binary number into ASCII characters. The conversion of an 8-bit binary number to octal will be considered here. We could use the H,L double register, as previously, to perform the necessary shifts. However, if the H,L register is needed for something else, such as a pointer to memory, then another method would be better.

The routine shown in Listing 8.14 performs the needed rotations in the accumulator. The original binary byte is in the C register. The byte is moved to the accumulator and two left circular rotate instructions are performed. This effectively moves the two high-order bits down to the two low-order positions. It is equivalent to performing six right circular rotations. The remaining bits are zeroed with a logical AND 3 step. The ubiquitous ASCII zero is added, and the result is sent to the console.

The middle character is obtained by rotating the original byte to the right. A logical AND 7 isolates these low-order bits. The ASCII zero is added and the byte is sent to the console. The original byte is retrieved a third time. Now the three bits of the third character are properly positioned. Hence no rotation is needed. The masking AND operation is still needed, however. Finally, the ASCII zero is added prior to printing.

Type the program shown in Listing 8.14. Add the new program to the end of the octal-to-binary program shown in Listing 8.7. Assemble the combination, load it into memory, and branch to the beginning. Remember that the input requires exactly three octal characters. If less than three are used, an error message of a question mark will be printed. If more than three characters are typed, only the first three will be used.

Listing 8.14. 8-bit binary in C to ASCII

```
                   ; THIS PROGRAM IS DESIGNED TO RUN WITH
                   ; THE SYSTEM MONITOR AND WITH THE
                   ; PROGRAM SHOWN IN LISTING 8.7
                   ;
                   ; JAN 22, 80
                   ;
                   ; ROUTINE TO OUTPUT AN 8-BIT BINARY
                   ; VALUE AS THREE OCTAL CHARACTERS
                   ;
5053 79            OCT:    MOV     A,C     ;GET IT
5054 07                    RLC             ;2 HIGH BITS
5055 07                    RLC
5056 E603                  ANI     3       ;MASK
5058 CD6550                CALL    OCT3
505B 79                    MOV     A,C     ;GET AGAIN
505C 0F                    RRC             ;MIDDLE BITS
505D 0F                    RRC
505E 0F                    RRC
505F CD6350                CALL    OCT2
5062 79                    MOV     A,C     ;RIGHT BITS
5063 E607          OCT2:   ANI     7       ;3 BITS
5065 C630          OCT3:   ADI     '0'     ;ASCII BIAS
5067 C30658                JMP     OUTT    ;PRINT
                   ;
506A                       END
```

CONVERSION OF A 16-BIT BINARY NUMBER TO SPLIT OCTAL

Some of the 8080 and Z-80 instructions can have either 8-bit or 16-bit operands. Typical 8-bit operations for the 8080 are

```
MVI     A,10
ADI     3
ANI     7
```

The 16-bit operations include

```
CALL    100H
JMP     5005H
LXI     H,0
```

But the 8-bit architecture of microcomputers means that 16-bit operands are stored as two consecutive bytes. Each byte can be represented as two hexadecimal characters. And the entire 16-bit value can be represented by a combination of all four of these hex characters. For example, the 16 bits

11110000 10101010

can be represented with the hex characters

F0AA

Alternately, the left byte can be represented by an F0 hex, and the right byte can be expressed as AA hex. There is no problem here since each hex character represents four bits, and both 8 and 16 are evenly divisible by 4.

Octal representation is more complex. In this case, each octal character represents three bits (or sometimes two). Since neither 8 nor 16 is evenly divisible by 3, a problem can occur. Consider the 16-bit value

1 111 010 011 101 010

It can be represented in the octal notation as

172352

This is the result that the program in Listing 8.13 would produce at the system console. The problem is that the 16 bits are actually stored in two adjacent 8-bit locations. If the bit pattern is grouped into 8-bit bytes, it looks like this.

11 110 100   11 101 010

In this arrangement, the two corresponding octal bytes are represented as

364 352

The result, which has been termed split octal or crazy octal, looks very different from the corresponding 16-bit octal value of 172352. The two octal bytes of split-octal notation are sometimes separated by a colon to distinguish the representation from the regular 16-bit octal.

364:352

The split-octal notation can be readily implemented, as shown in Listing 8.15. This program is adapted from the 8-bit octal-to-binary and binary-to-octal routines given in Listings 8.7 and 8.14. The first part takes two separate octal bytes and converts them to an 8-bit binary number. The second part converts the binary byte into two octal bytes. Each split-octal number is entered from the console as two groups of three characters. A space or colon can be used to separate the two parts.

Listing 8.15. Split-octal routines

```
                ; 6 OCTAL CHARACTERS SEPARATED BY A
                ; SPACE ARE ENTERED FROM THE CONSOLE.
                ; A 16-BIT BINARY NUMBER IS PRODUCED
                ; IN D,E. THIS IS RECONVERTED TO OCTAL
                ;
                ; THIS PROGRAM IS DESIGNED TO OPERATE
                ; WITH THE SYSTEM MONITOR AT 5800 HEX
                ;
                ; JAN 22, 80
                ;
5000            ORG     5000H
                ;
5800 =          MONIT   EQU     5800H
5806 =          OUTT    EQU     MONIT+6
580C =          INPLN   EQU     MONIT+0CH
580F =          GETCH   EQU     MONIT+0FH
5812 =          OUTHX   EQU     MONIT+12H
                ;
5000 3E0D       START:  MVI     A,0DH   ;CARR RET
5002 CD0658             CALL    OUTT
5005 3E0A               MVI     A,0AH   ;LINE FEED
5007 CD0658             CALL    OUTT
500A CD0C58             CALL    INPLN   ;GET A LINE
500D CD2350             CALL    OCT88   ;6 OCT CHAR
5010 4A                 MOV     C,D     ;FIRST
5011 CD1258             CALL    OUTHX   ;TO HEX
5014 4B                 MOV     C,E     ;SECOND
5015 CD1258             CALL    OUTHX   ;TO HEX
5018 3E20               MVI     A,' '   ;BLANK
501A CD0658             CALL    OUTT
501D CD6850             CALL    OUT80   ;TO BINARY
5020 C30050             JMP     START   ;NEXT VALUE
                ;
                ; 6 SPLIT-OCTAL CHAR TO 16-BIT BINARY
                ;
5023 CD3250     OCT88:  CALL    OCT8    ;FIRST
5026 51                 MOV     D,C     ;SAVE
5027 CD0F58             CALL    GETCH   ;SPACE
502A DA5F50             JC      ERR3    ;ONLY 1 CHAR
502D CD3250             CALL    OCT8    ;SECOND
5030 59                 MOV     E,C     ;SAVE
5031 C9                 RET
                ;
                ; CONVERT ASCII-ENCODED OCTAL
                ; TO 8-BIT BINARY NUMBER IN C
                ;
```

```
5032 CD4C50     OCT8:    CALL     OCTIN    ;1ST CHAR
5035 FE04                CPI      4        ;FIRST
5037 D25E50              JNC      ERROR    ;TOO LARGE
503A 87                  ADD      A        ;TIMES 2
503B 87                  ADD      A        ;TIMES 4
503C 87                  ADD      A        ;TIMES 8
503D 4F                  MOV      C,A      ;SAVE BYTE
503E CD4C50              CALL     OCTIN    ;2ND CHAR
5041 B1                  ORA      C        ;COMBINE
5042 87                  ADD      A        ;TIMES 2
5043 87                  ADD      A        ;TIMES 4
5044 87                  ADD      A        ;TIMES 8
5045 4F                  MOV      C,A
5046 CD4C50              CALL     OCTIN    ;3RD CHAR
5049 B1                  ORA      C        ;COMBINE
504A 4F                  MOV      C,A      ;SAVE IN C
504B C9                  RET
                ;
                ; CONVERT INPUT CHARACTER 0-7 TO BINARY
                ;
504C CD0F58     OCTIN:   CALL     GETCH
504F DA5D50              JC       ERR2     ;NO CHAR
5052 D630                SUI      '0'      ;ASCII BIAS
5054 DA5D50              JC       ERR2     ;TOO SMALL
5057 FE08                CPI      8
5059 D25D50              JNC      ERR2     ;TOO LARGE
505C C9                  RET
                ;
                ; PRINT ? ON IMPROPER INPUT
                ;
505D C1         ERR2:    POP      B        ;RAISE STACK
505E C1         ERROR:   POP      B
505F C1         ERR3:    POP      B
5060 3E3F                MVI      A,'?'
5062 CD0658              CALL     OUTT
5065 C30050              JMP      START    ;TRY AGAIN
                ;
                ; 16-BIT BINARY IN D,E TO SPLIT OCTAL
                ;
5068 4A         OUT80:   MOV      C,D      ;FIRST BYTE
5069 CD7250              CALL     OCT      ;TO OCT
506C 3E3A                MVI      A,':'
506E CD0658              CALL     OUTT     ;SEPARATOR
5071 4B                  MOV      C,E      ;SECOND
                ;
                ; ROUTINE TO OUTPUT AN 8-BIT BINARY
                ; VALUE AS THREE OCTAL CHARACTERS
                ;
5072 79         OCT:     MOV      A,C      ;GET IT
5073 07                  RLC               ;2 HIGH BITS
5074 07                  RLC
5075 E603                ANI      3        ;MASK
5077 CD8450              CALL     OCT3
507A 79                  MOV      A,C      ;GET AGAIN
507B 0F                  RRC               ;MIDDLE BITS
507C 0F                  RRC
507D 0F                  RRC
507E CD8250              CALL     OCT2
```

```
5081 79                 MOV     A,C     ;RIGHT BITS
5082 E607       OCT2:   ANI     7       ;3 BITS
5084 C630       OCT3:   ADI     '0'     ;ASCII BIAS
5086 C30658             JMP     OUTT    ;PRINT
                ;
5089                    END
```

The input number is converted into a 16-bit binary number in the D,E register pair by calling the 8-bit routine twice. The hex value is printed out as usual, then the split-octal version. A colon in the middle separates the two halves.

Assemble the program and try it out.

```
>177 377  (a space separator)
7FFF 177:377
>100:200  (a colon separator)
4080 100:200
```

This completes the base-conversion routines. At this point, you may want to incorporate some of these routines into your system monitor.

CHAPTER NINE

Paper Tape and Magnetic Tape Routines

It is possible to encode complete programs, such as a BASIC interpreter, into ROM so that calculations, such as the square root of 19, can be performed as soon as the computer is turned on. But more complicated problems will require a source program. If the source program is used frequently, then it would be inconvenient to reenter the source program each time it is needed. One way to avoid this reentry problem is to save the source program somehow and then reload it into the computer when it is needed.

But BASIC is not the only computer language. Some tasks are more easily performed with other languages, such as Pascal or APL. Assembly language is useful for systems programming. A full text editor program has more features than the most complex BASIC interpreter. Thus, it is better not to have the BASIC interpreter in ROM. Instead, a small monitor program, such as the one developed in Chapter 6, can be placed in ROM. We can use this small monitor to load larger programs from an external medium.

The floppy disk is a convenient medium for saving and reloading programs. The cost, however, is greater than other storage media such as paper tape or magnetic tape. Even if a floppy-disk system is utilized for program storage, it might be wise to make backup copies on magnetic or paper tape.

The simplest storage method is to utilize the paper tape accessory available on some Teletype machines. With this approach, a separate computer I/O port is unnecessary. Furthermore, paper tapes can be read directly on the Teletype, without using a computer. The disadvantage of this method is that the transfer rate is low, since the Teletype operates at only ten characters per second.

A more complicated, but faster, method utilizes an ordinary magnetic tape machine designed for home recording. In this case a separate I/O port will usually be necessary, but the advantage is that the recording rate is higher than for paper tape. Common data transfer rates range from 30 to over 120 characters per second.

The frequency response of audio tape recorders is limited to a maximum of 10 to 20 kHz. But since the computer operates at 2 or 4 MHz, the signals from the computer cannot be directly copied onto tape.

One solution is to convert the computer's digital signals into sine waves that can be easily recorded. A digital-to-analog (D/A) circuit is used for this purpose. When the recorded program is subsequently played back into the computer, a separate analog-to-digital (A/D) circuit reverses the process. It converts the signal from a sine wave back into the digital form. The D/A and A/D circuits are combined on a printed circuit board with the usual parallel-to-serial converter for the I/O port.

THE CHECKSUM METHOD

Two separate tape-handling routines are given in this chapter; both may be used with paper tape or magnetic tape. They are both suitable for storing binary object programs, ASCII-encoded source programs, or just a set of numbers. The information is stored in a file consisting of a sequence of records. Each record contains a checksum that is used to detect errors.

Errors can be introduced at several places in the tape-recording and playback process. The proximity of AC power cords to audio signal lines can change the transmitted signal. Oxide layers may fall off the tape, or there may be a defective spot on the recording surface. If the recording and playback heads are dirty, they can incorrectly render the signal. The A/D conversion step, when the tape is played back, can also give rise to errors.

In addition to the data, each record stored on the tape contains the record length, the memory address of the record, and a checksum byte. The checksum byte is obtained by adding all of the data bytes in the record. When the tape is read back, the computer sums up the data and compares the result to the checksum value written on the tape at the end of the record. A discrepancy indicates an error.

Since an 8-bit checksum is used in both programs, there are 256 possible combinations. Any single load error in the record will be discovered by this method. In principle, it is possible that two errors in the same record will combine to produce the expected checksum value. In practice, however, this is not likely to be a problem. Double errors occur much less frequently than single errrors. Furthermore, in a well-tuned system, single errors should be infrequent. They are most likely to be caused by low-quality tape, a dirty head, or a mistuned A/D converter circuit. Buy the best tape you can and clean the heads often. A routine that can be used for aligning the A/D circuit is incorporated in the second tape routine given later in this chapter.

AN ASCII-HEX TAPE PROGRAM

The tape program given in Listing 9.1 is based on an ASCII-encoded hexadecimal format. It can be used to produce paper tapes or magnetic tapes of computer programs. It can even produce a tape of itself. The program is self-contained and assembled to run at 100 hex. No outside routines are required. An additional feature of this program is that it can punch readable labels on the leader of a paper tape. (Of course, this feature is not very useful for magnetic tape recordings.) The label routine is discussed in the next section of this chapter.

Listing 9.1. Hexadecimal tape routines.

```
                ; HEXMON: A MONITOR TO DUMP, LOAD, AND
                ;       VERIFY INTEL HEX CHECKSUM TAPES
                ;       WITH TAPE LABEL FOR HEADER
                ;
                        TITLE   'hexmon with tlabel'
                ;
0100            ORG     100H
                ;
                ;
                ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
                ;
0010 =          RLEN    EQU     16      ;RECORD LENGTH
                ;
0010 =          CSTAT   EQU     10H     ;CONSOLE STATUS
0011 =          CDATA   EQU     CSTAT+1 ;CONSOLE DATA
0001 =          CIMSK   EQU     1       ;IN MASK
0002 =          COMSK   EQU     2       ;OUT MASK
0006 =          PSTAT   EQU     6       ;PUNCH STATUS
0007 =          PDATA   EQU     PSTAT+1 ;PUNCH DATA
0001 =          PIMSK   EQU     1       ;PUNCH IN MASK
0080 =          POMSK   EQU     80H     ;PUNCH OUT MASK
                ;
000D =          CR      EQU     13      ;CARRAGE RETURN
000A =          LF      EQU     10      ;LINE FEED
007F =          DEL     EQU     127
0008 =          CTRH    EQU     8       ;^H CONSOLE BACKUP
0000 =          NNULS   EQU     0       ;CONSOLE NULLS
                ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
                ;
0100 C37D01     START:  JMP     CONTIN
                ;
                ; INPUT A BYTE FROM THE CONSOLE
                ;
0103 DB10       INPUTT: IN      CSTAT
0105 E601               ANI     CIMSK
0107 CA0301             JZ      INPUTT
010A DB11               IN      CDATA
010C E67F               ANI     7FH     ;STRIP PARITY
010E C9                 RET
                ;
                ; OUTPUT A CHARACTER TO CONSOLE
                ;
```

```
010F F5         OUTT:   PUSH    PSW
0110 DB10       OUTW:   IN      CSTAT
0112 E602               ANI     COMSK
0114 CA1001             JZ      OUTW
0117 F1                 POP     PSW
0118 D311               OUT     CDATA
011A C9                 RET
                ; OUTPUT H,L TO CONSOLE
                ; 16-BIT BINARY TO HEX
                ;
011B 4C         OUTHL:  MOV     C,H     ;FETCH H
011C CD2001             CALL    OUTHX   ;PRINT IT
011F 4D                 MOV     C,L     ;FETCH L, PRINT IT
                ;
                ; CONVERT A BINARY NUMBER TO TWO
                ; HEX CHARACTERS, AND PRINT THEM
                ;
0120 79         OUTHX:  MOV     A,C
0121 1F                 RAR             ;ROTATE
0122 1F                 RAR             ; UPPER
0123 1F                 RAR             ; CHARACTER
0124 1F                 RAR             ; TO LOWER
0125 CD2901             CALL    HEX1    ;OUTPUT UPPER
0128 79                 MOV     A,C     ;OUTPUT LOWER
                ;
                ; OUTPUT A HEX CHARACTER
                ; FROM LOWER FOUR BITS
                ;
0129 E60F       HEX1:   ANI     0FH     ;TAKE 4 BITS
012B C690               ADI     144
012D 27                 DAA             ;DAA TRICK
012E CE40               ACI     64
0130 27                 DAA             ;ONCE AGAIN
0131 C30F01             JMP     OUTT
                ;
                ; CONVERT ASCII CHARACTER FROM CONSOLE
                ; TO 4 BINARY BITS
                ;
0134 D630       NIB:    SUI     '0'     ;ASCII BIAS
0136 D8                 RC              ; < '0'
0137 FE17               CPI     'F'-'0'+1
0139 3F                 CMC             ;INVERT
013A D8                 RC              ;ERROR, > 'F'
013B FE0A               CPI     10
013D 3F                 CMC             ;INVERT
013E D0                 RNC             ;NUMBER IS 0-9
013F D607               SUI     'A'-'9'-1
0141 FE0A               CPI     10
0143 C9                 RET             ;CHARACTER IS A-F
                ;
                ; INPUT H,L FROM CONSOLE
                ;
0144 D5         READHL: PUSH    D
0145 C5                 PUSH    B
0146 210000             LXI     H,0     ;START WITH 0
0149 CD0402     RDHL2:  CALL    GETCH
```

```
014C DA6801            JC      RDHL5    ;END OF LINE
014F CD3401            CALL    NIB      ;CONVERT TO BINARY
0152 DA5E01            JC      RDHL4    ;NOT HEX
0155 29                DAD     H        ;16-BIT
0156 29                DAD     H        ; SHIFT LEFT
0157 29                DAD     H
0158 29                DAD     H
0159 B5                ORA     L        ;COMBINE NEW
015A 6F                MOV     L,A
015B C34901            JMP     RDHL2    ;NEXT
                ;
                ; CHECK FOR COMMA OR BLANK
                ; AT END OF ADDRESS
                ;
015E FEFC       RDHL4: CPI     ','-'0'  ;COMMA?
0160 CA6801            JZ      RDHL5    ;YES, OK
0163 FEF0              CPI     ' '-'0'  ;BLANK?
0165 C26B01            JNZ     ERROR    ;NO
0168 C1         RDHL5: POP     B
0169 D1                POP     D
016A C9                RET
                ;
016B 3E3F       ERROR: MVI     A,'?'    ;IMPROPER INPUT
016D CD0F01            CALL    OUTT
0170 C30001            JMP     START    ;TELL HOW AGAIN
                ;
                ; SEND CHRACTERS POINTED TO BY D,E
                ; UNTIL A BINARY ZERO IS FOUND
                ;
0173 1A         SENDM: LDAX    D        ;NEXT BYTE
0174 B7                ORA     A        ;SEE IF ZERO
0175 C8                RZ               ;DONE
0176 CD0F01            CALL    OUTT
0179 13                INX     D
017A C37301            JMP     SENDM
                ;
017D 312106     CONTIN: LXI    SP,STACK
0180 118603            LXI     D,SIGN   ;MESSAGE
0183 CD7301            CALL    SENDM    ;SEND IT
                ;
0186 312106     RSTRT: LXI     SP,STACK
0189 CDAC01            CALL    INPLN    ;GET A LINE
018C CD0402            CALL    GETCH    ;INPUT THE TASK
018F FE57              CPI     'W'      ;DUMP
0191 CA1A02            JZ      PDUMP
0194 FE52              CPI     'R'      ;READ, NO AUTOSTART
0196 CAE602            JZ      PLOAD
0199 FE45              CPI     'E'      ;LOAD AND EXECUTE
019B CAE602            JZ      PLOAD
019E FE56              CPI     'V'      ;VERIFY
01A0 CAE602            JZ      PLOAD
01A3 FE47              CPI     'G'      ;GO SOMEWHERE
01A5 C26B01            JNZ     ERROR
                ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
                ;
```

```
                    ; JUMP TO ANOTHER PROGRAM
                    ;
01A8 CD4401                 CALL    READHL  ;JUMP ADDRESS
01AB E9             JPCHL:  PCHL            ;OK, GOODBYE
                    ;
                    ; INPUT A LINE FROM THE CONSOLE
                    ; AND PUT IT INTO A BUFFER
                    ;
01AC CDEE01         INPLN:  CALL    CRLF
01AF 3E3E                   MVI     A,'>'   ;COMMAND PROMPT
01B1 CD0F01                 CALL    OUTT    ;SEND TO CONSOLE
01B4 212406         INPL2:  LXI     H,IBUFF ;BUFFER ADDRESS
01B7 222106                 SHLD    IBUFP   ;INITIALIZE POINTER
01BA 0E00                   MVI     C,0     ;INITIALIZE COUNT
01BC CD0301         INPLI:  CALL    INPUTT  ;CHAR FROM CONSOLE
01BF FE20                   CPI     ' '     ;CONTROL CHAR?
01C1 DAE001                 JC      INPLC   ;YES, GO PROCESS
01C4 FE7F                   CPI     DEL     ;DELETE CHAR?
01C6 CAF801                 JZ      INPLB   ;YES
01C9 FE5B                   CPI     'Z'+1   ;UPPER CASE?
01CB DAD001                 JC      INPL3   ;NO
01CE E65F                   ANI     5FH     ;MAKE UPPER
01D0 77             INPL3:  MOV     M,A     ;PUT IN BUFFER
01D1 3E20                   MVI     A,32    ;GET BUFFER SIZE
01D3 B9                     CMP     C       ;TEST IF FULL
01D4 CABC01                 JZ      INPLI   ;YES, LOOP
01D7 7E                     MOV     A,M     ;RECALL CHARACTER
01D8 23                     INX     H       ;INCR POINTER
01D9 0C                     INR     C       ;AND INCR COUNT
01DA CD0F01         INPLE:  CALL    OUTT    ;ECHO CHARACTER
01DD C3BC01                 JMP     INPLI   ;GET NEXT CHAR
                    ;
                    ; PROCESS CONROL CHARACTER
                    ;
01E0 FE08           INPLC:  CPI     CTRH    ;^H?
01E2 CAF801                 JZ      INPLB   ;YES
01E5 FE0D                   CPI     CR      ;TEST IF RETURN
01E7 C2BC01                 JNZ     INPLI   ;NO, IGNORE CHAR
                    ;
                    ; END OF INPUT LINE
                    ;
01EA 79                     MOV     A,C     ;LINE COUNT
01EB 322306                 STA     IBUFC   ;SAVE
                    ;
                    ; CARRIAGE RETURN, LINE FEED
01EE 3E0D           CRLF:   MVI     A,CR
01F0 CD0F01                 CALL    OUTT
01F3 3E0A                   MVI     A,LF
                    ;
                    ; IF NULLS REQUIRED AFTER CR, LF
                    ; USE REPEAT MACRO
                    ;
```

```
                        IF      NNULS > 0 ;ASSEMBLE
                        CALL    OUTT
                        XRA     A       ;GET A NULL
                        REPT    NNULS-1
                        CALL    OUTT
                        ENDM
                        ENDIF           ;NULLS
                ;
01F5 C30F01             JMP     OUTT
                ;
                ; DELETE ANY PRIOR CHARACTER
                ;
01F8 79         INPLB:  MOV     A,C     ;CHAR COUNT
01F9 B7                 ORA     A       ;ZERO?
01FA CABC01             JZ      INPLI   ;YES
01FD 2B                 DCX     H       ;BACK POINTER
01FE 0D                 DCR     C       ;AND COUNT
01FF 3E08               MVI     A,CTRH  ;BACK CURSOR
0201 C3DA01             JMP     INPLE   ;PRINT IT
                ;
                ; OBTAIN A CHARACTER FROM THE CONSOLE
                ; BUFFER. SET CARRY IF EMPTY.
                ;
0204 E5         GETCH:  PUSH    H       ;SAVE REGS
0205 2A2106             LHLD    IBUFP   ;GET POINTER
0208 3A2306     GETC2:  LDA     IBUFC   ;GET COUNT
020B D601               SUI     1       ;DECR WITH CARRY
020D DA1802             JC      GETC4   ;NO CHARACTERS
0210 322306             STA     IBUFC   ;REPLACE COUNT
0213 7E                 MOV     A,M     ;GET CHARACTER
0214 23         GETC3:  INX     H       ;INCR POINTER
0215 222106             SHLD    IBUFP   ;REPLACE POINTER
0218 E1         GETC4:  POP     H       ;RESTORE REGS
0219 C9                 RET             ;CARRY IF NO CHAR
                ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
                ;
                ; PUNCH A PAPER TAPE
                ;
021A CD4401     PDUMP:  CALL    READHL  ;START ADDRESS
021D DA6B01             JC      ERROR   ;TOO FEW PARAM
0220 EB                 XCHG
0221 CD4401             CALL    READHL  ;STOP ADDRESS
0224 EB                 XCHG
0225 13                 INX     D
0226 E5                 PUSH    H
0227 CD4401             CALL    READHL  ;AUTOSTART ADDR
022A E3                 XTHL            ;PUT ON STACK
022B CD8502             CALL    LEADR   ;PUNCH LEADER
022E CD8404             CALL    LABEL   ;PUNCH LABEL
                ;
                ; START NEW RECORD,   ZERO THE CHECKSUM
                ; PUNCH CR, LF, 2 NULLS AND COLON
                ;
0231 CDD002     NEWREC: CALL    PCRLF   ;CR, LF, NULLS
                ;
                ; FIND THE RECORD LENGTH
                ;
```

```
0234 7B              MOV    A,E      ;COMPARE LOW STOP
0235 95              SUB    L        ; TO LOW POINTER
0236 4F              MOV    C,A      ;DIFFERENCE IN C
0237 7A              MOV    A,D      ;COMPARE HIGH STOP
0238 9C              SBB    H        ; TO HIGH POINTER
0239 47              MOV    B,A      ;DIFFERENCE TO B
023A DA6B01          JC     ERROR    ;IMPROPER, H,L > D,E
023D 3E10            MVI    A,RLEN   ;FULL RECORD
023F C24C02          JNZ    NEW2
0242 B9              CMP    C        ;COMPARE TO E-L
0243 DA4C02          JC     NEW2     ;FULL RECORD LENGTH
0246 78              MOV    A,B      ;ARE BOTH E-L AND
0247 B1              ORA    C        ; D-E ZERO?
0248 CA6802          JZ     DONE     ;YES, REC LENGTH = 0
024B 79              MOV    A,C      ;SHORT RECORD
024C 4F      NEW2:   MOV    C,A      ;RECORD LENGTH TO C
024D 0600            MVI    B,0      ;ZERO THE CHECKSUM
024F CD9502          CALL   PNHEX    ;PUNCH RECORD LENGTH
0252 CD9002          CALL   PUNHL    ;PUNCH H/L
0255 AF              XRA    A
0256 CD9502          CALL   PNHEX    ;PUNCH RECORD TYPE 0
0259 7E      PMEM:   MOV    A,M
025A CD9502          CALL   PNHEX    ;PUNCH MEMORY BYTE
025D 23              INX    H        ;INCR. MEMORY POINTER
025E 0D              DCR    C        ;DECR RECORD LENGTH
025F C25902          JNZ    PMEM
0262 CD6703          CALL   CSUM     ;PUNCH CHECKSUM
0265 C33102          JMP    NEWREC   ;NEXT RECORD
             ;
             ; FINISHED, PUNCH LAST RECORD, RECORD
             ; LENGTH 00, THE START ADDRESS,
             ; AND  A RECORD TYPE OF 01
             ;
0268 AF      DONE:   XRA    A
0269 47              MOV    B,A      ;ZERO CHECKSUM
026A CD9502          CALL   PNHEX    ;ZERO RECORD LEN.
026D E1              POP    H
026E CD9002          CALL   PUNHL    ;AUTOSTART H/L
0271 7C              MOV    A,H      ;CHECK FOR
0272 B5              ORA    L        ; AUTOSTART
0273 3E00            MVI    A,0      ;0 WITH CARRY
0275 CA7902          JZ     DON2     ;NO AUTOSTART
0278 3C              INR    A
0279 CD9502  DON2:   CALL   PNHEX    ;RECORD TYPE 1
027C CD6703          CALL   CSUM     ;PUNCH CHECKSUM
027F CD8502          CALL   LEADR    ;PUNCH TRAILER
0282 C38601          JMP    RSTRT    ;NEXT JOB
             ;
             ; PUNCH BLANK HEADER AND TRAILER
             ;
0285 AF      LEADR:  XRA    A
0286 063C            MVI    B,60     ;TAPE NULLS
0288 CDB802  NLDR:   CALL   POUT
028B 05              DCR    B
028C C28802          JNZ    NLDR
028F C9              RET
             ;
```

```
                    ; PUNCH THE H,L REGISTER PAIR
                    ;
0290 7C             PUNHL:  MOV     A,H     ;FETCH H
0291 CD9502                 CALL    PNHEX   ;PUNCH IT
0294 7D                     MOV     A,L     ;GET L, PUNCH IT
                    ;
                    ; CONVERT A BINARY NUMBER TO TWO HEX
                    ; CHARACTERS, PUNCH THEM, ADD TO CHECKSUM
                    ;
0295 F5             PNHEX:  PUSH    PSW     ;SAVE ON STACK
0296 80                     ADD     B       ;ADD TO CHECKSUM.
0297 47                     MOV     B,A     ;SAVE IT IN B
0298 F1                     POP     PSW     ;RETRIEVE BYTE
0299 F5                     PUSH    PSW
029A 1F                     RAR
029B 1F                     RAR             ;ROTATE UPPER
029C 1F                     RAR
029D 1F                     RAR             ;TO LOWER
029E CDA202                 CALL    PHEX1   ;LEFT CHARACTER
02A1 F1                     POP     PSW     ;RIGHT CHARACTER
                    ;
                    ; PUNCH A HEX CHARACTER FROM
                    ; LOWER FOUR BITS
                    ;
02A2 E60F           PHEX1:  ANI     0FH     ;MASK UPPER 4 BITS
02A4 C690                   ADI     144
02A6 27                     DAA
02A7 CE40                   ACI     64
02A9 27                     DAA
02AA C3B802                 JMP     POUT
                    ;
                    ; INPUT A HEX CHARACTER FROM TAPE
                    ;
02AD CDC402         HEX:    CALL    PIN
02B0 D630                   SUI     '0'
02B2 FE0A                   CPI     10
02B4 D8                     RC              ;0-9
02B5 D607                   SUI     7       ;A-F
02B7 C9                     RET
                    ;
                    ; OUTPUT A BYTE TO THE PUNCH
                    ;
02B8 F5             POUT:   PUSH    PSW
02B9 DB06           POUTW:  IN      PSTAT
02BB E680                   ANI     POMSK
02BD C2B902                 JNZ     POUTW
02C0 F1                     POP     PSW
02C1 D307                   OUT     PDATA
02C3 C9                     RET
                    ;
                    ; INPUT A BYTE FROM PAPER TAPE
                    ;
02C4 DB06           PIN:    IN      PSTAT
02C6 E601                   ANI     PIMSK
02C8 C2C472                 JNZ     PIN
02CB DB07                   IN      PDATA
02CD E67F                   ANI     7FH     ;STRIP PARITY
02CF C9                     RET
                    ;
```

```
                        ; PUNCH CR, LF, NULLS AND COLON
                        ;
02D0 3E0D       PCRLF:  MVI     A,CR
02D2 CDB802             CALL    POUT
02D5 3E0A               MVI     A,LF
02D7 CDB802             CALL    POUT
02DA AF                 XRA     A
02DB CDB802             CALL    POUT    ;TWO NULLS
02DE CDB802             CALL    POUT
02E1 3E3A               MVI     A,':'   ;COLON
02E3 C3B802             JMP     POUT
                        ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
                        ; ENTRY FOR LOAD, EXECUTE AND VERIFY
                        ;
02E6 320006     PLOAD:  STA     TASK
02E9 3E11               MVI     A,17    ;TAPE READER ON
02EB CDB802             CALL    POUT
02EE CD4401             CALL    READHL  ;OFFSET
02F1 220106             SHLD    OFSET   ;SAVE IT
                        ; PROCESS THE RECORD HEADING ON INPUT
                        ;
02F4 CDC402     HEAD:   CALL    PIN     ;INPUT FROM TAPE
02F7 FE3A               CPI     ':'     ;COLON?
02F9 C2F402             JNZ     HEAD    ;NO, TRY AGAIN
02FC 0600               MVI     B,0     ;ZERO THE CHECKSUM
02FE CD3303             CALL    PHEX    ;RECORD LENGTH
0301 B7                 ORA     A       ;IS IT ZERO?
0302 CA4403             JZ      ENDFL   ;YES, DONE
0305 4F                 MOV     C,A     ;SAVE REC. LEN.
0306 CD2B03             CALL    TAPEHL  ;GET H/L
0309 EB                 XCHG            ;ADDR TO D,E
030A 2A0106             LHLD    OFSET   ;GET OFFSET
030D 19                 DAD     D       ;ADD
030E CD3303     LOOP:   CALL    PHEX    ;INPUT DATA BYTE
0311 5F                 MOV     E,A     ;SAVE BYTE
0312 3A0006             LDA     TASK    ;GET TASK
0315 FE56               CPI     'V'     ;SEE IF VERIFYING
0317 7B                 MOV     A,E     ;MOVE BACK
0318 CA1C03             JZ      SKIP    ;JUMP IF VERIFYING
031B 77                 MOV     M,A     ;DATA TO MEMORY
031C BE         SKIP:   CMP     M       ;CHECK MEMORY
031D C27603             JNZ     MERROR  ;BAD MEMORY
0320 23                 INX     H       ;INCREMENT POINTER
0321 0D                 DCR     C       ;DECR RECORD LEN
0322 C20E03             JNZ     LOOP    ;NOT YET ZERO
0325 CD6D03             CALL    CHECK   ;PROCESS CHECKSUM
0328 C3F402             JMP     HEAD    ;START NEXT RECORD
                        ;
                        ; INPUT H,L AND RECORD TYPE FROM TAPE
                        ;
032B CD3303     TAPEHL: CALL    PHEX    ;READ H
032E 67                 MOV     H,A
032F CD3303             CALL    PHEX    ;READ L
0332 6F                 MOV     L,A     ;READ RECORD TYPE
                        ;
                        ; CONVERT 2 CHAR FROM TAPE TO ONE BINARY
                        ; WORD, STORE IN A AND ADD TO CHECKSUM
```

```
                     ;
0333 CDAD02     PHEX:   CALL    HEX     ;UPPER CHARACTER
0336 07                 RLC
0337 17                 RAL             ;MOVE TO UPPER
0338 17                 RAL
0339 17                 RAL             ;HALF
033A 5F                 MOV     E,A     ;SAVE IT
033B CDAD02             CALL    HEX     ;LOWER CHARACTER
033E 83                 ADD     E       ;COMBINE BOTH
033F 5F                 MOV     E,A     ;SAVE IT
0340 80                 ADD     B       ;ADD TO CHECKSUM
0341 47                 MOV     B,A     ;SAVE IT
0342 7B                 MOV     A,E     ;RETRIEVE DATA
0343 C9                 RET
                ;
                ; ROUTINE TO CHECK FOR AUTOSTART
                ;
0344 CD2B03     ENDFL:  CALL    TAPEHL  ;AUTOSTART ADDRESS
                                        ;AND RECORD TYPE
0347 F5                 PUSH    PSW     ;SAVE RECORD TYPE
0348 CD3303             CALL    PHEX    ;INPUT CHECKSUM
034B CD6203             CALL    TOFF    ;TAPE READER OFF
034E F1                 POP     PSW     ;RETRIEVE REC TYPE
034F FE01               CPI     1       ;AUTOSTART?
0351 C28601             JNZ     RSTRT   ;NO
0354 3A0006             LDA     TASK    ;CHECK TASK
0357 FE45               CPI     'E'     ;EXECUTE?
0359 CAAB01             JZ      JPCHL   ;YES, GO THERE
035C CD1B01             CALL    OUTHL   ;NO, PRINT HL
035F C38601             JMP     RSTRT   ;NEXT TASK
                ;
                ; TURN OFF TAPE READER
                ;
0362 3E13       TOFF:   MVI     A,19
0364 C3B802             JMP     POUT
                ;
                ; CALCULATE AND PUNCH THE CHECKSUM
                ;
0367 78         CSUM:   MOV     A,B     ;CHECKSUM TO A
0368 2F                 CMA             ;ONE'S COMPLEMENT
0369 3C                 INR     A       ;TWO'S COMPLEMENT
036A C39502             JMP     PNHEX   ;PUNCH CHECKSUM
                ;
                ; SEE IF CHECKSUM IS CORRECT (ZERO)
                ;
036D CD3303     CHECK:  CALL    PHEX    ;INPUT CHECKSUM
0370 AF                 XRA     A
0371 80                 ADD     B       ;IS CHECKSUM ZERO?
0372 C8                 RZ              ;YES, RETURN
                ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
                ;
                ; ERROR MESSAGES
                ;
0373 3E43               MVI     A,'C'   ;CHECKSUM ERROR
0375 01                 DB      1       ;DB TRICK TO SKIP
0376 3E4D       MERROR: MVI     A,'M'   ;M FOR BAD MEMORY
0378 F5                 PUSH    PSW
0379 CD6203             CALL    TOFF    ;TAPE READER OFF
```

```
037C F1                 POP     PSW
037D CD0F01             CALL    OUTT    ;PRINT ERROR TYPE
0380 CD1B01             CALL    OUTHL   ;PRINT H/L
0383 C38601             JMP     RSTRT
                ;
0386 0D0A       SIGN:   DB      CR,LF
0388 4865782070         DB      'Hex paper-tape program'
039E 0D0A0A             DB      CR,LF,LF
03A1 45202D206C         DB      'E - load and execute'
03B5 0D0A               DB      CR,LF
03B7 47202D2067         DB      'G - go to addresses given'
03D0 0D0A               DB      CR,LF
03D2 52202D2072         DB      'R - read tape into memory'
03EB 0D0A               DB      CR,LF
03ED 2020202028         DB      '    (with optional offset)'
0407 0D0A               DB      CR,LF
0409 56202D2076         DB      'V - verify tape against'
0420 206D656D6F         DB      ' memory',CR,LF
0429 57202D2077         DB      'W - write paper tape'
043D 2028616E64         DB      ' (and label)',CR,LF,'    '
044F 2877697468         DB      '(with optional autostart)'
0468 0D0A00             DB      CR,LF,0

046B 0D0A456E74LMESG:   DB      CR,LF,'Enter leader message'
0481 0D0A00             DB      CR,LF,0
                ;
                ; PUNCH READABLE LABELS ON PAPER TAPE
                ;
0484 E5         LABEL:  PUSH    H
0485 D5                 PUSH    D
0486 116B04             LXI     D,LMESG ;LABEL MESS.
0489 CD7301             CALL    SENDM   ;SEND IT
048C CDAC01             CALL    INPLN   ;GET A LINE
048F CD0402     LABL1:  CALL    GETCH   ;GET CHARACTER
0492 DABF04             JC      LABL2   ;DONE ON CARRY
0495 DE20               SBI     20H     ;ASCII BIAS
0497 DA8F04             JC      LABL1   ;< SPACE
049A FE3F               CPI     63
049C D28F04             JNC     LABL1   ;TOO BIG
049F 6F                 MOV     L,A
04A0 5F                 MOV     E,A
04A1 2600               MVI     H,0
04A3 1600               MVI     D,0
04A5 29                 DAD     H       ;DOUBLE IT
04A6 29                 DAD     H       ;TIMES 4
04A7 19                 DAD     D       ;TIMES FIVE
04A8 EB                 XCHG
04A9 21C504             LXI     H,TABL
04AC 19                 DAD     D
04AD 0E05               MVI     C,5
04AF 7E         NEXTC:  MOV     A,M
04B0 CDB802             CALL    POUT
04B3 23                 INX     H
04B4 0D                 DCR     C
04B5 C2AF04             JNZ     NEXTC
04B8 AF                 XRA     A
04B9 CDB802             CALL    POUT
04BC C38F04             JMP     LABL1   ;NEXT CHARACTER
```

```
04BF CD8502      LABL2:   CALL    LEADR
04C2 D1                   POP     D
04C3 E1                   POP     H
04C4 C9                   RET
                 ;
04C5 0000000000TABL:      DB      0,   0,   0,   0,   0     ; SPACE
04CA 0000CFCF00           DB      0,   0,   207,207,0       ; EXCL
04CF 0007000700           DB      0,   7,   0,   7,   0     ; "
04D4 28FE28FE28           DB      40,  254,40,  254,40      ; #
04D9 4689FF8972           DB      70,  137,255,137,114      ; $
04DE 462610C8C4           DB      70,  38,  16,  200,196    ; %
04E3 6C92AC40A0           DB      108,146,172,64,  160      ; &
04E8 0004030300           DB      0,   4   ,3,   3,   0     ; '
04ED 003C428100           DB      0,   60,  66,129,  0      ; (
04F2 0081423C00           DB      0,   129,66,  60,  0      ; )
04F7 8850F85088           DB      136,80,  248,80,  136     ; *
04FC 08087E0808           DB      8,   8,   126,8,   8      ; +
0501 0080703000           DB      0,   128,112,48,  0       ; ,
0506 0808080808           DB      8,   8,   8,   8,   8     ; -
050B 00C0C00000           DB      0,   192,192,0,   0       ; .
0510 4020100804           DB      64,  32,  16,  8,   4     ; /
0515 7EA189857E           DB      126,161,137,133,126       ; 0
051A 8482FF8080           DB      132,130,255,128,128       ; 1
051F C2A1918986           DB      194,161,145,137,134       ; 2
0524 4289898976           DB      66,  137,137,137,118      ; 3
0529 0C0A89FF88           DB      12,   10,137,255,136      ; 4
052E 6789898971           DB      103,137,137,137,113       ; 5
0533 7E89898972           DB      126,137,137,137,114       ; 6
0538 0101F90503           DB      1,   1,   249,5,   3      ; 7
053D 7689898976           DB      118,137,137,137,118       ; 8
0542 468989897E           DB      70,  137,137,137,126      ; 9
0547 00D8D80000           DB      0,   216,216,0,   0       ; :
054C 0080763600           DB      0,   128,118,54,  0       ; ;
0551 1028448200           DB      16,  40,  68,  130,0      ; <
0556 2828282828           DB      40,  40,  40,  40,  40    ; =
055B 8244281000           DB      130,68,  40,  16,  0      ; >
0560 0601B90906           DB      6,   1,   185,9,   6      ; ?
0565 7E819D910E           DB      126,129,157,145,14        ;
056A FE090909FE           DB      254,9,   9,   9,   254    ; A
056F 81FF898976           DB      129,255,137,137,118       ; B
0574 7E81818142           DB      126,129,129,129, 66       ; C
0579 81FF81817E           DB      129,255,129,129,126       ; D
057E FF89898989           DB      255,137,137,137,137       ; E
0583 FF09090901           DB      255,9,   9   ,9   ,1      ; F
0588 7E81919172           DB      126,129,145,145,114       ; G
058D FF080808FF           DB      255,8,   8,   8,   255    ; H
0592 0081FF8100           DB      0,   129,255,129,0        ; I
0597 6080817F01           DB      96,  128,129,127,1        ; J
059C FF081422C1           DB      255,8,   20,  34,  193    ; K
05A1 FF80808080           DB      255,128,128,128,128       ; L
05A6 FF020C02FF           DB      255,2,  12,   2,   255    ; M
05AB FF023C40FF           DB      255,2,  60,  64,   255    ; N
05B0 FF818181FF           DB      255,129,129,129,255       ; O
05B5 0509090906           DB      5,   9,   9,   9,   6     ; P
05BA 7E81A141BE           DB      126,129,161,65,  190      ; Q
05BF FF19294986           DB      255,25,  41,  73,  134    ; R
05C4 4689898972           DB      70,  137,137,137,114      ; S
05C9 0101FF0101           DB      1,   1,   255,1,   1      ; T
```

```
05CE 7F8080807F         DB      127,128,128,128,127 ; U
05D3 0F30C0300F         DB      15, 48, 192,48, 15  ; V
05D8 7F8070807F         DB      127,128,112,128,127 ; W
05DD C3241824C3         DB      195,36, 24, 36, 195 ; X
05E2 0304F80403         DB      3,  4,  248,4  ,3   ; Y
05E7 C1A1918987         DB      193,161,145,137,135 ; Z
05EC 00FF818181         DB      0,  255,129,129,129 ; [
05F1 0408102040         DB      4,  8,  16, 32, 64  ; \
05F6 818181FF00         DB      129,129,129,255,0   ; ]
05FB 0C0201020C         DB      12, 2,  1,  2,  12  ; ^

0600            TASK:   DS      1          ;SAVE IT
0601 0000       OFSET:  DW      0          ;LOAD OFFSET
0603                    DS      30         ;STACK SPACE
                STACK:
0621            IBUFP:  DS      2          ;BUFFER POINTER
0623            IBUFC:  DS      1          ;BUFFER COUNT
0624            IBUFF:  DS      20         ;INPUT BUFFER
                ;
0638                    END
```

```
Symbols:

0011 CDATA     036D CHECK     0001 CIMSK     0002 COMSK
017D CONTIN    000D CR        01EE CRLF      0010 CSTAT
0367 CSUM      0008 CTRH      007F DEL       0279 DON2
0268 DONE      0344 ENDFL     016B ERROR     0208 GETC2
0214 GETC3     0218 GETC4     0204 GETCH     02F4 HEAD
0129 HEX1      02AD HEX       0623 IBUFC     0624 IBUFF
0621 IBUFP     01B4 INPL2     01D0 INPL3     01F8 INPLB
01E0 INPLC     01DA INPLE     01BC INPLI     01AC INPLN
0103 INPUTT    01AB JPCHL     0484 LABEL     048F LABL1
04BF LABL2     0285 LEADR     000A LF        046B LMESG
030E LOOP      0376 MERROR    024C NEW2      0231 NEWREC
04AF NEXTC     0134 NIB       0288 NLDR      0000 NNULS
0601 OFSET     011B OUTHL     0120 OUTHX     010F OUTT
0110 OUTW      02D0 PCRLF     0007 PDATA     021A PDUMP
02A2 PHEX1     0333 PHEX      0001 PIMSK     02C4 PIN
02E6 PLOAD     0259 PMEM      0295 PNHEX     0080 POMSK
02B8 POUT      02B9 POUTW     0006 PSTAT     0290 PUNHL
0149 RDHL2     015E RDHL4     0168 RDHL5     0144 READHL
0010 RLEN      0186 RSTRT     0173 SENDM     0386 SIGN
031C SKIP      0621 STACK     0100 START     04C5 TABL
032B TAPEHL    0600 TASK      0362 TOFF
```

This program generates tapes with a modified Intel format. A typical record is shown in Figure 9.1.

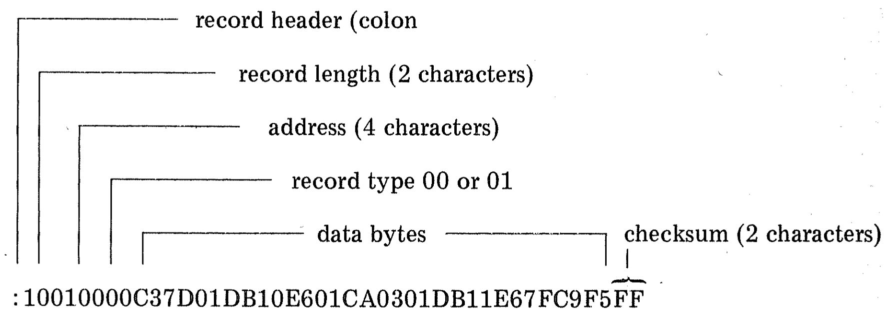


Figure 9.1. The hex tape format.

In this example, the record length is 10 hex, the address is 100 hex, the record type is 00, the first data byte is C3, the last data byte is F5 hex, and the checksum is FF hex. This is the format used by the CP/M assemblers ASM and MAC to generate HEX files.

Each record starts with a colon and is followed by the hex-encoded data. Since the data are encoded in ASCII, two bytes on the tape are needed to represent each original byte. Two hex characters, representing the record length, follow the header character. This hex value gives the actual number of data bytes contained in the record. It does not include the other characters in the record which designate the record address, the record type, and the checksum. A zero value for the record length is used for the last record to indicate the end of the file.

The record address, the address of the first byte in the record, comes next. This address is encoded into four hex characters, high-order byte first. The next item is the record type; it consists of two characters and can be 00 or 01. The value is usually 00, however; 01 is used for the end-of-file record for autostart tapes.

The data from memory are encoded next. Two bytes on the tape represent each data byte. The checksum byte is the last item in the record. It is formed by taking the two's complement of the sum of all the previous bytes in the record (in binary form). A carriage return, line feed, and optional nulls follow the checksum.

When the hex tape is loaded into the computer, the bytes within each record including the checksum are added together. The checksum byte is formed from the two's complement of the original sum. Since the sum of a number and its two's complement is zero, the total at this point should be zero. A nonzero result indicates an error.

The last record in the file has a record length of zero. This end-of-file record can indicate an autostart address in place of the usual record address. The computer can branch to this autostart address after the program is loaded. The record type in this case is 01—the end-of-file record.

Since the tape is written with an ASCII-encoded format, it is easy to read. Listing 9.2 shows the first part of the monitor made with the monitor itself.

Listing 9.2 A hex dump of the beginning of the monitor.

```
:10010000C37D01DB10E601CA0301DB11E67FC9F5FF
:10011000DB10E602CA1001F1D311C94CCD20014D0C
:10012000791F1F1F1FCD290179E60FC69027CE40EA
:1001300027C30F01D630D8FE173FD8FE0A3FD0D6CE
```

The disadvantage of this format is that the tape takes twice as long to make and twice as long to read as a corresponding binary tape. A BASIC interpreter that loads in 20 minutes from a binary format would take 40 minutes to load from hex format.

You may want to reassemble the hex tape routine for some other memory location, or you may want to incorporate it into your system monitor. But wait until you've learned about the binary tape monitor shown later in this chapter before you do.

Type in the hex program, assemble it, and start it up by branching to the beginning. A list of commands will be printed as a guide to operation.

Hex paper-tape program:

E	load and execute
G	go to address given
R	read tape into memory (with optional offset)
V	verify tape against memory
W	write and label paper tape (with optional autostart)

Console input is buffered just as it is in our system monitor. In fact, you will notice that many of the monitor I/O routines have been duplicated in this tape program. To create a tape, type the letter W (for write), the start address, and an optional autostart address. Finish the line with a carriage return. Typing errors can be corrected as usual with a DEL or backspace key. When the statement "Enter leader message" appears, either type the characters that you want on the tape leader, or just type a carriage return to skip the title.

After the tape has been made, it should be verified. Type the letter V and a carriage return. If you have a Teletype with an automatic tape reader, the computer can turn it on at the beginning and turn it off at the end. This is accomplished by sending a control-Q at the beginning and a control-S at the end. Each entry on the tape will be compared to the corresponding value in memory. If a checksum or verify error is detected, the process will be terminated. The appropriate error message and the address of the error will appear. In the case of a checksum error, the address is not meaningful.

A tape can be loaded into memory by typing the letter R (for read) and the optional offset value. This offset is added to the record address, after the record address has been added into the checksum. An offset of 1000 hex will load a program 4K bytes higher than the regular address. An offset of F000 hex will load the program 4K bytes lower. If you want, the computer will branch to the autostart address after the tape has been loaded; simply type an E (for execute) rather than an R.

If a leader message has been punched on the tape, it is not necessary to position the tape past this message. The loader routine is looking for a colon to indicate the start of a record. But the colon symbol will never appear in the label itself. There is a GO command so that you can easily leave the tape program. Type the letter G (GO) and the address.

A TAPE-LABELING ROUTINE

The assembly language instructions that punch readable labels on paper tape begin at LABEL and continue on down to TABL. The data used by this portion starts at TABL and goes down to TASK. The characters that are available include the complete set of ASCII characters from the blank (20 hex) through the underline (5F hex). The uppercase letters are included but the lowercase letters are not. One line of data in the source program is used for each character punched on the tape. The characters can be identified by the comment at the end of the line. The lines of data are arranged in sequence corresponding to the ASCII character set. The characters punched on the tape are generated in a five-by-eight matrix, with a row of blanks between the characters.

When the label subroutine is called, it first prints a message requesting input. In response, the user enters one line of characters and concludes the line with a carriage return. The message is then punched on the tape. Each ASCII character is obtained from the console input buffer as needed. The ASCII character is converted to binary by subtraction of an ASCII blank. The result is then used as a pointer to find the corresponding entry in the table. This is done by multiplying the binary value by 5 and adding it to the table address. The next five bytes of the table are then sent to the punch. The complete set of characters is shown in Figure 9.2.

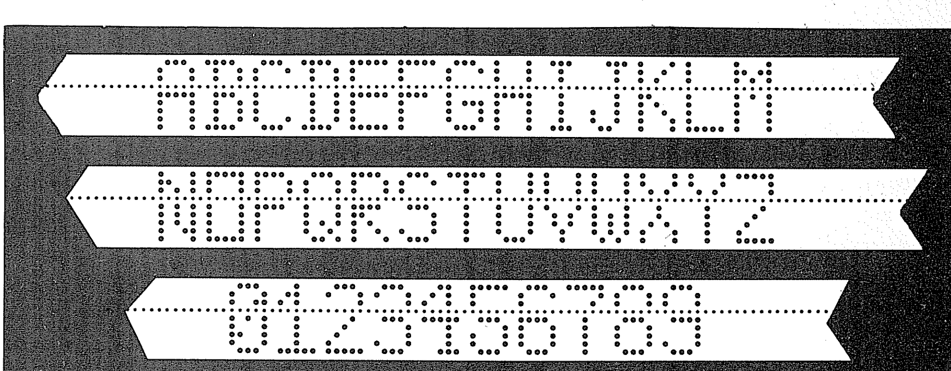


Figure 9.2. The set of letters and numbers produced by HEXMON.

The label subroutine with its necessary I/O routines can be removed from the tape program and used separately. It can be placed into high memory and called by another program such as a BASIC interpreter.

A BINARY TAPE MONITOR

The hexadecimal program given in the previous section is useful for paper tape. But a binary tape routine is much more suitable for magnetic tape. With a magnetic tape format, the tape cannot be visually inspected. Therefore, there is no advantage in coding the data in hexadecimal format.

Files are organized into records just as they are with the hex program. Each record starts with a record header, then continues with the record length, the address, the data, and the checksum. But the binary format is a bit different. As shown in Figure 9.3, there are three types of records: a header record, a data record, and an end-of-file record. The first record in the file is the file-header record. It begins with a 55-hex character. An optional filename appears next. The record header ends with a carriage-return character.

Header Record	55H	Filename	Carriage Return		
Data Record	3CH	Rec. Len.	Addr.	Data	Checksum
EOF Record	74H	Autostart Addr.	Checksum		

Figure 9.3. Three different record types are used with the binary tape format: A header record, one or more data records, and an end-of-file record.

The data record follows the header record. It starts with a 3C-hex header byte, a record-length byte, and two bytes giving the record address. The record address is written with the low-order byte first. The data bytes appear next, and then the checksum concludes the record.

With this format, the checksum byte consists of the sum of the data record bytes rather than the two's complement of the sum as used with the hex format. This checksum includes the record address and the data bytes, but it does not include the record length. The record length is not needed, since an incorrectly read record length will point to an incorrect byte that is to be used for the checksum.

The end-of-file record is four bytes long. It begins with a 74-hex character. It is followed by the autostart address, written as low byte, high byte. The fourth byte in this record is the checksum of the two-byte autostart address.

The routine given in Listing 9.3 is not a complete program. It is meant to be added to the end of the system monitor developed in Chapter 6. The

I/O and conversion routines of the monitor are used whenever possible. If you want to use it as a stand-alone program, you will have to include the necessary I/O routines. The addition of the tape program to the monitor will increase its size to one-and-a-half K bytes.

Add the new instructions to the end of the monitor, then change the monitor command-branch table for the letter T (tape)

```
DW        TAPE     ;T
```

so that a command line starting with the letter T will cause a branch to the new tape routines.

Listing 9.3. Binary tape routines.

```
                      ; LOCATIONS IN THE STACK AREA
                      ;
57C6 =                FBUF     EQU     IBUFF+32 ;FILENAME BUFF
57E6 =                OFSET    EQU     FBUF+20H ;LOAD OFFSET
57E8 =                TASK     EQU     OFSET+2
57E9 =                LFLAG    EQU     TASK+1   ;LOAD-ERROR FLAG
                      ;
0055 =                SBYTE    EQU     55H      ;SYNC BYTE
003C =                RHEAD    EQU     3CH      ;RECORD HEADER
0074 =                EOF      EQU     74H      ;END OF FILE
                      ;
0006 =                PSTAT    EQU     6        ;PUNCH STATUS
0007 =                PDATA    EQU     PSTAT+1  ;PUNCH DATA
0001 =                PIMSK    EQU     1        ;INPUT MASK
0080 =                POMSK    EQU     80H      ;OUTPUT MASK
                      ;
                      ;
5BFA 210000           TAPE:    LXI     H,0
5BFD 22E657                    SHLD    OFSET    ;ZERO OFFSET
5C00 CD2559                    CALL    GETCH    ;COMMAND
                      ;
                      ; TAPE-COMMAND PROCESSOR
                      ;
5C03 FE45                      CPI     'E'
5C05 CAD25C                    JZ      TLOAD    ;LOAD AND EXEC
5C08 FE4C                      CPI     'L'
5C0A CAD25C                    JZ      TLOAD    ;LOAD
5C0D FE4D                      CPI     'M'
5C0F CA805D                    JZ      TMAKE    ;MAKE TEST TAPE
5C12 FE4F                      CPI     'O'
5C14 CACA5C                    JZ      OFFST    ;OFFSET LOAD
5C17 FE54                      CPI     'T'
5C19 CA9A5D                    JZ      TTEST    ;TAPE TEST
5C1C FE56                      CPI     'V'
5C1E CAD25C                    JZ      TLOAD    ;VERIFY TAPE/MEM
5C21 FE44                      CPI     'D'      ;DUMP TO TAPE
5C23 C2D459                    JNZ     ERROR
                      ;
```

```
                      ; DUMP MEMORY TO TAPE IN 8-BIT BINARY FORMAT
                      ;
5C26 CD8659                   CALL    RDHLDE   ;ADDRESS LIMITS
5C29 E5                       PUSH    H        ;START ADDR
5C2A CD9D59                   CALL    READHL   ;AUTOSTART
5C2D E3                       XTHL             ;SWITCH WITH H,L
5C2E CD885C                   CALL    LEADR    ;TAPE LEADER
5C31 3E55                     MVI     A,SBYTE  ;SYNC BYTE
5C33 CD935C                   CALL    TOUT
5C36 CD2559          TDMP:    CALL    GETCH    ;FILE-NAME CHAR
5C39 DA425C                   JC      PD4      ;FILENAME END
5C3C CD935C                   CALL    TOUT     ;SEND TO TAPE
5C3F C3365C                   JMP     TDMP     ;NEXT CHAR
5C42 3E0D            PD4:     MVI     A,CR
5C44 CD935C                   CALL    TOUT     ;PUT CR ON TAPE
5C47 3E3C            PD0:     MVI     A,RHEAD  ;RECORD HEADER
5C49 CD935C                   CALL    TOUT     ;PUT ON TAPE
5C4C 7B                       MOV     A,E
5C4D 95                       SUB     L
5C4E 3C                       INR     A        ;RECORD LENGTH
5C4F 4F                       MOV     C,A
5C50 CD935C                   CALL    TOUT     ;PUT ON TAPE
5C53 7D                       MOV     A,L
5C54 CD935C                   CALL    TOUT     ;L TO TAPE
5C57 45                       MOV     B,L      ;START CHECKSUM
5C58 7C                       MOV     A,H
5C59 CD935C                   CALL    TOUT     ;H TO TAPE
5C5C 7E              PD1:     MOV     A,M      ;GET DATA
5C5D CD935C                   CALL    TOUT     ;SEND TO TAPE
5C60 0D                       DCR     C        ;RECORD COUNT
5C61 CA685C                   JZ      PD2      ;DONE
5C64 23                       INX     H        ;BUMP POINTER
5C65 C35C5C                   JMP     PD1      ;NEXT BYTE
                      ;
                      ; END OF RECORD
                      ;
5C68 78              PD2:     MOV     A,B      ;GET CHECKSUM
5C69 CD935C                   CALL    TOUT     ;SEND IT
5C6C 7C                       MOV     A,H
5C6D BA                       CMP     D        ;END OF FILE?
5C6E CA755C                   JZ      PD3      ;YES
5C71 23                       INX     H
5C72 C3475C                   JMP     PD0      ;NEXT RECORD
                      ;
                      ; END OF FILE
                      ;
5C75 3E74            PD3:     MVI     A,EOF
5C77 CD935C                   CALL    TOUT     ;SEND IT
5C7A E1                       POP     H        ;AUTOSTART ADDR
5C7B 7D                       MOV     A,L      ;LOW
5C7C CD935C                   CALL    TOUT     ;SEND IT
5C7F 45                       MOV     B,L      ;START CHECKSUM
5C80 7C                       MOV     A,H      ;HIGH
5C81 CD935C                   CALL    TOUT     ;SEND IT
5C84 78                       MOV     A,B      ;CHECKSUM
5C85 CD935C                   CALL    TOUT     ;SEND IT
```

```
                    ;
                    ; MAKE A LEADER/TRAILER ON TAPE
                    ;
5C88 AF             LEADR:  XRA     A       ;GET A NULL
5C89 0648                   MVI     B,72    ;# OF NULLS
5C8B CD935C         NLDR:   CALL    TOUT    ;SEND NULL
5C8E 05                     DCR     B
5C8F C28B5C                 JNZ     NLDR
5C92 C9                     RET
                    ;
                    ; OUTPUT BYTE TO TAPE, ADD TO CHECKSUM
                    ;
5C93 F5             TOUT:   PUSH    PSW
5C94 80                     ADD     B       ;TO CHECKSUM
5C95 47                     MOV     B,A     ;PUT BACK
5C96 F1                     POP     PSW
5C97 F5             POUT:   PUSH    PSW
5C98 DB06           POUTW:  IN      PSTAT   ;STATUS
5C9A E680                   ANI     POMSK
                    ;
                    ; NEXT LINE MAY NEED TO BE JZ POUTW
5C9C C2985C                 JNZ     POUTW   ;NOT READY
5C9F F1                     POP     PSW
5CA0 D307                   OUT     PDATA   ;SEND BYTE
5CA2 C9                     RET
                    ;
                    ; INPUT BYTE FROM TAPE
                    ; ADD TO CHECKSUM
                    ;
5CA3 CDB25C         TIN4:   CALL    PIN     ;GET BYTE
5CA6 F5                     PUSH    PSW
5CA7 80                     ADD     B       ;ADD TO CHECKSUM
5CA8 47                     MOV     B,A     ;PUT BACK
5CA9 F1                     POP     PSW
                    ;
                    ; SEND TO CONSOLE IF NOT A CONROL CHARACTER
                    ;
5CAA E67F                   ANI     7FH
5CAC FE20                   CPI     ' '
5CAE D8                     RC              ;CONTROL
5CAF D311                   OUT     CDATA
5CB1 C9                     RET
                    ;
5CB2 DB06           PIN:    IN      PSTAT   ;STATUS
5CB4 E601                   ANI     PIMSK
                    ;
                    ; NEXT LINE MAY NEED TO BE JZ  PIN
5CB6 C2B25C                 JNZ     PIN     ;NOT READY
5CB9 DB07                   IN      PDATA   ;GET BYTE
5CBB C9                     RET             ;KEEP 8 BITS
                    ;
                    ; LOOK FOR HEADER ON PAPER TAPE
                    ;
```

```
5CBC 3E11       PIN4:   MVI     A,CTRQ  ;^Q
5CBE CD975C             CALL    POUT    ;TURN ON READER
5CC1 CDB25C     PIN5:   CALL    PIN     ;GET BYTE
5CC4 FE55               CPI     SBYTE   ;SYNC BYTE?
5CC6 C2C15C             JNZ     PIN5    ;NO, TRY AGAIN
5CC9 C9                 RET
                ;
                ; LOAD A TAPE WITH 16-BIT OFFEST
                ;
5CCA 57         OFFST:  MOV     D,A     ;SAVE TASK
5CCB CD9D59             CALL    READHL  ;GET OFFSET
5CCE 22E657             SHLD    OFSET   ;SAVE IT
5CD1 7A                 MOV     A,D     ;RESTORE TASK
                ;
                ; LOAD/ VERIFY BINARY TAPE
                ;
5CD2 32E857     TLOAD:  STA     TASK    ;SAVE COMMAND
5CD5 21C657             LXI     H,FBUF  ;FILENAME BUFFER
5CD8 AF                 XRA     A       ;GET A ZERO
5CD9 32E957             STA     LFLAG   ;RESET FLAG
5CDC CD2559     TLD1:   CALL    GETCH   ;READ FILENAME
5CDF DAE75C             JC      TLD5    ;END OF FILENAME
5CE2 77                 MOV     M,A     ;PUT IN FBUF
5CE3 23                 INX     H
5CE4 C3DC5C             JMP     TLD1    ;NEXT CHARACTER
5CE7 3E0D       TLD5:   MVI     A,CR
5CE9 32A657             STA     IBUFF
5CEC CDBC5C             CALL    PIN4    ;FIND HEADER
5CEF 21C657             LXI     H,FBUF  ;FILENAME BUFFER
5CF2 7E                 MOV     A,M     ;1ST CHARACTER
5CF3 FE0D               CPI     CR
5CF5 CA1C5D             JZ      TLO     ;NO FILENAME
5CF8 11A657             LXI     D,IBUFF ;INPUT BUFFER
                ;
                ; INPUT FILENAME FROM TAPE, PUT IN IBUF
                ;
5CFB CDA35C     TLD2:   CALL    TIN4    ;BYTE FROM TAPE
5CFE 12                 STAX    D       ;PUT IN IBUFF
5CFF 13                 INX     D
5D00 FE0D               CPI     CR
5D02 CA125D             JZ      TLD4    ;NO FILENEME
5D05 BE                 CMP     M       ;COMP FILENAMES
5D06 23                 INX     H
5D07 CAFB5C             JZ      TLD2    ;NEXT CHARACTER
                ;
                ; FILENAME ENTERED FROM CONSOLE
                ; DOESN'T MATCH NAME ON TAPE
                ;
5D0A 3E46               MVI     A,'F'   ;SET ERROR FLAG
5D0C 32E957             STA     LFLAG
5D0F C3FB5C             JMP     TLD2
5D12 AF         TLD4:   XRA     A
5D13 1B                 DCX     D
5D14 12                 STAX    D
5D15 3AE957             LDA     LFLAG   ;CHECK FLAG
5D18 B7                 ORA     A       ;ZERO?
5D19 C2C05D             JNZ     FNERR   ;FILENAME ERROR
```

```
5D1C CDA35C     TLO:    CALL    TIN4     ;BYTE FROM TAPE
5D1F FE74               CPI     EOF
5D21 CA605D             JZ      EXEC     ;DONE
5D24 FE3C               CPI     RHEAD    ;RECORD HEADER
5D26 C21C5D             JNZ     TLO      ;TRY AGAIN
5D29 CDA35C             CALL    TIN4     ;RECORD LENGTH
5D2C 4F                 MOV     C,A      ;PUT IN C
5D2D CDA35C             CALL    TIN4     ;ADDR, LOW BYTE
5D30 5F                 MOV     E,A      ;INTO E
5D31 47                 MOV     B,A      ;PUT IN CHECKSUM
5D32 CDA35C             CALL    TIN4     ;ADDR, HIGH BYTE
5D35 57                 MOV     D,A      ;INTO D
5D36 2AE657             LHLD    OFSET    ;GET LOAD OFFSET
5D39 19                 DAD     D        ;ADD TO ADDRESS
5D3A 3AE857             LDA     TASK     ;GET TASK
5D3D 57                 MOV     D,A      ;PUT IN D
5D3E CDA35C     TL1:    CALL    TIN4     ;DATA BYTE
5D41 5F                 MOV     E,A      ;SAVE IN E
5D42 7A                 MOV     A,D      ;CHECK THE TASK
5D43 FE56               CPI     'V'      ;VERIFYING?
5D45 7B                 MOV     A,E      ;GET BYTE AGAIN
5D46 CA4A5D             JZ      SKIP     ;VERIFYING, SKIP
5D49 77                 MOV     M,A      ;INTO MEMORY
5D4A BE         SKIP:   CMP     M        ;IS IT THERE?
5D4B C2AE5D             JNZ     PERROR   ;NO
5D4E 23                 INX     H        ;INCR ADDRESS
5D4F 0D                 DCR     C        ;RECORD COUNT
5D50 C23E5D             JNZ     TL1      ;NEXT BYTE
                ;
                ; END OF RECORD, GET CHECKSUM
                ;
5D53 48                 MOV     C,B      ;CHECKSUM TO C
5D54 CDA35C             CALL    TIN4     ;TAPE CHECKSUM
5D57 B9                 CMP     C        ;THE SAME?
5D58 CA1C5D             JZ      TLO      ;YES
5D5B 3E43       CSERR:  MVI     A,'C'    ;ERROR
5D5D C3455A             JMP     ERR2
                ;
                ; END OF TAPE, GET AUTOSTART ADDRESS
                ; SEE IF EXECUTING
                ;
5D60 CDA35C     EXEC:   CALL    TIN4     ;START, LOW
5D63 6F                 MOV     L,A      ;PUT IN L
5D64 47                 MOV     B,A      ;START CHECKSUM
5D65 CDA35C             CALL    TIN4     ;START, HIGH
5D68 67                 MOV     H,A      ;PUT IN H
5D69 48                 MOV     C,B      ;GET SUM
5D6A CDA35C             CALL    TIN4     ;READ CHECKSUM
5D6D B9                 CMP     C        ;SAME
5D6E C25B5D             JNZ     CSERR    ;NO
5D71 CDB45D             CALL    POFF     ;READER OFF
5D74 7A                 MOV     A,D      ;CHECK TASK
5D75 FE45               CPI     'E'      ;EXECUTING?
5D77 CA7F5D             JZ      PCHLT    ;YES
5D7A 3E45               MVI     A,'E'    ;EXEC ADDRESS
5D7C C3455A             JMP     ERR2
```

```
                      ;
                      ; THE NEXT LABEL CAN BE PLACED AFTER
                      ; THE LABEL CALLS/GO NEAR THE BEGINNING
                      ;
5D7F E9               PCHLT:  PCHL
                      ;
                      ; MAKE BINARY TEST TAPE, EACH BYTE
                      ; IS ONE LARGER THAN PREVIOUS BYTE
                      ; WITH OVERFLOW, 00 IS AFTER FF
                      ;
5D80 21F401           TMAKE:  LXI     H,500     ;ABORT CHECK
5D83 7A               TMAK2:  MOV     A,D       ;GET A BYTE
5D84 CD935C                   CALL    TOUT      ;SEND TO TAPE
5D87 14                       INR     D         ;INCREMENT IT
5D88 2B                       DCX     H         ;DECR COUNT
5D89 7C                       MOV     A,H
5D8A B5                       ORA     L         ;SEE IF ZERO
5D8B C2835D                   JNZ     TMAK2     ;NO
5D8E CD2558                   CALL    INSTAT    ;TERMINATION?
5D91 CA805D                   JZ      TMAKE     ;NO
5D94 CD1558                   CALL    INPUTT    ;^X FOR ABORT
5D97 C3805D                   JMP     TMAKE     ;NO
                      ;
                      ; READ BINARY TEST TAPE
                      ; EACH BYTE MUST BE ONE LARGER
                      ; THAN THE PREVIOUS ONE
                      ;
5D9A CDB25C           TTEST:  CALL    PIN       ;GET A BYTE
5D9D 57                       MOV     D,A       ;SAVE IT
5D9E CDB25C           TTEST2: CALL    PIN       ;NEXT BYTE
5DA1 14                       INR     D         ;INCR FIRST
5DA2 BA                       CMP     D         ;SAME?
5DA3 CA9E5D                   JZ      TTEST2    ;YES
5DA6 3E43                     MVI     A,'C'     ;NO
5DA8 CD2A58                   CALL    OUTT      ;C FOR ERROR
5DAB C39A5D                   JMP     TTEST     ;START AGAIN
                      ;
                      ; TAPE ERROR, PRINT B AND ADDRESS
                      ;
5DAE CDB45D           PERROR: CALL    POFF
5DB1 C3435A                   JMP     ERRB
                      ;
                      ; TURN OFF TAPE READER
                      ;
5DB4 3E13             POFF:   MVI     A,CTRS    ;READER OFF
5DB6 C3935C                   JMP     TOUT
                      ;
                      ; FILENAME ERROR, PRINT ACTUAL
                      ; FILENAME ON TAPE
                      ;
5DB9 205472793AEMESS:         DB      ' Try: ',0
5DC0 CDB45D           FNERR:  CALL    POFF      ;READER ON
5DC3 11B95D                   LXI     D,EMESS   ;ERROR MESSAGE
5DC6 CD3B59                   CALL    SENDM     ;SEND IT
5DC9 11A657                   LXI     D,IBUFF   ;FILE NAME
5DCC C33B59                   JMP     SENDM     ;SEND IT TOO
                      ;
5DCF                          END
```

Symbols:

5B07	ADMP2	5B23	ADMP3	5B26	ADMP4	5B04	ADUMP
5AED	ALOD2	FFF7	APOS	5AD5	ASCII	5B2C	ASCS
0008	BACKUP	5B4C	BIT2	5B4A	BITS	5A09	CALLS
0011	CDATA	0011	CDATAO	5A3E	CHEKM	5809	CIN
5858	COLD	5806	COUT	000D	CR	59DC	CRHL
590F	CRLF	5D5B	CSERR	0010	CSTAT	0010	CSTATO
0008	CTRH	0011	CTRQ	0013	CTRS	0018	CTRX
007F	DEL	5945	DUMP	5948	DUMP2	594B	DUMP3
595E	DUMP4	5967	DUMP5	5DB9	EMESS	0074	EOF
5A45	ERR2	5A43	ERRB	59D4	ERROR	5A42	ERRP
001B	ESC	5D60	EXEC	57C6	FBUF	5A5D	FILL
5A66	FILL2	5A6C	FILL3	5A75	FILL4	5DC0	FNERR
580F	GCHAR	5939	GETC4	5925	GETCH	5A08	GO
59F5	HEX1	5991	HHLDE	5A7C	HLDEBC	5A8A	HLDECK
5B73	HMATH	57A5	IBUFC	57A6	IBUFF	57A3	IBUFP
00DB	INC	580C	INLN	0001	INMSK	58D5	INPL2
58F1	INPL3	5919	INPLB	5901	INPLC	58FB	INPLE
58DD	INPLI	58D0	INPLN	581B	INPUT2	5815	INPUTT
5825	INSTAT	5B3D	IPORT	5BA5	JERR	5B85	JUST
5B89	JUST2	5B92	JUST3	5C88	LEADR	000A	LF
57E9	LFLAG	5A0D	LOAD	5A10	LOAD2	5A33	LOAD3
5A30	LOAD4	5A37	LOAD6	5A96	MOVDN	5A93	MOVE
5AA0	MOVIN	5878	MSIZE	59C4	NIB	5C8B	NLDR
586A	NPAGE	5CCA	OFFST	57E6	OFSET	0002	OMSK
5B5A	OPORT	5800	ORGIN	582B	OUT2	5839	OUT3
5844	OUT4	00D3	OUTC	5812	OUTH	59E4	OUTHEX
59DF	OUTHL	59EC	OUTHX	59E3	OUTLL	59E7	OUTSP
582A	OUTT	5981	PASC2	5983	PASC3	5976	PASCI
5D7F	PCHLT	5C47	PD0	5C5C	PD1	5C68	PD2
5C75	PD3	5C42	PD4	0007	PDATA	5DAE	PERROR
0001	PIMSK	5CB2	PIN	5CBC	PIN4	5CC1	PIN5
5DB4	POFF	0080	POMSK	57A0	PORTN	5C97	POUT
5C98	POUTW	0006	PSTAT	5B63	PUTIO	59A2	RDHL2
59B7	RDHL4	59C1	RDHL5	5989	RDHLD2	5986	RDHLDE
599D	READHL	5A4E	REGS	5BB4	REPL	5BC1	REPL2
5BCC	REPL3	5803	RESTRT	00C9	RETC	003C	RHEAD
0055	SBYTE	5AAD	SEAR2	5AB8	SEAR3	5ACF	SEAR4
5AC9	SEAR5	5AAA	SEARCH	593B	SENDM	584F	SIGNON
5D4A	SKIP	57A0	STACK	5800	START	0009	TAB
589C	TABLE	5BFA	TAPE	57E8	TASK	5C36	TDMP
5CA3	TIN4	5D1C	TL0	5D3E	TL1	5CDC	TLD1
5CFB	TLD2	5D12	TLD4	5CE7	TLD5	5CD2	TLOAD
5D83	TMAK2	5D80	TMAKE	0018	TOP	5C93	TOUT
5A00	TSTOP	5D9A	TTEST	5D9E	TTEST2	5BD2	VERM
5BD5	VERM2	5BF3	VERM3	3831	VERS	5861	WARM
5A55	ZERO						

There are seven tape commands that can be given. The appropriate command letter must immediately follow the letter T. The first thing that should be done is to make a test tape. In fact, the test pattern should be recorded onto the beginning of each tape you own. Give the command

```
>TM
```

(tape make), and start recording. The computer will send a sequence of bytes, each being one larger than the previous one. When the maximum value of FF hex has been sent, the next byte will be a zero. Record the pattern for 15 to 30 seconds. Type a control-X to end the procedure.

The integrity of the whole tape-recording system can now be tested. Rewind the test section of the tape and play it back. Type a command of

```
>TT
```

(tape test). The computer will read a byte from the tape, increment the value, then read another byte. If the two values don't agree, then a letter C will be printed on the console. The process will then be repeated until you terminate it.

If the two values do agree, then the computer will increment the first byte again, read another byte, and compare the two. This process will be repeated over and over. Each byte on the tape will be compared to see that it is exactly one larger than the previous byte. If you can go through a one-hour cassette recording without a single error, then you have a reliable system.

You may have an adjustment on the A/D converter circuit that allows you to set the frequency of the phase-locked loop (PPL). The MITS audio cassette boards have this feature. Make a test tape at least 10 minutes long, then play it back. Adjust the PPL frequency until a stream of Cs appear on the console; then adjust the PPL frequency in the opposite direction. The Cs should not print. Continue adjusting the PPL in the opposite direction again until the Cs appear again. You have now bracketed the usable range of the PPL. Set the PPL adjustment halfway between these two extremes.

There are some other tests you can make while the test tape is playing. Try moving both the audio cables and the AC line cords around to see if certain positions will cause an error. You can also try tapping the tape recorder, the computer case, the boards in the computer, and so on to see how delicate the system is. Pretty soon you will know if you have a reliable tape storage system. Remember to run the test tape at least once a week.

To save data from the computer's memory, including the tape program itself, type

```
>TD<start> <stop> <autostart> <filename>
```

After the D (for dump), give the start address, the stop address, the required autostart address, and an optional filename. A filename is important on magnetic tape to ensure that you are loading the right program. The autostart address can be the system monitor address. CP/M programs can be saved on tape using the original filename. Load the program into memory with DDT, SID, or the program FETCH given in the next chapter. Branch to the tape program and give the command

```
>TD100 9AF 100 TAPE.ASM
```

The filename can include the decimal point and the file type.

Each time a new tape is made, it should be verified with the command

```
>TV<filename>
```

A tape can be loaded into memory, at its normal position by typing

```
>TL<filename>
```

At the conclusion of the load, the autostart address is printed on the console then control returns to the monitor. If an incorrect filename is entered, the load operation is aborted and the actual filename on the tape is printed on the console.

During the load operation, all printable characters are displayed on the console. If your console is a printer rather than a video console, you will have to remove this feature to prevent the computer from falling behind. In this case, take out the following three lines from subroutine POUTW.

```
CPI     ' '
RC
OUT     CDATA
```

It is not necessary to verify the load step because of the checksum feature. If the load step was completed, then the tape was correctly read into memory.

The E command can be used instead of the L command for loading data. In this case the computer will branch to the autostart address at the completion of the load. A BASIC interpreter could be saved with the autostart address set to the BASIC start address. BASIC will then automatically start up at the completion of the E load command. The O command is used to load a tape at an offset from its regular address.

While this binary format is designed for magnetic tape systems, it can be used for paper tape as well. If a tape is punched on a Teletype machine, there will be a strange sound. The reason is that binary, rather than ASCII characters, are being punched. The printout will also be meaningless. But when the resulting tape is read back, the operation is quiet, since the inputted characters are not echoed back on the Teletype.

CHAPTER TEN

Linking Programs to the CP/M Operating System

The system monitor we developed in Chapters 6 and 7 allows us to communicate with the computer through the console. But this program only provides the bare minimum of operations such as memory display, block move, hex addition and subtraction, and so on. Computers are capable of performing more complicated tasks such as decimal arithmetic, keeping business records, formatting text, and game playing. Computer languages, such as FORTRAN, COBOL, and Pascal, make these tasks easier. These languages will operate on programs (called source programs) that are written by the user.

As an example, a BASIC interpreter, which may be as large as 24K bytes, needs a user's source program for direction. An assembler needs a source file to generate the desired machine code. And, of course, a formatting program needs a work file to produce a finished file.

Because programs to perform these common tasks are so large, an efficient method of loading them into memory is needed. In addition, the output generated by these programs needs to be stored somewhere. Magnetic tape can be used for this purpose, but it tends to be slow. The floppy disk currently is a better medium. It is relatively inexpensive, and the recording medium, the diskette, can be removed. This means that backup copies can be easily generated. In addition, programs can be readily exchanged between users.

We wrote our system monitor to transfer data between the console and the computer proper. Likewise, we need a disk-operating system (DOS) to effect an orderly transfer of information between the disk and the computer. Several different floppy disk systems are available for the 8080 and the Z-80. The disk-operating system that is provided with the disk drives may be relatively primitive, or it may be very elaborate.

A very popular DOS available today for the 8080 and Z-80 is called CP/M. CP/M was initially developed for the 8080 S-100 buss and the 8-inch soft-sectored floppy disk. It is now available for most of the other configurations, including the 5-inch floppies and the Winchester hard disk. Versions are supplied for computers that don't have an S-100 buss, such as the TRS-80.

CP/M is a software system that must be interfaced to each different computer configuration through a set of software interface routines. Once the interfacing is accomplished, the computer programs that run with CP/M become system independent.

The operating system integrates all of the common tasks. Before CP/M came along, each assembler and BASIC interpreter had to incorporate its own text editor. The user could then write and correct the source program with the built-in editor. But with the CP/M operating system, the editing can be performed separately from program execution. An independent system editor is used to create an ASCII source file. Then a processor program such as BASIC, FORTRAN, Pascal, an assembler, or a text-output formatter can be directed to utilize the source file. The results can be stored in a separate ASCII file on the disk.

It is possible to generate an assembly-language program with the system editor, then assemble it as we did when we developed the system monitor in Chapters 6 and 7. In this case we had to tailor the console input and output routines to the host computer. In particular, we included in the program the address of the console status and data ports, and the read-ready and write-ready status bits. The sense of the status flags was selected with a JZ command corresponding to an active-high flag. As a result of this approach, our system monitor is not portable. The monitor runs on your system, but it might not run on someone else's unless the I/O details are changed.

We approached a solution to this problem in Chapter 8, where we wrote some base-conversion routines. We did not have to write the I/O subroutines into each program because we utilized the ones in the monitor.

When we wrote the monitor, we placed five jump vectors at the beginning. These provided an easy access to the commonly used routines. While this technique greatly simplifies things, it has the disadvantage that three bytes must be set aside for each different entry point. Also, the programmer must keep track of which entry is used for which task.

A better approach is to have only a single entry address for all operations. When this special address is called by an external program, the values in the CPU registers indicate the desired operation and provide the data. The CP/M operating system utilizes this approach. All of the systems operations are performed by calling memory address 5. Up to 37 different operations can be performed in this way. Operations 1 through 11 allow interaction with the four logical peripherals named CONSOLE, LIST, PUNCH, and READER. They are summarized in Table 10.1. Operation 12 allows a program to determine the current CP/M version. The remaining function numbers are used for disk operations such as reading, writing, and head positioning.

The desired operation is selected by placing the proper function number into register C. Sixteen-bit data are transferred in the DE register.

Eight-bit data are transferred in Register E or the accumulator. For example, the console input status can be determined by placing the function number of 11 (decimal) in the C register, and calling address 5.

```
MVI     C,11
CALL    5
```

Table 10.1. CP/M I/O operations. The function number is placed into register C. Single bytes are in register E. Double bytes are in the D, E register pair.

Function number (in C)	Operation	Value sent	Value returned
1	read CONSOLE		char in A
2	write CONSOLE	char in E	
3	read READER		char in A
4	write PUNCH	char in E	
5	write LIST	char in E	
6	direct console I/O	FF (input; char (output)	char/stat in A
7	get I/O byte		IOBYTE in A
8	set I/O byte	IOBYTE in E	
9	print CONSOLE buffer	buffer address in D, E	
10	read CONSOLE buffer	buffer address in D, E	characters in buffer:
11	CONSOLE status		0 = not ready FF = ready

On return, the accumulator and register E contain the value of FF hex if the console is ready for input, or a zero if not. The byte in the console data register can then be read by calling address 5 with the value of 1 in register C. The byte that is read will be returned in the accumulator.

CP/M MEMORY ORGANIZATION

When CP/M is in operation, the main memory is divided into several regions. In order of decreasing memory, they are:

1. Basic Input/Output System (BIOS)
2. Basic Disk-Operating System (BDOS)
3. Console-Command Processor (CCP)
4. Transient-Program Area (TPA)
5. System parameter area

If the BIOS is especially tailored to a particular system, then it is called the customized BIOS or CBIOS. The BIOS and BDOS regions are collectively known as the Full Disk-Operating System (FDOS). The memory layout is shown in Figure 10.1.

```
                !---------------------! high memory
          / --  !        BIOS         !
FDOS     {      !---------------------!
          \ --  !        BDOS         !
                !---------------------!
                !        CCP          !
                !---------------------!
                !        TPA          !
                !---------------------! 100 hex
                ! system parameters   !
                !---------------------! low memory
```

Figure 10.1 A memory map for CP/M.

BIOS contains the tailor-made routines for operation of all the peripheral devices such as the console, printer, disk drives, phone modem, and tape drives. BDOS has the hardware-independent routines for the disk operating system. The CCP handles the console commands. It contains the built-in routines such as DIR, LIST, and ERA. Separate programs such as PIP, DDT, and user-written programs execute in the TPA.

The lowest memory area contains several different items. The first is a warm-boot entry to BIOS. The first few entries are:

Address	Action	Purpose
0	JMP BIOS+3	warm start entry
3	IOBYTE	peripheral assignment
4	disk	current drive number
5	JMP BDOS+6	peripheral control
5C hex	FCB	command-line argument

CHANGING THE PERIPHERAL ASSIGNMENT

CP/M can interact with four logical peripheral devices. These are termed:

CON: console
LST: list (printer)
PUN: punch
RDR: (tape) reader

Each of these four logical devices can be assigned to four different physical devices. The 16 different possible combinations can all be encoded into the IOBYTE located at memory address 3. The colon is part of the name; it is used when referring to peripheral devices. Disk filenames, on the other hand, are written without the colon.

The actual mapping of the logical devices into the desired physical devices must be accomplished in BIOS. Since there are so many different possibilities, it is not likely that the system supplied by your dealer will have the IOBYTE features fully implemented. The CP/M logical console is probably mapped into your video terminal and the logical list device will be mapped into your printer. But the punch and reader might be mapped into your video terminal.

Even if you have only two peripherals, it may be useful to set up the IOBYTE feature. For example, suppose that you have both a video terminal and a printer such as a Decwriter or Teletype that has its own keyboard. Then either peripheral can be used for the console input and either can be used for the console output. The four physical console arrangements could correspond to:

IOBYTE	Input	Output
0	video	video (default)
1	video	printer
2	printer	printer
3	printer	video

Your customized BIOS (CBIOS) will set the default combination on a cold start. But any user program can change the IOBYTE to a new configuration. Also, the CP/M programs STAT and DDT can be used to alter the IOBYTE. The instructions in CBIOS will sample the IOBYTE and act accordingly.

Suppose that the IOBYTE is set to zero, and you load a BASIC interpreter. You develop a program that sends information to the console. The program will include statements like

```
. . .
PRINT I, X(I), Y(I)
. . .
```

The resulting output appears on the console video screen. After the program is debugged, you would like a hard-copy output of the program results. This is easily done if you have incorporated the IOBYTE feature. You can change the IOBYTE with the POKE command in the BASIC interpreter.

```
POKE 3, 1
RUN
```

The POKE command will change the IOBYTE at address 3 from a zero to a 1. Console output will now appear at the printer when the BASIC program is run. Of course, console input has not changed; it still comes from the video terminal. At the conclusion of the run, the IOBYTE can be restored with another POKE command.

```
POKE 3, 0
```

As another example, suppose that you have a Teletype for your list device and it has a paper tape reader and punch. You can easily make BASIC tapes and reload them using the IOBYTE features. POKE the IOBYTE to a value of 1 for punching a source tape. This will map the Teletype output into the logical console output. Give the BASIC commands

```
NULL 3
LIST
```

to make a tape of the source program.

Later, you can reload the tape into BASIC by setting the IOBYTE to a value of 3.

```
POKE  3, 3
```

Just put the tape into the reader and turn it on. With this configuration, the Teletype keyboard and tape reader perform the console input. But the video screen is used for console output. After the tape has been read, give the command

```
POKE 3, 0
```

from the Teletype keyboard to return to the console keyboard.

INCORPORATING THE IOBYTE INTO YOUR CBIOS

You will have to reassemble your CBIOS if you want to incorporate the IOBYTE feature. A sample CBIOS that uses the IOBYTE is given in Listing 5.1. This version additionally features such things as an interrupt-driven keyboard and connection to a telephone modem.

At the beginning of BIOS there are seven jump vectors that are used by other parts of CP/M.

```
JMP     INIT      ;initialization
JMP     CONST     ;console status
JMP     CONIN     ;console input
JMP     CONOUT    ;console output
JMP     LOUT      ;list output
JMP     PUNCH     ;punch
JMP     READR     ;reader
```

Because of these fixed entry points, BIOS can be altered without having to alter any other part of CP/M. There are four regions of BIOS that need to be changed if the IOBYTE feature is incorporated into console routines. These are the routines for initialization, console status, console input, and console output. We have to set the IOBYTE to the default value in the initialization section of BIOS. Then, at the beginning of the three console routines, we have to read the IOBYTE at address 3, and branch to the appropriate subroutine.

In accordance with the step-by-step approach to programming we used in Chapter 6, we will not make all the changes at one time. The first alteration will be to the initialization and output routines.

Incorporating the IOBYTE into Output Routines

First locate the initialization region of BIOS. This area is referenced by the first jump instruction at the beginning of BIOS. Add the following instructions somewhere in the initialization routine.

```
MVI     A,0     ;set a zero
STA     3       ;SET IOBYTE
```

The safest places to put them are at the very beginning of the initialization section or at the end, just before the RET instruction. We could, of course, use an XRA A instruction rather than the MVI A,0 instruction shown above. This would reduce the size of BIOS by one byte. But if we later wanted BIOS to initialize the IOBYTE to 3 we would have to reassemble BIOS. With the MVI instruction, we can easily change the argument of zero to a 3 by using the system debugger.

Refer to the label START at the beginning of Listing 4.1 where there is a series of jump instructions.

```
        . . .
START:
        JMP     INIT    ;INITIALIZATION
        JMP     CONST   ;CONSOLE STATUS
        JMP     CONIN   ;CONSOLE INPUT
        JMP     CONOUT  ;CONSOLE OUTPUT
        . . .
```

Locate the address of the console-output routine. This can be found from the fourth jump instruction, JMP CONOUT in this case. We will now alter the console-output routine so that it will read the IOBYTE at memory address 3 and act accordingly. The upper six bits, which refer to the list, punch, and reader, are reset by performing a masking AND with the value of 3.

```
XXXX XX01   or   XXXX XX10   IOBYTE
       11               11   AND with 3
---------        ---------
0000 0001   or   0000 0010   IOBYTE
```

The two low-order bits then determine whether output is sent to the console or to the printer. If the resulting value is a 1 or a 2, then a branch is made to the printer-output routine. Otherwise, output goes to the video console. Printer output corresponds to the following IOBYTE bit patterns.

```
0000 0001   or    0000 0010    IOBYTE
       11                11    AND with 3
---------         ---------
0000 0001 = 1     0000 0010   = 2
```

The other two possible IOBYTE patterns correspond to console output.

```
0000 0000 = 0   or   0000 0011 = 3
```

Notice that parity is odd for printer output but parity is even for console output. Thus the BIOS code at this point is programmed to jump to the list-output routine if parity is odd.

```
;
; LOGICAL CONSOLE OUTPUT
;
CONOUT: LDA      IOBYTE   ;GET ASSIGNMENT
        ANI      3        ;MASK FOR CONSOLE
        JPO      LOUT     ;LIST OUTPUT
;
; VIDEO-OUTPUT ROUTINE
;
VIDEO:. . . (existing routine)
        . . .
;
; LIST-OUTPUT ROUTINE
;
LOUT:   . . . (existing routine)
        . . .
```

Assemble the new BIOS and try it out. We can use the system debugger DDT or SID to load the new CBIOS over the old one. The command is

```
A>DDT CBIOS.HEX
```

One word of caution: Since the debugger uses the routines in CBIOS, there may be a problem. A safer way to put the new CBIOS into place is to first load it somewhere else.

Use the hex arithmetic command of DDT to calculate the offset. Suppose that BIOS is assembled for the address DB00 hex and you want to load it temporarily at 100 hex. Then the command

```
H100 DB00
```

will give both the sum and difference of the two addresses. The difference is what we need. Load CBIOS with the indicated offset

```
ICBIOS.HEX
R(offset)
```

DDT will indicate the location of the end of BIOS. The move command can now be used to put BIOS into its proper place.

```
M100  2FF DB00
```

The second address will be the present end of BIOS. The new BIOS is now in place. If the debugger no longer works, there may be an error in BIOS or there may have been an error in the move command.

If everything appears to be all right, try out the IOBYTE feature. Use the S (set) command of the debugger to change the IOBYTE at address 3. Change the value of the IOBYTE from a zero to a 1.

```
S3
0003  00   1 (type a 1)
0004  00   . (type a decimal point)
```

The (logical) console output should now appear at the printer instead of the console. Notice that this is different from typing a control-P when the IOBYTE has a value of zero. In fact, if a control-P is typed at this time, each typed character should be printed twice. Return the logical console output to the console by changing the IOBYTE back to zero using DDT.

```
S3
0003  01   0 (type a 0)
0004  00   . (type a decimal point)
```

Output should return to the console.

If everything appears to be all right, you are still not finished. You have a working copy of BIOS in memory; now you must get a copy onto the system tracks of the disk. *Continue with this section if you want to write a permanent copy of the new BIOS onto the disk. Otherwise, go on to the second part of the alteration where you will add the IOBYTE feature to the console input routines.* Then come back to this section to save the completed BIOS. If you have a Lifeboat version of CP/M, you can easily copy the new version of BIOS to the disk. Just give the command

```
A>SAVEUSER
```

and the new BIOS will be copied to the system disk.

Another method of getting the new CBIOS on the disk system tracks is to use SYSGEN. The current version of CPM is loaded into memory with the DDT command

```
A>DDT CPMXX.COM
```

where XX refers to the memory size. Then move the working version of BIOS down to the proper SYSGEN position using the DDT move command. Alternately, the HEX file of CBIOS could be copied from disk into memory.

The SYSGEN location for BIOS should be given in your CP/M documentation. Perform a warm start with a control-C, and then give the command

```
A>SYSGEN
```

Answer the first question about where to get the system by typing just a carriage return. The second question asks where to put the system. Give the appropriate drive name, A, B, etc. The new BIOS will now be copied to disk.

Incorporating the IOBYTE into Input Routines.

We are now ready to implement the second part of the console IOBYTE feature. This will allow the logical console input to be obtained from either of two keyboards. One of these keyboards will be the video terminal; the other will be the printer. If you don't have a second keyboard, go on to the next section.

Locate the console status and the console input routines. These are found from the second and third entries of the jump table of BIOS. Your present console-input routine may be coded in one of two ways. One method is straight-forward. The input routine has a status check independent of the regular BIOS status routine.

```
;
; CONSOLE INPUT ROUTINE
;
CONIN:  IN      CSTAT   ;GET STATUS
        ANI     CIMSK   ;MASK FOR INPUT
        JZ      CONIN   ;LOOP UNTIL READY
        . . .
```

The other method uses the BIOS status routine.

```
;
; CONSOLE INPUT ROUTINE
;
CONIN:  CALL    CONST   ;GET STATUS
        JZ      CONIN   ;LOOP UNTIL READY
        . . .
```

Although either method can be altered to include the IOBYTE feature, the former method will be demonstrated here. We will need two separate input routines. One is for the video screen and the other is for the printer or other keyboard. Our new console input routine might look like this.

```
;
; LOGICAL CONSOLE INPUT
;
CONIN:  LDA     IOBYTE  ;GET ASSIGNMENT
        ANI     2       ;MASK FOR LIST
        JNZ     LISTIN  ;LIST INPUT
```

```
;
; VIDEO INPUT
;
VIDIN:  IN      CSTAT   ;GET STATUS
        ANI     CIMSK   ;MASK FOR INPUT
        JZ      VIDIN   ;LOOP UNTIL READY
        IN      CDATA   ;GET DATA
        ANI     7FH     ;MASK PARITY
        RET
;
; INPUT FROM PRINTER
;
LISTIN: IN      LSTAT   ;GET STATUS
        ANI     LIMSK   ;MASK FOR INPUT
        JZ      LISTIN  ;LOOP UNTIL READY
        IN      CDATA   ;GET DATA
        ANI     7FH     ;MASK PARITY
        RET
```

The third jump statement at the beginning of BIOS should branch to the label CONIN. The IOBYTE at memory address 3 is read. All bits but bit 1 (the second bit) are zeroed with the ANI 2 command. List input corresponds to an IOBYTE of 2 or 3. The logical AND with 2 and either value will produce the result of 2.

```
0000 0010   or   0000 0011   IOBYTE
       10               10   AND with 2
---------        ---------
0000 0010        0000 0010   = 2
```

The JNZ instruction will then cause a branch to the list-input routine. Otherwise, program flow will continue on to the console-input routine.

A similar construction can be used for the console status routine.

```
;
; LOGICAL CONSOLE-INPUT STATUS
;
CONST:  LDA     IOBYTE  ;GET ASSIGNMENT
        ANI     2       ;MASK FOR LIST
        JNZ     LISTST  ;LIST
;
; INPUT FROM VIDEO
;
VSTAT:  IN      CSTAT   ;CHECK STATUS
        ANI     CIMSK   ;MASK FOR INPUT
        RZ              ;NO READY
        MVI     A,0FFH  ;SET FOR READY
        RET
;
; INPUT FROM LIST
;
LISTST: IN      LSTAT   ;CHECK STATUS
        ANI     LIMSK   ;MASK FOR INPUT
        RZ              ;NOT READY
        MVI     A,0FFH  ;SET FOR READY
        RET
```

Assemble the new version of BIOS and try it out as we did in the previous section. Load the debugger, then use it to change the IOBYTE at address 3. First, change the IOBYTE to a value of 3. This will assign the logical console input to the printer (or other alternate keyboard) and logical console output to the video screen. As soon as you make the change, all further input must come from the printer. Use the debugger to change the IOBYTE to a value of 2. Now logical console input must come from the printer, and logical console output will appear at the printer. Finally, change the IOBYTE back to the value of zero. Console input and output should both go through the video terminal.

USING STAT TO CHANGE THE IOBYTE

The CP/M program called STAT can be used to symbolically change the IOBYTE. STAT can also be used to determine the current value of the IOBYTE. Each of the four logical devices CON: (console), RDR: (reader), PUN: (punch), and LST: (list) can be assigned to four different physical devices.

Logical device		Physical device			
		1	2	3	4
CON:	console	TTY:	CRT:	BAT:	UC1:
RDR:	reader	TTY:	PTP:	UR1:	UR2:
PUN:	punch	TTY:	PTP:	UP1:	UP2:
LST:	list	TTY:	CRT:	LPT:	UL1:

The first column represents the four logical peripherals. The remaining entries on each line represent the four physical devices. You can easily change these names in STAT to something more descriptive. Load STAT into memory with the debugger.

```
DDT STAT.COM
```

Then dump the first few lines.

```
D100 16F
```

You will see the names of the logical peripherals and the physical peripherals encoded in ASCII. You can now change any of the names to something else. You might want to choose the names

CRT:
LST:
LPT:
LCR:

for the four console devices. These four names correspond to the IOBYTE values of 0 through 3. After you change the names with the debugger, return to CP/M and save the altered STAT. If the IOBYTE is currently set to zero and you type the command

```
A>STAT  CON:=LST:
```

STAT will change the IOBYTE to a value of 1 and console output will appear at the printer. The IOBYTE can be set back to zero with the command

```
A>STAT CON:=CRT:
```

If you have other peripherals, such as a phone modem, these can also be incorporated into IOBYTE. The logical punch can then be sent to the modem, the printer, or the console. A disk file can be sent over the phone modem with the command

```
A>STAT  PUN:=B:CPMIO.ASM
```

where CPMIO.ASM is a disk file on drive B. As you can see, there is room for a lot of imagination.

A ROUTINE TO GO ANYWHERE IN MEMORY

The assembly language program shown in Listing 10.1 can be used to branch to any location in memory. Executable programs in CP/M are usually designed to be run starting at address 100 hex, since this is where programs are loaded by CP/M. There are times, however, when it is desirable to assemble a program for operation at some other location. The system monitor developed in Chapter 6 is one such program. If this monitor is to be located in ROM, then it must be placed above the CP/M operating system. A location of F000 hex would be ideal. But there is no easy way to get to the monitor from the CP/M system. If the GO routine is located on disk drive A, then we have only to give the CP/M command

```
A>GO  F000
```

and a branch will occur to the address F000 hex.

The GO program demonstrates several of the I/O features available with CP/M. The branch address given as an argument on the above command line is read from a region of memory called the file-control block (FCB). The address of the FCB is 5C hex but the ASCII-encoded address starts at 5D hex. The input address, F000 in this example, is a valid hexadecimal number; it is to be converted from ASCII into a 16-bit binary number. The result is placed into the CPU program counter so that the computer will branch to the desired address. The address format is free-form, and leading zeros are

Listing 10.1 A program to go anywhere in memory.

```
                ; (date goes here)
                ;
                        TITLE   'GO (jump anywhere)'
                ;
                ; USAGE: TYPE GO F800 TO JUMP TO F800 HEX
                ;
0100            ORG     100H
                ;
0005 =          BDOS    EQU     5       ;DOS ENTRY POINT
005C =          FCB     EQU     5CH     ;FILE CONTROL BLOCK
0009 =          PBUF    EQU     9       ;PRINT BUFFER
000A =          RDBUF   EQU     10      ;READ CONSOLE BUFFER
000D =          CR      EQU     0DH     ;CARRIAGE RETURN
000A =          LF      EQU     0AH     ;LINE FEED
                ;
                START:
0100 215D00             LXI     H,FCB+1 ;GET ARGUMENT IF ANY
0103 7E                 MOV     A,M     ;FIRST BYTE
0104 FE20               CPI     ' '     ;BLANK?
0106 CA3B01             JZ      ERROR   ;NO ARGUMENT
0109 229B01     AGAIN:  SHLD    RBUFP   ;SAVE POINTER
010C CD1001             CALL    READHL  ;GET ADDRESS
010F E9                 PCHL            ;GO TO ADDRESS
                ;
                ; CONVERT ASCII-HEX CHARACTERS
                ; TO 16-BIT BINARY NUMBER IN H,L
                ;
0110 210000     READHL: LXI     H,0     ;START WITH 0
0113 CD6301     RDHL2:  CALL    GETCH   ;GET A BYTE
0116 FE20               CPI     ' '     ;END?
0118 C8                 RZ              ;YES
0119 CD2801             CALL    NIB     ;TO BINARY
011C DA3801             JC      RDHL4   ;NOT HEX
011F 29                 DAD     H       ;TIMES 2
0120 29                 DAD     H       ;TIMES 4
0121 29                 DAD     H       ;TIMES 8
0122 29                 DAD     H       ;TIMES 16
0123 B5                 ORA     L       ;COMBINE NEW
0124 6F                 MOV     L,A     ;PUT BACK
0125 C31301             JMP     RDHL2   ;NEXT
                ;
                ; CONVERT ASCII TO BINARY
                ;
0128 D630       NIB:    SUI     '0'     ;ASCII BIAS
012A D8                 RC              ; < 0
012B FE17               CPI     'F'-'0'+1
012D 3F                 CMC
012E D8                 RC              ; > F
012F FE0A               CPI     10
0131 3F                 CMC
0132 D0                 RNC             ;A NUMBER 0-9
0133 D607               SUI     'A'-'9'-1
0135 FE0A               CPI     10
0137 C9                 RET
                ;
```

```
                        ; BLANK AT END OF LINE IS OK
                        ; ELSE AN ERROR
                        ;
0138 FEF0               RDHL4:  CPI     ' '-'0'
013A C8                         RZ
                        ;
                        ; IMPROPER ARGUMENT, TRY AGAIN
                        ;
013B 117501             ERROR:  LXI     D,MESG  ;POINT TO MESSAGE
013E CD5901                     CALL    PRINT   ;SEND IT
0141 119D01                     LXI     D,RBUFM ;INPUT BUFFER
0144 CD5E01                     CALL    READB   ;GET A LINE
0147 1600                       MVI     D,0
0149 3A9E01                     LDA     RBUFL   ;BUFFER LENGTH
014C 5F                         MOV     E,A
014D 219F01                     LXI     H,RBUF
0150 19                         DAD     D       ;PAST BUFFER
0151 3620                       MVI     M,' '   ;PUT IN BLANK
0153 219F01                     LXI     H,RBUF
0156 C30901                     JMP     AGAIN   ;TRY AGAIN
                        ;
                        ; PRINT CHARACTERS UNTIL $ IS FOUND
                        ;
0159 0E09               PRINT:  MVI     C,PBUF  ;SET FOR PRINT
015B C30500                     JMP     BDOS
                        ;
                        ; INPUT A LINE FROM CONSOLE
                        ;
015E 0E0A               READB:  MVI     C,RDBUF ;READ INPUT BUFFER
0160 C30500                     JMP     BDOS
                        ;
                        ; GET A CHARACTER FROM THE INPUT BUFFER
                        ;
0163 E5                 GETCH:  PUSH    H
0164 2A9B01                     LHLD    RBUFP   ;GET POINTER
0167 7E                         MOV     A,M     ;GET NEXT CHAR
0168 23                         INX     H       ;INCREMENT POINTER
0169 229B01                     SHLD    RBUFP   ;SAVE POINTER
016C FE61                       CPI     'Z'+7   ;UPPER CASE?
016E DA7301                     JC      GETC2   ;NO
0171 E65F                       ANI     5FH     ;MAKE UPPER CASE
0173 E1                 GETC2:  POP     H
0174 C9                         RET
                        MESG:
0175 474F206572                 DB      'GO error. Input '
0185 7468652061                 DB      'the address again.'
0197 0D0A2A24                   DB      CR,LF,'*$'
                        ;
019B 9F01               RBUFP:  DW      RBUF    ;BUFFER POINTER
019D 0A                 RBUFM:  DB      10      ;MAX SIZE
019E                    RBUFL:  DS      1       ;ACTUAL SIZE
019F                    RBUF:   DS      1       ;INPUT BUFFER
                        ;
01A0                            END
```

unnecessary. If more than four characters are entered, only the last four are used. Thus, all of the following are valid.

```
0        F000
900      PF800
6000
```

If no argument is given to the GO command, or if an invalid hexadecimal address is entered, then an error message is printed. This step uses the console string-output feature, selected with the function code of 9 in register C. The DE register pair is loaded with a pointer to the string location in memory. A dollar sign ($) is used to indicate the end of the string. In subroutine SENDM of Chapter 6, a binary zero is used for this purpose since it requires less code.

The user can retype the desired address after the error message has been printed. This time, however, the program reads the input string data in a different way. The console string-input operation is selected by loading the C register with the function number of 10. If the new string is a valid hex number, it is converted to a 16-bit binary number. The computer then jumps to this address. If the input is still invalid, the error routine is repeated again.

During the input operation, the usual CP/M commands are available for error correction. For example, the most recently typed character can be deleted by pressing the DEL key. A control-R will reprint the current line in its corrected form. A control-U cancels the entire line so that it can be retyped. If Version 2 of CP/M is being used, then the backspace character, control-H, can also be used for correcting errors. Finally, a control-C can be entered to abort the entire program. This returns control to the CP/M operating system.

Enter the GO program in Listing 10.1. Assemble it and try it out. GO is a universal program; it will work on any system. If you have a monitor located in memory, use GO to branch to it; if not, you can still try out the GO program. Type just the command of GO without an argument. An error message should be printed. Type some characters, then delete some of them with the DEL key. The deleted characters will be printed a second time. Reprint the line with a control-R to see the correct version. Finally return to CP/M by giving the address of zero.

```
GO 0
```

A control-C can also be used to return to CP/M.

A LIST ROUTINE WITH DATE AND TIME

In Chapter 7 we developed a monitor routine for sending data to a separate list device. When this routine is activated with a control-P command, the output appears at both the console and the printer. CP/M has a similar arrangement. A control-P command will activate the list device, too. Output is then sent to both the console and the printer.

But the console and list routines are entirely separate in CP/M. Output can therefore be sent specifically to the list device and it will not appear on the console. In CP/M I/O operations, a function number of 2 corresponds to console output, and a function number of 5 corresponds to list output. If memory address 5 is called with a value of 2 in the C register, then the byte in the E register will be displayed on the system console. On the other hand, if a function number of 5 is placed in the C register, then the byte in register E will appear on the list device. The byte in the E register can be sent to the punch by loading a function number of 4 into the C register.

This complexity may seem unnecessary, since the list device can be turned on by typing a control-P. If the command

```
A>TYPE <filename>
```

were given, then the file would be sent to both the console and the printer. The disadvantage of this method is that the TYPE command line will appear on the printer output. Also, when the listing is finished, the new prompt of A> will be printed on the listing.

The CP/M utility program PIP can be used to send a disk file specifically to the list device. The command is

```
A>PIP  LST:=<filename>[T8]
```

The argument T8, embedded in brackets, will expand the ASCII tab character to 8-column fields. PIP, however, does not automatically eject a new page when a form feed is encountered.

The LIST program in Listing 10.2 can be used to send an ASCII disk file to the printer, too. It will automatically expand the ASCII tab character. Furthermore, when a form-feed character is encountered, LIST will add the correct number of lines to the end of the page. At the end of the disk file, additional line feeds are issued to finish the page. This will ensure that the printer will start the next task on a new page. If the file contains an odd number of pages, then an extra blank page is added to the end. Without this feature you will have to refold about half of the output.

If your computer keeps track of the date and time, LIST can print current values at the top of the first page. The name of the disk file is also printed on this line.

LIST is a very small program, requiring only 1K bytes of memory. It was derived from the program called DUMP in the CP/M *Interface Guide.* LIST bears only a passing resemblance to DUMP, however. DUMP is designed to convert binary files to ASCII-hex characters and display them on the console. LIST, on the other hand, prints ASCII files directly with no conversion. Since the CP/M BDOS is used for all I/O and disk operations, LIST will operate with all standard CP/M systems.

Listing 10.2. List an ASCII disk file.

```
                        ; SEND ASCII DISK FILE TO LIST
                        ; PUT DATE AND TIME AT TOP OF PAGE
                        ; ENTIRE FILE LOADS INTO MEMORY FIRST
                        ; FORMFEEDS ADDED WITH F ARGUMENT
                        ; EXTRA PAGE ADDED TO MAKE TOTAL EVEN
                        ;  UNLESS A P OPTION IS GIVEN
                        ;
                        ; USAGE:
                        ;       LIST <filename>
                        ;       LIST <filename> F (add form feeds)
                        ;       LIST <filename> P (no extra page)
                        ;       LIST <filename> n (skip n lines)
                        ;
                        ; (date goes here)
                        ;
                        TITLE   'List an ASCII disk file.'
                        ;
0100                    ORG     100H
0005 =                  BDOS    EQU     5       ;DOS ENTRY POINT
0001 =                  CONS    EQU     1       ;READ CONSOLE
0005 =                  TYPEF   EQU     5       ;LIST OUTPUT
0009 =                  PBUF    EQU     9       ;PRINT CONSOLE BUFFER
000B =                  BRKF    EQU     11      ;KILL? (TRUE IF CHAR)
000F =                  OPENF   EQU     15      ;FILE OPEN
0014 =                  READF   EQU     20      ;READ FUNCTION
000D =                  CR      EQU     0DH     ;CARRIAGE RETURN
000A =                  LF      EQU     0AH     ;LINE FEED
001A =                  EOF     EQU     1AH     ;END OF FILE
0009 =                  TAB     EQU     9       ;^I
000C =                  FORMFD  EQU     0CH     ;FORM FEED
003A =                  LINES   EQU     58      ;LINES/PAGE
0042 =                  LMAX    EQU     66      ;MAX LINES
                        ;
005C =                  FCB     EQU     5CH     ;FILE CONTROL BLOCK
0080 =                  BUFF    EQU     80H     ;DISK BUFFER ADDR
                        ;
                        ; FILE-CONTROL BLOCK DEFINITIONS
                        ;
005D =                  FCBFN   EQU     FCB+1   ;FILE NAME
0068 =                  FCBRL   EQU     FCB+12  ;CURRENT REEL #
007C =                  FCBCR   EQU     FCB+32  ;NEXT REC #(0-127)
                        ;
                        ; TIME AND DATE FROM A COMPU/TIME BOARD
                        ;
00C4 =                  ADATA   EQU     0C4H    ;PORT A DATA
00C5 =                  ACONT   EQU     ADATA+1 ;PORT A CONTROL
00C7 =                  BCONT   EQU     ADATA+3 ;PORT B CONTROL
00C6 =                  BDATA   EQU     ADATA+2 ;PORT B DATA
                        ;
                        ; SAVE OLD STACK AND SET UP A NEW ONE
                        ;
0100 210000             START:  LXI     H,0
0103 39                         DAD     SP
0104 22E004                     SHLD    OLDSP   ;SAVE STACK
0107 310005                     LXI     SP,STACK
010A 111E04                     LXI     D,RULES
010D CD3E02                     CALL    PRINT   ;HOW TO ABORT
```

```
                ;
                ; HOW MANY LINES TO SKIP BEFORE STARTING?
                ;
0110 210000             LXI     H,0
0113 016D00             LXI     B,6DH   ;3RD ARGUMENT
0116 0A         SKIP2:  LDAX    B       ;GET CHARACTER
0117 FE20               CPI     ' '     ;BLANK AT END
0119 CA4001             JZ      SKIP3   ;DONE
011C FE46               CPI     'F'     ;NEED FORM FEEDS?
011E CA3C01             JZ      FORM    ;YES
0121 FE50               CPI     'P'     ;EXTRA PAGE?
0123 CA3601             JZ      NOPAGE  ;NO
0126 D630               SUI     '0'     ;REMOVE ASCII BIAS
                ;
                ; CONVERT ASCII DECIMAL TO BINARY IN H,L
                ;
0128 54                 MOV     D,H     ;DUPLICATE
0129 5D                 MOV     E,L     ;H,L IN D,E
012A 29                 DAD     H       ;TIMES 2
012B 29                 DAD     H       ;TIMES 4
012C 19                 DAD     D       ;TIMES 5
012D 29                 DAD     H       ;TIMES 10
012E 5F                 MOV     E,A
012F 1600               MVI     D,0
0131 19                 DAD     D       ;ADD NEW BYTE
0132 03                 INX     B       ;INCR POINTER
0133 C31601             JMP     SKIP2   ;NEXT
                ;
0136 32D904     NOPAGE: STA     PFLAG   ;NO EXTRA PAGE
0139 C33F01             JMP     ZERO
                ;
013C 32D804     FORM:   STA     FFLAG   ;SET FOR FORM FEEDS
013F AF         ZERO:   XRA     A       ;RESET COUNT
0140 22DA04     SKIP3:  SHLD    SKIPB   ;SAVE BINARY CNT
                ;
                ; READ AS MUCH AS POSSIBLE INTO MEMORY
                ;
0143 CDB302             CALL    SETUP   ;SET UP INPUT FILE
0146 3E80               MVI     A,80H
0148 32DE04             STA     IBP     ;SET POINTER TO 80H
014B 32D304             STA     TIME2   ;SET 1ST PASS
                ;
014E 2ADA04             LHLD    SKIPB   ;HOW MANY LINES?
0151 7C         MAIN6:  MOV     A,H
0152 B5                 ORA     L       ;H,L = 0?
0153 CA6401             JZ      MAIN5   ;NO SKIP
0156 E5         MAIN7:  PUSH    H
0157 CD4302             CALL    GNB     ;NEXT BYTE
015A E1                 POP     H
015B FE0D               CPI     CR
015D C25601             JNZ     MAIN7   ;LOOK FOR CR
0160 2B                 DCX     H       ;DECR COUNT
0161 C35101             JMP     MAIN6
                ;
0164 AF         MAIN5:  XRA     A
0165 32D504             STA     FULL    ;RESET FLAG
0168 CD4302     MAIN2:  CALL    GNB     ;GET A BYTE
016B E5                 PUSH    H
```

```
016C 2ADC04             LHLD    BUFFP   ;MEMORY POINTER
016F 77                 MOV     M,A     ;PUT BYTE IN
0170 23                 INX     H
0171 22DC04             SHLD    BUFFP   ;SAVE POINTER
0174 47                 MOV     B,A
0175 3EFF               MVI     A,0FFH
0177 BD                 CMP     L       ;L=0?
0178 C28B01             JNZ     MAIN4   ;NO
                ; CHECK FDOS
017B 3A0700             LDA     7       ;FDOS
017E D60A               SUI     10      ;CCP -1
0180 BC                 CMP     H       ;TOO BIG?
0181 D28B01             JNC     MAIN4   ;NO
0184 3E1A               MVI     A,EOF
0186 77                 MOV     M,A     ;PUT IN MEMORY
0187 32D504             STA     FULL    ;SET FOR FULL
018A 47                 MOV     B,A
018B 78         MAIN4:  MOV     A,B     ;GET BYTE
018C E1                 POP     H
018D FE1A               CPI     EOF
018F C26801             JNZ     MAIN2   ;NO
                ;
                ; CHECK FOR EOF AT END
                ;
0192 2ADC04             LHLD    BUFFP   ;GET POINTER
0195 2B                 DCX     H
0196 3E1A               MVI     A,EOF
0198 BE                 CMP     M       ;EOF?
0199 CAA101             JZ      MAIN3   ;YES
019C 23                 INX     H
019D 77                 MOV     M,A     ;PUT IN EOF
019E 22DC04             SHLD    BUFFP
                ;
01A1 CD7C02     MAIN3:  CALL    RESET   ;POINTER
                ;
                ; PUT TIME AT START OF LISTING
                ; REMOVE FORMFEED IF FIRST
                ;
01A4 CD5D03             CALL    CLOCK   ;GET TIME
01A7 3AD804             LDA     FFLAG   ;FORMFEEDS?
01AA B7                 ORA     A
01AB C4F601             CNZ     TWOLN   ;YES
01AE CD5F02             CALL    GETB    ;GET BYTE
01B1 FE0C               CPI     FORMFD  ;FORMFEED
01B3 C2BC01             JNZ     GLOP2   ;NO
01B6 CDF601             CALL    TWOLN   ;SEND 2 LF
                ;
01B9 CD5F02     GLOOP:  CALL    GETB    ;GET NEXT BYTE
01BC 47         GLOP2:  MOV     B,A     ;SAVE BYTE
01BD CDFD02             CALL    TABO    ;PRINT BYTE
01C0 78                 MOV     A,B     ;GET BYTE AGAIN
01C1 FE0A               CPI     LF      ;END OF LINE?
01C3 C2B901             JNZ     GLOOP   ;NO
01C6 3AD604             LDA     LCOUNT  ;GET COUNT
01C9 3C                 INR     A       ;INCREMENT IT
01CA 32D604             STA     LCOUNT  ;SAVE IT
01CD FE42               CPI     LMAX    ;TOO MANY?
01CF D43202             CNC     NPAGE   ;YES, RESET
01D2 3AD804             LDA     FFLAG   ;FORM FEEDS?
```

```
01D5 B7                 ORA     A
01D6 CAB901             JZ      GLOOP    ;NO
01D9 3AD604             LDA     LCOUNT   ;GET COUNT
01DC FE3A               CPI     LINES    ;END OF PAGE?
01DE DAB901             JC      GLOOP    ;NO
01E1 CD1B02             CALL    FILL     ;NEW PAGE
01E4 CDF601             CALL    TWOLN
01E7 C3B901             JMP     GLOOP
                ;
                ; CHECK FOR ABORT, ANY KEY PRESSED
                ;
01EA E5         ABORT:  PUSH    H        ;SAVE
01EB D5                 PUSH    D
01EC C5                 PUSH    B
01ED 0E0B               MVI     C,BRKF   ;CONSOLE READY?
01EF CD0500     PCHAR2: CALL    BDOS
01F2 C1                 POP     B        ;RESTORE
01F3 D1                 POP     D
01F4 E1                 POP     H
01F5 C9                 RET
                ;
01F6 060A       TWOLN:  MVI     B,LF     ;TWO LINES
01F8 3AD604             LDA     LCOUNT
01FB 3C                 INR     A        ;ADD 2 TO
01FC 3C                 INR     A        ;COUNT
01FD 32D604             STA     LCOUNT
0200 CD0302     OUTT2:  CALL    OUTT     ;DOUBLE OUTPUT
                ;
0203 78         OUTT:   MOV     A,B      ;OUTPUT FROM B
                ;
                ; SEND CHARACTER FROM A TO LIST
                ;
0204 E5         PCHAR:  PUSH    H
0205 D5                 PUSH    D
0206 C5                 PUSH    B        ;SAVED
0207 0E05               MVI     C,TYPEF  ;LIST
0209 5F                 MOV     E,A
020A FE0C               CPI     FORMFD   ;FORMFEED?
020C C2EF01             JNZ     PCHAR2   ;PRINT BYTE
                ;
                ; FILL OUT PAGE AFTER FORMFEED
                ;
020F CD1B02             CALL    FILL
0212 060A               MVI     B,LF     ;FOR FORMFEED
0214 CD0302             CALL    OUTT
0217 C1                 POP     B
0218 D1                 POP     D
0219 E1                 POP     H
021A C9                 RET
                ;
                ; FILL OUT END OF PAGE
                ;
021B 3AD604     FILL:   LDA     LCOUNT   ;LINE COUNT
021E 4F                 MOV     C,A
021F CD3202             CALL    NPAGE    ;INCR PAGE
0222 3E42               MVI     A,LMAX
0224 91                 SUB     C
0225 D8                 RC               ;TOO BIG
```

```
0226 C8                  RZ
0227 4F                  MOV     C,A
0228 060A                MVI     B,LF
022A CD0302     FILL2:   CALL    OUTT    ;SEND LF
022D OD                  DCR     C
022E C22A02              JNZ     FILL2
0231 C9                  RET
                ;
                ; RESET LINE COUNT, INCREMENT PAGES
                ;
0232 AF         NPAGE:   XRA     A       ;GET A ZERO
0233 32D604              STA     LCOUNT  ;LINE COUNT
0236 3AD704              LDA     PAGES   ;PAGE COUNT
0239 3C                  INR     A
023A 32D704              STA     PAGES
023D C9                  RET
                ;
                ; SEND MESSAGE TO CONSOLE
                ;
023E 0E09       PRINT:   MVI     C,PBUF
0240 C30500              JMP     BDOS
                ;
                ; GET NEXT BYTE FROM DISK BUFFER
                ;
0243 3ADE04     GNB:     LDA     IBP
0246 FE80                CPI     80H
0248 C24F02              JNZ     READ
                ;
                ; READ ANOTHER BUFFER
                ;
024B CDDA02              CALL    DISKR
024E AF                  XRA     A
                ;
                ; READ THE BYTE AT BUFF+REG A
                ;
024F 5F         READ:    MOV     E,A
0250 1600                MVI     D,0
0252 3C                  INR     A
0253 32DE04              STA     IBP
                ; POINTER IS INCREMENTED
                ; SAVE THE CURRENT FILE ADDRESS
0256 E5                  PUSH    H
0257 218000              LXI     H,BUFF
025A 19                  DAD     D
025B 7E                  MOV     A,M
                ; BYTE IS IN THE ACCUMULATOR
                ; RESTORE FILE ADDRESS AND INCREMENT
025C E1                  POP     H
025D 23                  INX     H
025E C9                  RET
                ;
                ; GET A BYTE FROM MEMORY BUFFER
                ;
025F E5         GETB:    PUSH    H
0260 2ADC04              LHLD    BUFFP
0263 7E                  MOV     A,M
0264 23                  INX     H
0265 22DC04              SHLD    BUFFP
```

```
0268 E1                     POP     H
0269 E67F                   ANI     7FH         ;STRIP PARITY
026B FE1A                   CPI     EOF
026D C0                     RNZ
026E F1                     POP     PSW         ;RAISE STACK
                    ;
026F 3AD504                 LDA     FULL        ;CHECK FLAG
0272 B7                     ORA     A           ;ZERO?
0273 CA8E02                 JZ      FINIS       ;YES, DONE
0276 CD7C02                 CALL    RESET       ;POINTER
0279 C36401                 JMP     MAIN5       ;GET MORE
                    ;
                    ; RESET MEMORY POINTER
                    ;
027C E5             RESET:  PUSH    H
027D 210005                 LXI     H,BUFFER
0280 22DC04                 SHLD    BUFFP
0283 E1                     POP     H
0284 C9                     RET
                    ;
0285 110204         NONAME: LXI     D,MES1      ;POINT TO MESSAGE
0288 CD3E02         FINI3:  CALL    PRINT
028B C3AE02                 JMP     ABOR2
                    ;
                    ; NORMAL END OF OF LISTING
                    ;
028E 32D404         FINIS:  STA     EOFFL       ;SET EOF FLAG
                    ;
0291 CD1B02                 CALL    FILL        ;OUT PAGE
                    ;
                    ; ADD AN EXTRA PAGE IF THERE IS AN ODD NUMBER
                    ; AND P FLAG IS NOT SET
                    ;
0294 3AD904                 LDA     PFLAG
0297 B7                     ORA     A           ;ZERO?
0298 C2AE02                 JNZ     ABOR2       ;NO
029B 3AD704                 LDA     PAGES       ;HOW MANY?
029E E601                   ANI     1           ;ODD?
02A0 CAAE02                 JZ      ABOR2       ;NO
                    ;
                    ; ADD BLANK PAGE TO MAKE EVEN
                    ;
                    ; (CAN WE CALL FILL?)
02A3 0642                   MVI     B,LMAX      ;LINES
02A5 3E0A           EPAGE:  MVI     A,LF
02A7 CD0402                 CALL    PCHAR
02AA 05                     DCR     B
02AB C2A502                 JNZ     EPAGE
                    ;
                    ABOR2:
                    ABOR3:
02AE 2AE004                 LHLD    OLDSP       ;OLD STACK POINTER
02B1 F9                     SPHL
02B2 C9                     RET
                    ;
                    ; SETUP FILE AND OPEN FOR INPUT
                    ;
```

```
02B3 115C00     SETUP:  LXI     D,FCB
02B6 0E0F               MVI     C,OPENF
02B8 CD0500             CALL    BDOS
                ;
                ; CHECK FOR ERRORS
                ;
02BB FEFF               CPI     255
02BD CAC502             JZ      BADOPN   ;NO GOOD
                ;
                ; OPEN IS OK
                ;
02C0 AF                 XRA     A
02C1 327C00             STA     FCBCR
02C4 C9                 RET
                ;
                ; BAD OPEN
                ;
02C5 215D00     BADOPN: LXI     H,FCBFN  ;1ST CHAR
02C8 7E                 MOV     A,M      ;GET IT
02C9 FE20               CPI     ' '      ;FILE NAME?
02CB CA8502             JZ      NONAME   ;NO
02CE 213F24             LXI     H,'?$'   ;SET UP FOR PRINT
02D1 226800             SHLD    FCBRL    ;USE INPUT FILENAME
02D4 115D00             LXI     D,FCBFN  ;FILENAME
02D7 C38802             JMP     FINI3    ;QUIT
                ;
                ; READ DISK FILE RECORD
                ;
02DA E5         DISKR:  PUSH    H
02DB D5                 PUSH    D
02DC C5                 PUSH    B
02DD 115C00             LXI     D,FCB
02E0 0E14               MVI     C,READF
02E2 CD0500             CALL    BDOS
02E5 C1                 POP     B
02E6 D1                 POP     D
02E7 E1                 POP     H
02E8 B7                 ORA     A        ;CHECK FOR ERRS
02E9 C8                 RZ               ;OK
                ; MAY BE EOF
02EA FE01               CPI     1
02EC CAF502             JZ      FEND     ;EOF
                ;
02EF 111104             LXI     D,MES2
02F2 C3AE02             JMP     ABOR3
                ;
                ; FOUND DISK EOF
                ;
02F5 2ADC04     FEND:   LHLD    BUFFP    ;GET POINTER
02F8 361A               MVI     M,EOF    ;PUT IN 1A
02FA C3A101             JMP     MAIN3
                ;
                ; TAB COUNTER ROUTINE
                ; JUMP HERE WITH BYTE IN B
                ;
```

```
02FD 78         TAB0:   MOV     A,B      ;GET BYTE
02FE FE20               CPI     ' '      ;CONTROL CHAR?
0300 DA1E03             JC      TABCR    ;YES
0303 CD0903             CALL    TABN     ;INCR COUNTER
0306 C30302             JMP     OUTT     ;SEND BYTE
                ;
                ; INCREMENT TAB COUNTER
                ; MAKE MODULO 8
                ;
0309 3AD204     TABN:   LDA     TABC     ;GET TAB COUNT
030C 3C                 INR     A        ;INCREMENT IT
030D E607               ANI     7        ;MODULO 8
030F 32D204             STA     TABC     ;SAVE IT
0312 C9                 RET
                ;
                ; READ THE BYTE AFTER ABORT
                ;
0313 CD1B02     DONE:   CALL    FILL     ;OUT PAGE
0316 0E01               MVI     C,CONS   ;READ CONSOLE
0318 CD0500             CALL    BDOS
031B C3AE02             JMP     ABOR3    ;RETURN
                ;
                ; IF CARRIAGE RETURN THEN ZERO COUNT
                ;
031E FE0D       TABCR:  CPI     CR
0320 C23103             JNZ     TABI     ;NOT CR
0323 CDEA01             CALL    ABORT
0326 0F                 RRC              ;KEY PRESSED?
0327 DA1303             JC      DONE     ;QUIT
032A AF                 XRA     A        ;GET A ZERO
032B 32D204             STA     TABC     ;SET TO ZERO
032E C30302             JMP     OUTT     ;SEND CR
                ;
                ; CHECK FOR TAB (CONTROL-I)
                ;
0331 FE09       TABI:   CPI     TAB
0333 C24003             JNZ     TFORM    ;NOT TAB
0336 CD5803     TAB2:   CALL    BLANK    ;SEND A BLANK
0339 CD0903             CALL    TABN     ;INCR TAB COUNT
033C C23603             JNZ     TAB2     ;MORE
033F C9                 RET              ;NO PRINT
                ;
                ; CHECK FOR FORMFEED
                ;
0340 FE0C       TFORM:  CPI     FORMFD
0342 C20302             JNZ     OUTT     ;NO
0345 3AD804             LDA     FFLAG    ;FORM FEED OPTION?
0348 B7                 ORA     A
0349 C25403             JNZ     TFOR2    ;OTHER CONTROL
034C 060A               MVI     B,LF
034E CD0302             CALL    OUTT
0351 C31B02             JMP     FILL     ;NORMAL FORMFEED
0354 F1         TFOR2:  POP     PSW      ;RESTORE STACK
0355 C3B901             JMP     GLOOP    ;NEXT BYTE
                ;
                ; SEND A BLANK
                ;
```

```
0358 3E20      BLANK:  MVI     A,' '
035A C30402            JMP     PCHAR    ;SEND IT
               ;
035D DBC4      CLOCK:  IN      ADATA    ;BOARD PRESENT?
035F FEFF              CPI     0FFH
0361 C8                RZ               ;NO
0362 3AD304            LDA     TIME2    ;PASS?
0365 B7                ORA     A        ;ZERO?
0366 C8                RZ               ;NOT 1ST
0367 AF                XRA     A
0368 32D304            STA     TIME2    ;SET 1ST
036B 21DE03            LXI     H,MON
036E CDA703            CALL    DATE     ;GET IT
0371 21ED03            LXI     H,HOUR
0374 CDC003            CALL    TIME
0377 21D703            LXI     H,PDATE  ;POINT DATE/TIME
037A CD8A03            CALL    SEND
037D AF                XRA     A        ;GET A ZERO
037E 326900            STA     FCB+13   ;NAME END
0381 215D00            LXI     H,FCBFN  ;NAME START
0384 CD8A03            CALL    SEND     ;SHOW
0387 21FF03            LXI     H,SCRLF
               ;
               ; SEND MESSAGE TO LIST
               ;
038A 7E        SEND:   MOV     A,M      ;GET BYTE
038B B7                ORA     A        ;ZERO AT END
038C C8                RZ               ;DONE
038D CD0402            CALL    PCHAR    ;SEND CHARACTER
0390 23                INX     H        ;INCREMENT POINTER
0391 C38A03            JMP     SEND
               ;
               ; READ A DIGIT
               ;
0394 7A        RDIGIT: MOV     A,D      ;SELECT DIGIT
0395 D3C4              OUT     ADATA
0397 DBC4              IN      ADATA    ;RESET INTERRUPT
0399 DBC5      DWAIT:  IN      ACONT    ;DIGIT PRESENT?
039B E680              ANI     80H
039D CA9903            JZ      DWAIT    ;LOOP UNTIL READY
03A0 DBC4              IN      ADATA    ;READ A DIGIT
03A2 E60F              ANI     0FH      ;MASK
03A4 F630              ORI     30H      ;CONVERT TO ASCII
03A6 C9                RET
               ;
               ;READ-DATE ROUTINE
               ;
03A7 AF        DATE:   XRA     A        ;DATE DISPLAY MODE
03A8 D3C6              OUT     BDATA
03AA 4F                MOV     C,A      ;THIS IS DATE
               ;
               ; READ FOUR DIGITS
               ;
03AB 1600      READ4:  MVI     D,0      ;SELECT FIRST DIGIT
03AD CD9403    RD4:    CALL    RDIGIT   ;DELAY ONE DIGIT SCAN
03B0 CDCD03            CALL    RSDIG    ;READ & STORE DIGIT
03B3 7A                MOV     A,D
03B4 FE20              CPI     20H      ;TWO DIGITS DONE?
```

```
03B6 C2BA03             JNZ     SKIP      ;SKIP A PLACE
03B9 23                 INX     H         ;SKIP : OR /
03BA FE40       SKIP:   CPI     40H
03BC C2AD03             JNZ     RD4       ;GET ANOTHER DIGIT
03BF C9                 RET
                ;
                ; READ TIME & READ AND STORE DIGIT
                ;
03C0 3E40       TIME:   MVI     A,40H     ;TIME DISPLAY MODE
03C2 D3C6               OUT     BDATA
03C4 0E01               MVI     C,1       ;THIS IS TIME
03C6 CDAB03             CALL    READ4     ;GET 4 DIGITS
03C9 23                 INX     H         ;SKIP COLON
03CA CDCD03             CALL    RSDIG
03CD CD9403     RSDIG:  CALL    RDIGIT
03D0 77                 MOV     M,A       ;STORE BYTE
03D1 23                 INX     H         ;INCR POINTER
03D2 7A                 MOV     A,D
03D3 C610               ADI     10H
03D5 57                 MOV     D,A
03D6 C9                 RET
                ;
                ;STORAGE AREA
                ;
                PDATE:
03D7 0D0A446174         DB      CR,LF,'Date '
03DE 78782F     MON:    DB      'xx/'     ;MONTH
03E1 78782F             DB      'xx/'     ;DAY
03E4 3830               DB      '80'      ;YEAR
03E6 202054696D         DB      '  Time '
03ED 78783A     HOUR:   DB      'xx:'     ;HOURS
03F0 78783A             DB      'xx:'     ;MINUTES
03F3 7878               DB      'xx'      ;SECONDS
03F5 2020204669         DB      '   File: ',0
03FF 0D0A00     SCRLF:  DB      CR,LF,0
0402 0D0A4E6F20MES1:    DB      CR,LF,'No file name$'
0411 0D0A446973MES2:    DB      CR,LF,'Disk error$'
041E 0D0A       RULES:  DB      CR,LF
0420 50726F6772         DB      'Program to list ASCII files'
043B 0D0A204F70         DB      CR,LF,' Options:'
0446 202863686F         DB      ' (choose one)',CR,LF
0455 2020462061         DB      '  F adds form feeds',CR,LF
046A 2020502073         DB      '  P skips extra even page'
0483 2061742065         DB      ' at end',CR,LF
048C 2020446563         DB      '  Decimal number skips '
04A3 6C696E6573         DB      'lines at beginning',CR,LF,LF
04B8 2050726573         DB      ' Press any key to abort.'
04D0 0A24               DB      LF,'$'
04D2 00         TABC:   DB      0         ;TAB COUNTER
04D3 80         TIME2:  DB      80H       ;PASS
04D4 FF         EOFFL:  DB      0FFH      ;EOF FLAG
04D5 00         FULL:   DB      0         ;FULL FLAG
04D6 02         LCOUNT: DB      2         ;LINE COUNT
04D7 00         PAGES:  DB      0         ;PAGE COUNT
04D8 00         FFLAG:  DB      0         ;FORMFEED FLAG
04D9 00         PFLAG:  DB      0         ;EXTRA-PAGE FLAG
04DA 0000       SKIPB:  DW      0         ;LINES TO SKIP
04DC 0005       BUFFP:  DW      BUFFER    ;POINTER
04DE            IBP:    DS      2         ;INPUT BUFF POINTER
```

```
                  ;
                  ;      STACK AREA
04E0              OLDSP:  DS      2       ;OLD STACK
04E2                      DS      30      ;STACK SPACE
                  STACK:                  ;NEW STACK
0500              BUFFER: DS      1       ;MEMORY BUFFER
                  ;
0501                      END
```

LIST can be executed by the command

```
A>LIST <filename> <optional argument>
```

The CP/M system copies LIST into memory at 100 hex and branches to it. The first step is to save the incoming stack pointer and set up a new one. The clock routine is executed next. The routine shown in Listing 10.2 is written for a Computime R clock board, but the program can be altered to accommodate other methods of timekeeping. The first step of the clock routine is to see if there is a clock board in the computer. One of the clock ports, at address ADATA, is read. If there is no port at this I/O address, the computer will read a value of FF hex. This will harmlessly terminate the subroutine with a return instruction.

If a clock board is present, then the first line of the printer output will appear in the form

```
Date 06/15/80 Time 13:18:33   FILE: <filename>
```

The next step is to see if one of the three optional arguments was given after the filename. These arguments are:

F	(add form feeds)
P	(no extra page at end)
decimal number	(skip lines at beginning)

The letter F is used to insert form feeds every 58 lines of the listing. The command looks like this.

```
A>LIST SORT.PAS  F
```

This feature is useful for listing BASIC, FORTRAN, Pascal, or assembly-language source programs. If this argument is selected, the proper number of line feeds is generated as the end of the page is approached. This step causes the printer to skip over the fold in the paper. Also, any form feeds that are encountered are ignored.

If the letter P is given for the argument, then no extra blank page is generated at the end. This argument can be used to save paper when several one-page files are printed.

Another possible argument to LIST is a decimal number. In this case, the argument tells how many lines of the file are to be skipped. For example, the command

```
A>LIST LINFIT.FOR 500
```

will skip the first 500 lines and start the listing with line 501. Use this option if you want to print the last part of a long file but don't want to wait for the first part to be printed.

At this point, the disk directory is searched for the requested filename. This filename is retrieved from the file-control block at address 5C hex. If the requested filename can't be found in the directory, then the filename is printed on the console along with a question mark. LIST is then aborted. If no filename was entered, an error message stating this fact appears on the console. LIST is aborted in this case also. LIST could have been programmed to handle input errors the way that the GO routine does. Then, instead of aborting, LIST would ask for the filename to be entered again. The addition of this feature is left as an exercise for the reader.

If the filename exists in the directory, then the requested disk file is read. The proper number of lines are skipped over if the second argument of the command line was a valid decimal number. At this point, the disk file is read into memory. The FDOS address located at address 7 is checked to see how much memory is available. The entire file will be copied into memory if there is room. A check is made to see if the disk end-of-file character is encountered before the available memory is filled.

In either case, the date and time of day are printed first. Then the data in memory are printed. The first character is ignored if it is an ASCII form feed. This will prevent a page eject immediately after the date and time. The ASCII tab character is properly expanded by the routine TABO. The number of lines is counted. When a form-feed character is encountered, the proper number of line feeds is issued to fill out the page. If the F argument was given, the form feeds will be automatically issued.

LIST can be aborted at any time during the printing by pressing any console key. The console is checked for this after each carriage return. The CP/M program DUMP is set up a little differently. The disk data are read into a 128-byte buffer, rather than into the memory area above 100 hex. Using a small buffer has the advantage that programs can be stored in memory during the DUMP operation and they will not be erased. But, in return, there is a lot of disk activity. The disk must be accessed every 128 bytes.

COPY A DISK FILE INTO MEMORY

If you have programmed one of the tape routines given in Chapter 8, then you can make backup copies of your disk files. But you have to first copy the disk file into memory. The copy step can be performed with the debugger DDT or SID.

```
A>SID <filename>
```

This command loads the debugger at address 100 hex and branches to it. The debugger relocates itself into high memory, then copies the requested disk file into memory starting at 100 hex. The address of the end of the disk file in memory is printed in hexadecimal. This address can be used in conjunction with the memory map given in Appendix B to determine the number of 256-byte blocks in the file. For example, if DDT gives a value of 13AF, then the disk file occupies 19 (decimal) blocks of 256 bytes.

When the file has been copied into memory, the G command in the debugger can be used to branch to the tape routine. Then the SAVE command can be given to make a copy on tape. The process can be reversed by loading the file into memory from tape. Branch to address zero to restart CP/M. Finally, give the SAVE command

```
A>SAVE XX <filename>
```

where XX is the decimal number of 256-byte blocks to be saved.

This procedure can be simplified by using the FETCH program given in Listing 10.3. FETCH uses CP/M for all I/O and disk operations, so it should work with all standard CP/M systems. There are, however, 2 items that need to be customized. Fetch is initially loaded into memory at 100 hex. It then automatically relocates itself to higher memory. The relocation address is chosen to be F400 hex in Listing 10.3. You may have to change it to some other address if this region is not available. The address is defined by the label ORIGIN in the source program. After FETCH copies the requested disk file into memory, it branches to your tape routine. This address, defined by the label MONIT, is chosen to be F000 hex in Listing 10.3 MONIT should be changed to your tape address.

Listing 10.3. Copy a disk file into memory.

```
                 ; (date goes here)
                 ;
                 TITLE   'Copy a disk file into memory.'
                 ;
                 ; THIS PROGRAM RELOCATES ITSELF TO HIGH MEMORY THEN
                 ; COPIES A DISK FILE TO MEMORY STARTING AT 100 HEX.
                 ; ASCII AND COM FILES ARE CORRECTLY HANDLED.
                 ; GIVE A THIRD ARGUMENT OF B FOR OTHER BINARY FILES.
                 ; A 1A EOF CHARACTER IS PUT AT THE END ASCII FILES.
                 ; THE LAST ADDRESS AND DECIMAL BLOCK SIZE ARE GIVEN.
                 ; FINALLY A JUMP IS MADE TO THE ADDRESS OF MONIT.
                 ;
                 ; QUITS IF NO READ-WRITE MEMORY AT NEW LOCATION
                 ;
F000 =           MONIT   EQU     0F000H  ;GO HERE WHEN DONE
F400 =           ORIGIN  EQU     0F400H  ;RELOCATE FETCH HERE
                 ;
0100 =           BUFFER  EQU     100H    ;START MEMORY BUFFER
0005 =           BDOS    EQU     5       ;DOS ENTRY POINT
```

```
0002 =          TYPEF   EQU     2          ;CONSOLE OUTPUT
0009 =          PBUF    EQU     9          ;PRINT CONSOLE BUFFER
000F =          OPENF   EQU     15         ;FILE OPEN
0014 =          READF   EQU     20         ;READ FUNCTION
000D =          CR      EQU     0DH        ;CARRIAGE RETURN
000A =          LF      EQU     0AH        ;LINE FEED
001A =          EOF     EQU     1AH        ;END OF FILE
                ;
005C =          FCB     EQU     5CH        ;FILE CONTROL BLOCK
0080 =          BUFF    EQU     80H        ;INPUT DISK BUFFER ADDR
                ;
                ;
                ; FILE CONTROL BLOCK DEFINITIONS
                ;
005D =          FCBFN   EQU     FCB+1      ;FILE NAME
0065 =          FCBFT   EQU     FCB+9      ;FILE TYPE
0068 =          FCBRL   EQU     FCB+12     ;CURRENT REEL #
007C =          FCBCR   EQU     FCB+32     ;NEXT REC # (0-127)
                ;
0100            ORG     100H
                ;
                ; SAVE OLD STACK AND SET UP NEW STACK
                ;
0100 210000     BEGIN:  LXI     H,0        ;ZERO HL
0103 39                 DAD     SP         ;ADD IN STACK
0104 22AEF5             SHLD    OLDSP      ;SAVE STACK
0107 31C8F5             LXI     SP,STACK ;DEFINE NEW STACK
010A 210001             LXI     H,BUFFER
010D 22A9F5             SHLD    BUFFP      ;MEMORY POINTER
                ;
                ; BLOCK MOVE REST OF PROGRAM
                ;
0110 112A01             LXI     D,OLDST ;OLD START
0113 2100F4             LXI     H,START ;NEW START
0116 01AC01             LXI     B,IBP-START ;LENGTH
0119 1A         LOOP:   LDAX    D          ;GET BYTE
011A 77                 MOV     M,A        ;MOVE TO NEW
011B BE                 CMP     M          ;CHECK MEMORY
011C C20000             JNZ     0          ;QUIT, BAD
011F 23                 INX     H          ;INCREMENT
0120 13                 INX     D          ; POINTERS
0121 0B                 DCX     B          ;DECR COUNT
0122 78                 MOV     A,B        ;SEE IF DONE
0123 B1                 ORA     C          ;ALL MOVED?
0124 C21901             JNZ     LOOP       ;NO
0127 C300F4             JMP     START      ;DONE
                ;
                ;ORIGINAL START OF PROGRAM
                ;
                OLDST:
                ;
F400            ORG     ORIGIN
                ;
F400 C346F4     START:  JMP     MAIN       ;FINAL START
F403 CD0BF4     FINIS:  CALL    CRLF
F406 2AAEF5             LHLD    OLDSP      ;ORIGINAL STACK
F409 F9                 SPHL
F40A C9                 RET
```

```
                    ;
F40B 3E0D           CRLF:   MVI     A,CR
F40D CD12F4                 CALL    PCHAR
F410 3E0A                   MVI     A,LF
                    ;
                    ; OUTPUT A CHARACTER FROM A
                    ;
F412 E5             PCHAR:  PUSH    H
F413 D5                     PUSH    D
F414 C5                     PUSH    B       ;SAVED
F415 0E02                   MVI     C,TYPEF
F417 5F                     MOV     E,A
F418 CD0500                 CALL    BDOS
F41B C1                     POP     B
F41C D1                     POP     D
F41D E1                     POP     H       ;RESTORED
F41E C9                     RET
                    ;
                    ; CARRIAGE RET, LINE FEED AND PRINT
                    ;
F41F CD0BF4         PRINTC: CALL    CRLF
                    ;
                    ; PRINT BUFFER UNTIL $ FOUND
                    ;
F422 E5             PRINT:  PUSH    H
F423 0E09                   MVI     C,PBUF
F425 CD0500                 CALL    BDOS
F428 E1                     POP     H
F429 C9                     RET
                    ;
                    ; GET NEXT BYTE FROM DISK BUFFER
                    ;
F42A 3AACF5         GNB:    LDA     IBP
F42D FE80                   CPI     80H
F42F C236F4                 JNZ     GO
                    ;
                    ; READ ANOTHER BUFFER
                    ;
F432 CD35F5                 CALL    DISKR
F435 AF                     XRA     A
                    ;
                    ; READ THE BYTE AT BUFF+REG A
                    ;
F436 5F             GO:     MOV     E,A
F437 1600                   MVI     D,0
F439 3C                     INR     A
F43A 32ACF5                 STA     IBP
                    ; POINTER IS INCREMENTED
                    ; SAVE THE CURRENT FILE ADDRESS
F43D E5                     PUSH    H
F43E 218000                 LXI     H,BUFF
F441 19                     DAD     D
F442 7E                     MOV     A,M
                    ; BYTE IS IN THE ACCUMULATOR
                    ; RESTORE FILE ADDRESS AND INCREMENT
F443 E1                     POP     H
F444 23                     INX     H
F445 C9                     RET
```

```
                        ;
                        MAIN:
                        ;
                        ; READ BYTES FROM DISK AND PUT IN MEMORY
                        ;
F446 CDD6F4                     CALL    SETUP   ;SET UP INPUT FILE
F449 3E80                       MVI     A,80H
F44B 32ACF5                     STA     IBP     ;BUFFER POINTER TO 80H
                        ;
F44E CD2AF4             MAIN2:  CALL    GNB     ;GET A BYTE
F451 E5                         PUSH    H
F452 2AA9F5                     LHLD    BUFFP   ;MEMORY POINTER
F455 77                         MOV     M,A     ;PUT BYTE IN
F456 23                         INX     H
F457 22A9F5                     SHLD    BUFFP   ;SAVE POINTER
F45A 47                         MOV     B,A
F45B 3EFF                       MVI     A,0FFH
F45D BD                         CMP     L       ;L=0?
F45E C26AF4                     JNZ     MAIN4   ;NO
F461 3A0700                     LDA     7       ;FDOS
F464 D60A                       SUI     10      ;CCP -1
F466 BC                         CMP     H       ;TOO BIG?
F467 DA2FF5                     JC      TOOBIG  ;WON'T FIT
                        ;
F46A E1                 MAIN4:  POP     H
F46B 3AABF5                     LDA     BFLAG   ;BINARY FILE?
F46E B7                         ORA     A
F46F C24EF4                     JNZ     MAIN2   ;YES
F472 78                         MOV     A,B     ;GET BYTE
F473 FE1A                       CPI     EOF
F475 C24EF4                     JNZ     MAIN2   ;NO
F478 2AA9F5                     LHLD    BUFFP   ;POINTER
F47B 2B                 MAIN5:  DCX     H
F47C 118AF5             DONE:   LXI     D,MESA
F47F CD22F4                     CALL    PRINT
F482 CDBDF4                     CALL    OUTHL   ;PRINT IT
F485 1199F5                     LXI     D,MESB
F488 CD22F4                     CALL    PRINT
                        ;
                        ; CONVERT FILE LENGTH TO DECIMAL K
                        ;
F48B 7C                         MOV     A,H
F48C 2600                       MVI     H,0     ;LEADING 0 FLAG
F48E 1664                       MVI     D,100
F490 0E2F               HM3:    MVI     C,'0'-1
F492 0C                 HM2:    INR     C
F493 92                         SUB     D       ;100/10
F494 D292F4                     JNC     HM2     ;STILL PLUS
F497 82                         ADD     D       ;ADD BACK
F498 47                         MOV     B,A     ;SAVE REMAINDER
F499 79                         MOV     A,C     ;GET 100S/TENS
                        ;
                        ; SUPPRESS LEADING ZEROS
                        ;
F49A FE31                       CPI     '1'     ;LESS THAN 1?
F49C D2A5F4                     JNC     HM4     ;NO
F49F 7C                         MOV     A,H     ;CHECK FLAG
F4A0 B7                         ORA     A       ;ZERO?
```

```
F4A1 79                 MOV     A,C        ;RESTORE BYTE
F4A2 CAAAF4             JZ      HM5        ;IF LEADING 0
F4A5 CD12F4     HM4:    CALL    PCHAR      ;1ST, 2ND DIGIT
F4A8 26FF               MVI     H,OFFH     ;SET FLAG
F4AA 7A         HM5:    MOV     A,D
F4AB D65A               SUI     90         ;100 TO 10
F4AD 57                 MOV     D,A
F4AE 78                 MOV     A,B
F4AF D290F4             JNC     HM3        ;ONCEMORE
F4B2 C630               ADI     '0'        ;ASCII BIAS
F4B4 CD12F4             CALL    PCHAR      ;3RD DIGIT
F4B7 CD0BF4             CALL    CRLF
F4BA C300F0             JMP     MONIT      ;DONE
                ;
                ; CONVERT BINARY IN H,L TO ASCII HEX
                ;
F4BD 4C         OUTHL:  MOV     C,H
F4BE CDC2F4             CALL    OUTHX
F4C1 4D                 MOV     C,L
F4C2 79         OUTHX:  MOV     A,C
F4C3 1F                 RAR
F4C4 1F                 RAR
F4C5 1F                 RAR
F4C6 1F                 RAR
F4C7 CDCBF4             CALL    HEX1
F4CA 79                 MOV     A,C
F4CB E60F       HEX1:   ANI     OFH        ;OUTPUT HEX BYTE
F4CD C690               ADI     90H
F4CF 27                 DAA                ;INTEL DAA TRICK
F4D0 CE40               ACI     40H
F4D2 27                 DAA
F4D3 C312F4             JMP     PCHAR
                ;
                ; SETUP FILE AND OPEN FOR INPUT
                ; CHECK FOR BINARY FILE
                ;
F4D6 AF         SETUP:  XRA     A          ;ZERO
F4D7 32ABF5             STA     BFLAG      ;NOT BINARY FILE
F4DA 2A6500             LHLD    FCBFT      ;FILE TYPE
F4DD 7D                 MOV     A,L        ;1ST CHAR
F4DE FE43               CPI     'C'
F4E0 C200F5             JNZ     BINCH      ;NO
F4E3 7C                 MOV     A,H        ;SECOND CHAR
F4E4 FE4F               CPI     'O'
F4E6 C200F5             JNZ     BINCH      ;NO
F4E9 3E1A       OPN2:   MVI     A,EOF      ;SET FLAG
F4EB 32ABF5             STA     BFLAG
                ;
                ; SETUP FILE AND OPEN FOR INPUT
                ;
F4EE 115C00     OPN3:   LXI     D,FCB
F4F1 0E0F               MVI     C,OPENF
F4F3 CD0500             CALL    BDOS
                ;
                ; CHECK FOR ERRORS
                ;
F4F6 FEFF               CPI     255
F4F8 CA0EF5             JZ      BADOPN     ;NO GOOD
```

```
                        ;
                        ; OPEN IS OK
                        ;
F4FB AF                         XRA     A
F4FC 327C00                     STA     FCBCR
F4FF C9                         RET
                        ;
                        ; B FOR THIRD ARGUMENT MEANS BINARY FILE
                        ;
F500 216D00             BINCH:  LXI     H,6DH
F503 7E                         MOV     A,M     ;GET THIRD ARGUMENT
F504 E657                       ANI     57H     ;LOWER TO UPPER
F506 FE42                       CPI     'B'
F508 C2EEF4                     JNZ     OPN3    ;NOT BINARY
F50B C3E9F4                     JMP     OPN2    ;BINARY
                        ;
                        ; BAD OPEN
                        ;
F50E 215D00             BADOPN: LXI     H,FCBFN ;1ST CHAR
F511 7E                         MOV     A,M     ;GET IT
F512 FE20                       CPI     ' '     ;FILE NAME?
F514 CA26F5                     JZ      NONAME  ;NO
F517 213F24                     LXI     H,'?$'  ;SET UP FOR PRINT
F51A 226800                     SHLD    FCBRL   ;USE INPUT FILENAME
F51D 115D00                     LXI     D,FCBFN ;FILENAME
F520 CD22F4                     CALL    PRINT
F523 C303F4                     JMP     FINIS   ;QUIT
                        ;
F526 1162F5             NONAME: LXI     D,MES1  ;POINT TO MESSAGE
F529 CD1FF4             NONAM2: CALL    PRINTC
F52C C303F4                     JMP     FINIS
                        ;
F52F 117AF5             TOOBIG: LXI     D,MES4
F532 C329F5                     JMP     NONAM2
                        ;
                        ; READ DISK FILE RECORD
                        ;
F535 E5                 DISKR:  PUSH    H
F536 D5                         PUSH    D
F537 C5                         PUSH    B
F538 115C00                     LXI     D,FCB
F53B 0E14                       MVI     C,READF
F53D CD0500                     CALL    BDOS
F540 C1                         POP     B
F541 D1                         POP     D
F542 E1                         POP     H
F543 B7                         ORA     A       ;CHECK FOR ERRS
F544 C8                         RZ              ;OK
                        ;
                        ; MAY BE EOF
                        ;
F545 FE01                       CPI     1
F547 CA53F5                     JZ      FEND    ;EOF
                        ;
                        ; INCORRECT FILE NAME
                        ;
F54A 116FF5                     LXI     D,MES2
F54D CD1FF4                     CALL    PRINTC
F550 C303F4                     JMP     FINIS
```

```
                        ;
                        ; FOUND DISK EOF
                        ;
F553 2AA9F5     FEND:      LHLD      BUFFP     ;GET POINTER
F556 3AABF5                LDA       BFLAG     ;COM FILE?
F559 B7                    ORA       A
F55A C27BF4                JNZ       MAIN5     ;YES
F55D 361A                  MVI       M,EOF     ;PUT EOF
F55F C37CF4                JMP       DONE
                        ;
                        ;STORAGE AREA
                        ;
F562 4E6F206669MES1:       DB        'No file name$'
F56F 4469736B20MES2:       DB        'Disk error$'
F57A 46696C6520MES4:       DB        'File is too big$'
F58A 4C61737420MESA:       DB        'Last address: $'
F599 206865782CMESB:       DB        ' hex,  Blocks: $'
F5A9 0001       BUFFP:     DW        BUFFER    ;POINTER
                        ;
F5AB            BFLAG:     DS        1         ;BINARY FLAG
F5AC            IBP:       DS        2         ;INPUT BUFF POINTER
                        ;
                        ;          STACK AREA
                        ;
F5AE            OLDSP:     DS        2         ;OLD STACK
F5B0                       DS        24        ;STACK SPACE
                STACK:
                        ;
F5C8                       END
```

One of the advantages of FETCH is that it is considerably smaller than DDT or SID. In addition, the decimal number of blocks to be saved and the last address of the program are given. FETCH is executed just like the debugger.

```
A>FETCH <filename>.<extension>
```

The CP/M system loads FETCH at 100 hex and then branches to it. The instructions at the beginning of FETCH are used to relocate the rest of the program into a preselected memory area. A jump is then made to the relocated FETCH.

FETCH loads the selected disk file into memory starting at 100 hex. An ASCII file is copied up to the 1A end-of-file mark. Typical extension names include the following.

ASM	(assembly language)
PAS	(Pascal)
FOR	(FORTRAN)
BAS	(BASIC)
HEX	(hex-encoded binary)
TEX	(text formatter)
LIB	(library)
MAC	(assembly language)

Executable binary files will have a file extension of COM; the entire file will be copied in this case. If a nonexecutable binary file is loaded, an additional argument of B (for binary) must be given.

```
FETCH  SORT.REL  B
```

Examples of executable binary files are

REL	(relocatable)
INT	(intermediate)

Relocatable files generated by the Microsoft assembler, and the FORTRAN, COBOL, and BASIC compiler are of this type. Intermediate files are produced by CBASIC.

After the disk file is loaded, FETCH prints two numbers. One number is the memory address of the end of the file expressed in hex. The other number is the size of the file. The number of 256-byte blocks is printed, in decimal.

Type up the program given in the listing. Assemble it and load it into memory with the debugger

```
A>DDT  FETCH.HEX
```

The debugger will load the move routine, the first part of FETCH, into memory at 100 hex. The main part of the program, however, will be loaded at the address of ORIGIN, 0F400 hex in this case. Use the debugger to move the main part of FETCH back down to the beginning of the user area.

```
MF400  F5FF  12A
```

Now, return to the CP/M system with a control-C and save the combination of the move program and the main part of FETCH.

```
A>SAVE 2 FETCH.COM
```

FETCH is now ready for use.

Appendixes

Appendix A	The ASCII Character Set	253
Appendix B	A 64K Memory Map	255
Appendix C	The 8080 Instruction Set (Alphabetic)	258
Appendix D	The 8080 Instruction Set (Numeric)	261
Appendix E	The Z-80 Instruction Set (Alphabetic)	264
Appendix F	The Z-80 Instruction Set (Numeric)	272
Appendix G	Cross-Reference of 8080 and Z-80 Instructions	280
Appendix H	Details of the Z-80 and 8080 Instruction Set	283
Appendix I	Abbreviations and Acronyms	311
Appendix J	Undocumented Z-80 Instructions	313

APPENDIX A
The ASCII Character Set

The ASCII character set is listed in numerical order with the corresponding decimal, hexadecimal, and octal values. The control characters are indicated with a caret (^). For example, the horizontal tab (HT) is formed with a control-I.

	decim	hex	octal						
NUL	0	00	000	^@	Null	@	64	40	100
SOH	1	01	001	^A	Start of heading	A	65	41	101
STX	2	02	002	^B	Start of text	B	66	42	102
ETX	3	03	003	^C	End of text	C	67	43	103
EOT	4	04	004	^D	End of transmission	D	68	44	104
ENQ	5	05	005	^E	Enquiry	E	69	45	105
ACK	6	06	006	^F	Acknowledge	F	70	46	106
BEL	7	07	007	^G	Bell	G	71	47	107
BS	8	08	010	^H	Backspace	H	72	48	110
HT	9	09	011	^I	Horizontal tab	I	73	49	111
LF	10	0A	012	^J	Line feed	J	74	4A	112
VT	11	0B	013	^K	Vertical tab	K	75	4B	113
FF	12	0C	014	^L	Form feed	L	76	4C	114
CR	13	0D	015	^M	Carriage return	M	77	4D	115
SO	14	0E	016	^N	Shift out	N	78	4E	116
SI	15	0F	017	^O	Shift in	O	79	4F	117
DLE	16	10	020	^P	Data link escape	P	80	50	120
DC1	17	11	021	^Q	Device control 1	Q	81	51	121
DC2	18	12	022	^R	Device control 2	R	82	52	122
DC3	19	13	023	^S	Device control 3	S	83	53	123
DC4	20	14	024	^T	Device control 4	T	84	54	124
NAK	21	15	025	^U	Negative acknowledge	U	85	55	125
SYN	22	16	026	^V	Synchronous idle	V	86	56	126
ETB	23	17	027	^W	End transmission block	W	87	57	127
CAN	24	18	030	^X	Cancel	X	88	58	130
EM	25	19	031	^Y	End of medium	Y	89	59	131
SUB	26	1A	032	^Z	Substitute	Z	90	5A	132
ESC	27	1B	033	^[	Escape	[	91	5B	133
FS	28	1C	034	^\	File separator	\	92	5C	134
GS	29	1D	035	^]	Group separator	]	93	5D	135
RS	30	1E	036	^^	Record separator	^	94	5E	136
US	31	1F	037	^_	Unit separator	_	95	5F	137

SP	32	20	040	Space
!	33	21	041	
"	34	22	042	
#	35	23	043	
$	36	24	044	
%	37	25	045	
&	38	26	046	
'	39	27	047	
(	40	28	050	
)	41	29	051	
*	42	2A	052	
+	43	2B	053	
,	44	2C	054	
-	45	2D	055	
.	46	2E	056	
/	47	2F	057	
0	48	30	060	
1	49	31	061	
2	50	32	062	
3	51	33	063	
4	52	34	064	
5	53	35	065	
6	54	36	066	
7	55	37	067	
8	56	38	070	
9	57	39	071	
:	58	3A	072	
;	59	3B	073	
<	60	3C	074	
=	61	3D	075	
>	62	3E	076	
?	63	3F	077	

`	96	60	140	
a	97	61	141	
b	98	62	142	
c	99	63	143	
d	100	64	144	
e	101	65	145	
f	102	66	146	
g	103	67	147	
h	104	68	150	
i	105	69	151	
j	106	6A	152	
k	107	6B	153	
l	108	6C	154	
m	109	6D	155	
n	110	6E	156	
o	111	6F	157	
p	112	70	160	
q	113	71	161	
r	114	72	162	
s	115	73	163	
t	116	74	164	
u	117	75	165	
v	118	76	166	
w	119	77	167	
x	120	78	170	
y	121	79	171	
z	122	7A	172	
{	123	7B	173	
|	124	7C	174	
}	125	7D	175	
~	126	7E	176	
DEL	127	7F	177	Delete

APPENDIX B

A 64K Memory Map

The 8080 and Z-80 microprocessors can directly address 64K bytes of memory. The memory area is mapped out in the chart that follows. Each entry represents a 256-byte block. The high-order byte of the address is given in hex then in octal. For example, the first entry of the second column is:

```
20    040      32
```

This represents an address range of 2000 to 2FFF hex, or 040-000 to 040-777 octal. The third column gives the decimal number of 1K blocks. The fourth column is the decimal number of 256-byte blocks starting at the address 100 hex. As an example, suppose that a CP/M program runs from 100 hex to 3035 hex. The 30 hex entry in the table shows that the program contains 48 decimal blocks of 256-byte size. The program can be saved with the CP/M command:

```
A>SAVE 48  filename
```

As another example, if you have two, 16K memory boards starting at address zero, then your top of memory is located at address 7FFF hex.

Hex	Oct	K	B1	Hex	Oct	K	B1	Hex	Oct	K	B1	Hex	Oct	K	B1
00	000		0	20	040		32	40	100		64	60	140		96
01	001		1	21	041		33	41	101		65	61	141		97
02	002		2	22	042		34	42	102		66	62	142		98
03	003	1	3	23	043	9	35	43	103	17	67	63	143	25	99
04	004		4	24	044		36	44	104		68	64	144		100
05	005		5	25	045		37	45	105		69	65	145		101
06	006		6	26	046		38	46	106		70	66	146		102
07	007	2	7	27	047	10	39	47	107	18	71	67	147	26	103
08	010		8	28	050		40	48	110		72	68	150		104
09	011		9	29	051		41	49	111		73	69	151		105
0A	012		10	2A	052		42	4A	112		74	6A	152		106
0B	013	3	11	2B	053	11	43	4B	113	19	75	6B	153	27	107
0C	014		12	2C	054		44	4C	114		76	6C	154		108
0D	015		13	2D	055		45	4D	115		77	6D	155		109
0E	016		14	2E	056		46	4E	116		78	6E	156		110
0F	017	4	15	2F	057	12	47	4F	117	20	79	6F	157	28	111
10	020		16	30	060		48	50	120		80	70	160		112
11	021		17	31	061		49	51	121		81	71	161		113
12	022		18	32	062		50	52	122		82	72	162		114
13	023	5	19	33	063	13	51	53	123	21	83	73	163	29	115
14	024		20	34	064		52	54	124		84	74	164		116
15	025		21	35	065		53	55	125		85	75	165		117
16	026		22	36	066		54	56	126		86	76	166		118
17	027	6	23	37	067	14	55	57	127	22	87	77	167	30	119
18	030		24	38	070		56	58	130		88	78	170		120
19	031		25	39	071		57	59	131		89	79	171		121
1A	032		26	3A	072		58	5A	132		90	7A	172		122
1B	033	7	27	3B	073	15	59	5B	133	23	91	7B	173	31	123
1C	034		28	3C	074		60	5C	134		92	7C	174		124
1D	035		29	3D	075		61	5D	135		93	7D	175		125
1E	036		30	3E	076		62	5E	136		94	7E	176		126
1F	037	8	31	3F	077	16	63	5F	137	24	95	7F	177	32	127

Hex	Oct	K	Bl	Hex	Oct	K	Bl	Hex	Oct	K	Bl	Hex	Oct	K	Bl
80	200		128	A0	240		160	C0	300		192	E0	340		224
81	201		129	A1	241		161	C1	301		193	E1	341		225
82	202		130	A2	242		162	C2	302		194	E2	342		226
83	203	33	131	A3	243	41	163	C3	303	49	195	E3	343	57	227
84	204		132	A4	244		164	C4	304		196	E4	344		228
85	205		133	A5	245		165	C5	305		197	E5	345		229
86	206		134	A6	246		166	C6	306		198	E6	346		230
87	207	34	135	A7	247	42	167	C7	307	50	199	E7	347	58	231
88	210		136	A8	250		168	C8	310		200	E8	350		232
89	211		137	A9	251		169	C9	311		201	E9	151		233
8A	212		138	AA	252		170	CA	312		202	EA	352		234
8B	213	35	139	AB	253	43	171	CB	313	51	203	EB	353	59	235
8C	214		140	AC	254		172	CC	314		204	EC	354		236
8D	215		141	AD	255		173	CD	315		205	ED	355		237
8E	216		142	AE	256		174	CE	316		206	EE	356		238
8F	217	36	143	AF	257	44	175	CF	317	52	207	EF	357	60	239
90	220		144	B0	260		176	D0	320		208	F0	360		240
91	221		145	B1	261		177	D1	321		209	F1	361		241
92	222		146	B2	262		178	D2	322		210	F2	362		242
93	223	37	147	B3	263	45	179	D3	323	53	211	F3	363	61	243
94	224		148	B4	264		180	D4	324		212	F4	364		244
95	225		149	B5	265		181	D5	325		213	F5	365		245
96	226		150	B6	266		182	D6	326		214	F6	366		246
97	227	38	151	B7	267	46	183	D7	327	54	215	F7	367	62	247
98	230		152	B8	270		184	D8	330		216	F8	370		248
99	231		153	B9	271		185	D9	331		217	F9	371		249
9A	232		154	BA	272		186	DA	332		218	FA	372		250
9B	233	39	155	BB	273	47	187	DB	333	55	219	FB	373	63	251
9C	234		156	BC	274		188	DC	334		220	FC	374		252
9D	235		157	BD	275		189	DD	335		221	FD	375		253
9E	236		158	BE	276		190	DE	336		222	FE	376		254
9F	237	40	159	BF	227	48	191	DF	337	56	223	FF	377	64	255

APPENDIX C

The 8080 Instruction Set (Alphabetic)

The 8080 instruction set is listed alphabetically with the corresponding hexadecimal code. The following representations apply.

nn 8-bit argument
nnnn 16-bit argument

Hex		Mnemonic		Hex		Mnemonic	
CE	nn	ACI	nn	2F		CMA	
8F		ADC	A	3F		CMC	
88		ADC	B	BF		CMP	A
89		ADC	C	B8		CMP	B
8A		ADC	D	B9		CMP	C
8B		ADC	E	BA		CMP	D
8C		ADC	H	BB		CMP	E
8D		ADC	L	BC		CMP	H
8E		ADC	M	BD		CMP	L
87		ADD	A	BE		CMP	M
80		ADD	B	D4	nnnn	CNC	nnnn
81		ADD	C	C4	nnnn	CNZ	nnnn
82		ADD	D	F4	nnnn	CP	nnnn
83		ADD	E	EC	nnnn	CPE	nnnn
84		ADD	H	FE	nn	CPI	nn
85		ADD	L	E4	nnnn	CPO	nnnn
86		ADD	M	CC	nnnn	CZ	nnnn
C6	nn	ADI	nn	27		DAA	
A7		ANA	A	09		DAD	B
A0		ANA	B	19		DAD	D
A1		ANA	C	29		DAD	H
A2		ANA	D	39		DAD	SP
A3		ANA	E	3D		DCR	A
A4		ANA	H	05		DCR	B
A5		ANA	L	0D		DCR	C
A6		ANA	M	15		DCR	D
E6	nn	ANI	nn	1D		DCR	E
CD	nnnn	CALL	nnnn	25		DCR	H
DC	nnnn	CC	nnnn	2D		DCR	L
FC	nnnn	CM	nnnn	35		DCR	M

Hex		Mnemonic		Hex	Mnemonic	
0B		DCX	B	43	MOV	B,E
1B		DCX	D	44	MOV	B,H
2B		DCX	H	45	MOV	B,L
3B		DCX	SP	46	MOV	B,M
F3		DI		4F	MOV	C,A
FB		EI		48	MOV	C,B
76		HLT		49	MOV	C,C
DB	nn	IN	nn	4A	MOV	C,D
3C		INR	A	4B	MOV	C,E
04		INR	B	4C	MOV	C,H
0C		INR	C	4D	MOV	C,L
14		INR	D	4E	MOV	C,M
1C		INR	E	57	MOV	D,A
24		INR	H	50	MOV	D,B
2C		INR	L	51	MOV	D,C
34		INR	M	52	MOV	D,D
03		INX	B	53	MOV	D,E
13		INX	D	54	MOV	D,H
23		INX	H	55	MOV	D,L
33		INX	SP	56	MOV	D,M
DA	nnnn	JC	nnnn	5F	MOV	E,A
FA	nnnn	JM	nnnn	58	MOV	E,B
C3	nnnn	JMP	nnnn	59	MOV	E,C
D2	nnnn	JNC	nnnn	5A	MOV	E,D
C2	nnnn	JNZ	nnnn	5B	MOV	E,E
F2	nnnn	JP	nnnn	5C	MOV	E,H
EA	nnnn	JPE	nnnn	5D	MOV	E,L
E2	nnnn	JPO	nnnn	5E	MOV	E,M
CA	nnnn	JZ	nnnn	67	MOV	H,A
3A	nnnn	LDA	nnnn	60	MOV	H,B
0A		LDAX	B	61	MOV	H,C
1A		LDAX	D	62	MOV	H,D
2A	nnnn	LHLD	nnnn	63	MOV	H,E
01	nnnn	LXI	B, nnnn	64	MOV	H,H
11	nnnn	LXI	D, nnnn	65	MOV	H,L
21	nnnn	LXI	H, nnnn	66	MOV	H,M
31	nnnn	LXI	SP, nnnn	6F	MOV	L,A
7F		MOV	A,A	68	MOV	L,B
78		MOV	A,B	69	MOV	L,C
79		MOV	A,C	6A	MOV	L,D
7A		MOV	A,D	6B	MOV	L,E
7B		MOV	A,E	6C	MOV	L,H
7C		MOV	A,H	6D	MOV	L,L
7D		MOV	A,L	6E	MOV	L,M
7E		MOV	A,M	77	MOV	M,A
47		MOV	B,A	70	MOV	M,B
40		MOV	B,B	71	MOV	M,C
41		MOV	B,C	72	MOV	M,D
42		MOV	B,D	73	MOV	M,E

Hex		Mnemonic		Hex		Mnemonic	
74		MOV	M,H	CF		RST	1
75		MOV	M,L	D7		RST	2
3E	nn	MVI	A, nn	DF		RST	3
06	nn	MVI	B, nn	E7		RST	4
0E	nn	MVI	C, nn	EF		RST	5
16	nn	MVI	D, nn	F7		RST	6
1E	nn	MVI	E, nn	FF		RST	7
26	nn	MVI	H, nn	C8		RZ	
2E	nn	MVI	L, nn	9F		SBB	A
36	nn	MVI	M, nn	98		SBB	B
00		NOP		99		SBB	C
B7		ORA	A	9A		SBB	D
B0		ORA	B	9B		SBB	E
B1		ORA	C	9C		SBB	H
B2		ORA	D	9D		SBB	L
B3		ORA	E	9E		SBB	M
B4		ORA	H	DE	nn	SBI	nn
B5		ORA	L	22	nnnn	SHLD	nnnn
B6		ORA	M	F9		SPHL	
F6	nn	ORI	nn	32	nnnn	STA	nnnn
D3	nn	OUT	nn	37		STC	
E9		PCHL		02		STAX	B
C1		POP	B	12		STAX	D
D1		POP	D	97		SUB	A
E1		POP	H	90		SUB	B
F1		POP	PSW	91		SUB	C
C5		PUSH	B	92		SUB	D
D5		PUSH	D	93		SUB	E
E5		PUSH	H	94		SUB	H
F5		PUSH	PSW	95		SUB	L
17		RAL		96		SUB	M
1F		RAR		D6	nn	SUI	nn
D8		RC		EB		XCHG	
C9		RET		AF		XRA	A
07		RLC		A8		XRA	B
F8		RM		A9		XRA	C
D0		RNC		AA		XRA	D
C0		RNZ		AB		XRA	E
F0		RP		AC		XRA	H
E8		RPE		AD		XRA	L
E0		RPO		AE		XRA	M
0F		RRC		EE	nn	XRI	nn
C7		RST	0	E3		XTHL	

APPENDIX D

The 8080 Instruction Set (Numeric)

The 8080 instruction set is listed alphabetically with the corresponding hexadecimal code. The following representations apply.

nn 8-bit argument
nnnn 16-bit argument

Hex		Mnemonic	
00		NOP	
01	nnnn	LXI	B, nnnn
02		STAX	B
03		INX	B
04		INR	B
05		DCR	B
06	nn	MVI	B, nn
07		RLC	
08		(not used)	
09		DAD	B
0A		LDAX	B
0B		DCX	B
0C		INR	C
0D		DCR	C
0E	nn	MVI	C, nn
0F		RRC	
10		(not used)	
11	nnnn	LXI	D, nnnn
12		STAX	D
13		INX	D
14		INR	D
15		DCR	D
16	nn	MVI	D, nn
17		RAL	
18		(not used)	
19		DAD	D
1A		LDAX	D
1B		DCX	D
1C		INR	E
1D		DCR	E

Hex		Mnemonic	
1E	nn	MVI	E, nn
1F		RAR	
20		(not used)	
21	nnnn	LXI	H, nnnn
22	nnnn	SHLD	nnnn
23		INX	H
24		INR	H
25		DCR	H
26	nn	MVI	H, nn
27		DAA	
28		(not used)	
29		DAD	H
2A	nnnn	LHLD	nnnn
2B		DCX	H
2C		INR	L
2D		DCR	L
2E	nn	MVI	L, nn
2F		CMA	
30		(not used)	
31	nnnn	LXI	SP, nnnn
32	nnnn	STA	nnnn
33		INX	SP
34		INR	M
35		DCR	M
36	nn	MVI	M, nn
37		STC	
38		(not used)	
39		DAD	SP
3A	nnnn	LDA	nnnn
3B		DCX	SP

Hex		Mnemonic		Hex	Mnemonic	
3C		INR	A	6D	MOV	L,L
3D		DCR	A	6E	MOV	L,M
3E	nn	MVI	A, nn	6F	MOV	L,A
3F		CMC		70	MOV	M,B
40		MOV	B,B	71	MOV	M,C
41		MOV	B,C	72	MOV	M,D
42		MOV	B,D	73	MOV	M,E
43		MOV	B,E	74	MOV	M,H
44		MOV	B,H	75	MOV	M,L
45		MOV	B,L	76	HLT	
46		MOV	B,M	77	MOV	M,A
47		MOV	B,A	78	MOV	A,B
48		MOV	C,B	79	MOV	A,C
49		MOV	C,C	7A	MOV	A,D
4A		MOV	C,D	7B	MOV	A,E
4B		MOV	C,E	7C	MOV	A,H
4C		MOV	C,H	7D	MOV	A,L
4D		MOV	C,L	7E	MOV	A,M
4E		MOV	C,M	7F	MOV	A,A
4F		MOV	C,A	80	ADD	B
50		MOV	D,B	81	ADD	C
51		MOV	D,C	82	ADD	D
52		MOV	D,D	83	ADD	E
53		MOV	D,E	84	ADD	H
54		MOV	D,H	85	ADD	L
55		MOV	D,L	86	ADD	M
56		MOV	D,M	87	ADD	A
57		MOV	D,A	88	ADC	B
58		MOV	E,B	89	ADC	C
59		MOV	E,C	8A	ADC	D
5A		MOV	E,D	8B	ADC	E
5B		MOV	E,E	8C	ADC	H
5C		MOV	E,H	8D	ADC	L
5D		MOV	E,L	8E	ADC	M
5E		MOV	E,M	8F	ADC	A
5F		MOV	E,A	90	SUB	B
60		MOV	H,B	91	SUB	C
61		MOV	H,C	92	SUB	D
62		MOV	H,D	93	SUB	E
63		MOV	H,E	94	SUB	H
64		MOV	H,H	95	SUB	L
65		MOV	H,L	96	SUB	M
66		MOV	H,M	97	SUB	A
67		MOV	H,A	98	SBB	B
68		MOV	L,B	99	SBB	C
69		MOV	L,C	9A	SBB	D
6A		MOV	L,D	9B	SBB	E
6B		MOV	L,E	9C	SBB	H
6C		MOV	L,H	9D	SBB	L

Hex		Mnemonic	
9E		SBB	M
9F		SBB	A
A0		ANA	B
A1		ANA	C
A2		ANA	D
A3		ANA	E
A4		ANA	H
A5		ANA	L
A6		ANA	M
A7		ANA	A
A8		XRA	B
A9		XRA	C
AA		XRA	D
AB		XRA	E
AC		XRA	H
AD		XRA	L
AE		XRA	M
AF		XRA	A
B0		ORA	B
B1		ORA	C
B2		ORA	D
B3		ORA	E
B4		ORA	H
B5		ORA	L
B6		ORA	M
B7		ORA	A
B8		CMP	B
B9		CMP	C
BA		CMP	D
BB		CMP	E
BC		CMP	H
BD		CMP	L
BE		CMP	M
BF		CMP	A
C0		RNZ	
C1		POP	B
C2	nnnn	JNZ	nnnn
C3	nnnn	JMP	nnnn
C4	nnnn	CNZ	nnnn
C5		PUSH	B
C6	nn	ADI	nn
C7		RST	0
C8		RZ	
C9		RET	
CA	nnnn	JZ	nnnn
CB		(not used)	
CC	nnnn	CZ	nnnn
CD	nnnn	CALL	nnnn
CE	nn	ACI	nn

Hex		Mnemonic	
CF		RST	1
D0		RNC	
D1		POP	D
D2	nnnn	JNC	nnnn
D3	nn	OUT	nn
D4	nnnn	CNC	nnnn
D5		PUSH	D
D6	nn	SUI	nn
D7		RST	2
D8		RC	
D9		(not used)	
DA	nnnn	JC	nnnn
DB	nn	IN	nn
DC	nnnn	CC	nnnn
DD		(not used)	
DE	nn	SBI	nn
DF		RST	3
E0		RPO	
E1		POP	H
E2	nnnn	JPO	nnnn
E3		XTHL	
E4	nnnn	CPO	nnnn
E5		PUSH	H
E6	nn	ANI	nn
E7		RST	4
E8		RPE	
E9		PCHL	
EA	nnnn	JPE	nnnn
EB		XCHG	
EC	nnnn	CPE	nnnn
ED		(not used)	
EE	nn	XRI	nn
EF		RST	5
F0		RP	
F1		POP	PSW
F2	nnnn	JP	nnnn
F3		DI	
F4	nnnn	CP	nnnn
F5		PUSH	PSW
F6	nn	ORI	nn
F7		RST	6
F8		RM	
F9		SPHL	
FA	nnnn	JM	nnnn
FB		EI	
FC	nnnn	CM	nnnn
FD		(not used)	
FE	nn	CPI	nn
FF		RST	7

APPENDIX E

The Z-80 Instruction Set (Alphabetic)

The Zilog Z-80 instruction set is listed alphabetically with the corresponding hexadecimal values. The following representations apply.

nn	8-bit arguments
nnnn	16-bit arguments
dd	8-bit signed displacement
*	instructions common to the 8080

Hex		Mnemonic		Hex		Mnemonic	
8E	*	ADC	A,(HL)	19	*	ADD	HL,DE
DD 8Edd		ADC	A,(IX+dd)	29	*	ADD	HL,HL
FD 8Edd		ADC	A,(IY+dd)	39	*	ADD	HL,SP
8F	*	ADC	A,A	DD 09		ADD	IX,BC
88	*	ADC	A,B	DD 19		ADD	IX,DE
89	*	ADC	A,C	DD 29		ADD	IX,IX
8A	*	ADC	A,D	DD 39		ADD	IX,SP
8B	*	ADC	A,E	FD 09		ADD	IY,BC
8C	*	ADC	A,H	FD 19		ADD	IY,DE
8D	*	ADC	A,L	FD 29		ADD	IY,IY
CE nn	*	ADC	A, nn	FD 39		ADD	IY,SP
ED 4A		ADC	HL,BC	A6	*	AND	(HL)
ED 5A		ADC	HL,DE	DD A6dd		AND	(IX+dd)
ED 6A		ADC	HL,HL	FD A6dd		AND	(IY+dd)
ED 7A		ADC	HL,SP	A7	*	AND	A
86	*	ADD	A,(HL)	A0	*	AND	B
DD 86dd		ADD	A,(IX+dd)	A1	*	AND	C
FD 86dd		ADD	A,(IY+dd)	A2	*	AND	D
87	*	ADD	A,A	A3	*	AND	E
80	*	ADD	A,B	A4	*	AND	H
81	*	ADD	A,C	A5	*	AND	L
82	*	ADD	A,D	E6 nn	*	AND	nn
83	*	ADD	A,E	CB 46		BIT	0,(HL)
84	*	ADD	A,H	DD CBdd46		BIT	0,(IX+dd)
85	*	ADD	A,L	FD CBdd46		BIT	0,(IY+dd)
C6 nn	*	ADD	A, nn	CB 47		BIT	0,A
09	*	ADD	HL,BC	CB 40		BIT	0,B

Hex	Mnemonic		Hex		Mnemonic	
CB 41	BIT	0,C	CB 68		BIT	5,B
CB 42	BIT	0,D	CB 69		BIT	5,C
CB 43	BIT	0,E	CB 6A		BIT	5,D
CB 44	BIT	0,H	CB 6B		BIT	5,E
CB 45	BIT	0,L	CB 6C		BIT	5,H
CB 4E	BIT	1,(HL)	CB 6D		BIT	5,L
DD CBdd4E	BIT	1,(IX+dd)	CB 76		BIT	6,(HL)
FD CBdd4E	BIT	1,(IY+dd)	DD CBdd76		BIT	6,(IX+dd)
CB 4F	BIT	1,A	FD CBdd76		BIT	6,(IY+dd)
CB 48	BIT	1,B	CB 77		BIT	6,A
CB 49	BIT	1,C	CB 70		BIT	6,B
CB 4A	BIT	1,D	CB 71		BIT	6,C
CB 4B	BIT	1,E	CB 72		BIT	6,D
CB 4C	BIT	1,H	CB 73		BIT	6,E
CB 4D	BIT	1,L	CB 74		BIT	6,H
CB 56	BIT	2,(HL)	CB 75		BIT	6,L
DD CBdd56	BIT	2,(IX+dd)	CB 7E		BIT	7,(HL)
FD CBdd56	BIT	2,(IY+dd)	DD CBdd7E		BIT	7,(IX+dd)
CB 57	BIT	2,A	FD CBdd7E		BIT	7,(IY+dd)
CB 50	BIT	2,B	CB 7F		BIT	7,A
CB 51	BIT	2,C	CB 78		BIT	7,B
CB 52	BIT	2,D	CB 79		BIT	7,C
CB 53	BIT	2,E	CB 7A		BIT	7,D
CB 54	BIT	2,H	CB 7B		BIT	7,E
CB 55	BIT	2,L	CB 7C		BIT	7,H
CB 5E	BIT	3,(HL)	CB 7D		BIT	7,L
DD CBdd5E	BIT	3,(IX+dd)	DC nnnn	*	CALL	C, nnnn
FD CBdd5E	BIT	3,(IY+dd)	FC nnnn	*	CALL	M, nnnn
CB 5F	BIT	3,A	D4 nnnn	*	CALL	NC, nnnn
CB 58	BIT	3,B	CD nnnn	*	CALL	nnnn
CB 59	BIT	3,C	C4 nnnn	*	CALL	NZ, nnnn
CB 5A	BIT	3,D	F4 nnnn	*	CALL	P, nnnn
CB 5B	BIT	3,E	EC nnnn	*	CALL	PE, nnnn
CB 5C	BIT	3,H	E4 nnnn	*	CALL	PO, nnnn
CB 5D	BIT	3,L	CC nnnn	*	CALL	Z, nnnn
CB 66	BIT	4,(HL)	3F	*	CCF	
DD CBdd66	BIT	4,(IX+dd)	BE	*	CP	(HL)
FD CBdd66	BIT	4,(IY+dd)	DD BEdd		CP	(IX+dd)
CB 67	BIT	4,A	FD BEdd		CP	(IY+dd)
CB 60	BIT	4,B	BF	*	CP	A
CB 61	BIT	4,C	B8	*	CP	B
CB 62	BIT	4,D	B9	*	CP	C
CB 63	BIT	4,E	BA	*	CP	D
CB 64	BIT	4,H	BB	*	CP	E
CB 65	BIT	4,L	BC	*	CP	H
CB 6E	BIT	5,(HL)	BD	*	CP	L
DD CBdd6E	BIT	5,(IX+dd)	FE nn	*	CP	nn
FD CBdd6E	BIT	5,(IY+dd)	ED A9		CPD	
CB 6F	BIT	5,A	ED B9		CPDR	

Hex		Mnemonic		Hex		Mnemonic	
ED A1		CPI		13	*	INC	DE
ED B1		CPIR		1C	*	INC	E
2F	*	CPL		24	*	INC	H
27	*	DAA		23	*	INC	HL
35	*	DEC	(HL)	DD 23		INC	IX
DD 35dd		DEC	(IX+dd)	FD 23		INC	IY
FD 35dd		DEC	(IY+dd)	2C	*	INC	L
3D	*	DEC	A	33	*	INC	SP
05	*	DEC	B	ED AA		IND	
0B	*	DEC	BC	ED BA		INDR	
0D	*	DEC	C	ED A2		INI	
15	*	DEC	D	ED B2		INIR	
1B	*	DEC	DE	E9	*	JP	(HL)
1D	*	DEC	E	DD E9		JP	(IX)
25	*	DEC	H	FD E9		JP	(IY)
2B	*	DEC	HL	DA nnnn	*	JP	C, nnnn
DD 2B		DEC	IX	FA nnnn	*	JP	M, nnnn
FD 2B		DEC	IY	D2 nnnn	*	JP	NC, nnnn
2D	*	DEC	L	C3 nnnn	*	JP	nnnn
3B	*	DEC	SP	C2 nnnn	*	JP	NZ, nnnn
F3	*	DI		F2 nnnn	*	JP	P, nnnn
10 dd		DJNZ	dd	EA nnnn	*	JP	PE, nnnn
FB	*	EI		E2 nnnn	*	JP	PO, nnnn
E3	*	EX	(SP),HL	CA nnnn	*	JP	Z, nnnn
DD E3		EX	(SP),IX	38 dd		JR	C, dd
FD E3		EX	(SP),IY	18 dd		JR	dd
08		EX	AF,AF'	30 dd		JR	NC, dd
EB	*	EX	DE,HL	20 dd		JR	NZ, dd
D9		EXX		28 dd		JR	Z, dd
76	*	HALT		02	*	LD	(BC),A
ED 46		IM	0	12	*	LD	(DE),A
ED 56		IM	1	77	*	LD	(HL),A
ED 5E		IM	2	70	*	LD	(HL),B
ED 78		IN	A,(C)	71	*	LD	(HL),C
DB nn	*	IN	A,(nn)	72	*	LD	(HL),D
ED 40		IN	B,(C)	73	*	LD	(HL),E
ED 48		IN	C,(C)	74	*	LD	(HL),H
ED 50		IN	D,(C)	75	*	LD	(HL),L
ED 58		IN	E,(C)	36 nn	*	LD	(HL), nn
ED 60		IN	H,(C)	DD 77dd		LD	(IX+dd),A
ED 68		IN	L,(C)	DD 70dd		LD	(IX+dd),B
34	*	INC	(HL)	DD 71dd		LD	(IX+dd),C
DD 34dd		INC	(IX+dd)	DD 72dd		LD	(IX+dd),D
FD 34dd		INC	(IY+dd)	DD 73dd		LD	(IX+dd),E
3C	*	INC	A	DD 74dd		LD	(IX+dd),H
04	*	INC	B	DD 75dd		LD	(IX+dd),L
03	*	INC	BC	DD 36ddnn		LD	(IX+dd), nn
0C	*	INC	C	FD 77dd		LD	(IY+dd),A
14	*	INC	D	FD 70dd		LD	(IY+dd),B

Hex		Mnemonic		Hex		Mnemonic	
FD 71dd		LD	(IY+dd),C	4B	*	LD	C,E
FD 72dd		LD	(IY+dd),D	4C	*	LD	C,H
FD 73dd		LD	(IY+dd),E	4D	*	LD	C,L
FD 74dd		LD	(IY+dd),H	0E nn	*	LD	C, nn
FD 75dd		LD	(IY+dd),L	56	*	LD	D,(HL)
FD 36ddnn		LD	(IY+dd), nn	DD 56dd		LD	D,(IX+dd)
32 nnnn	*	LD	(nnnn),A	FD 56dd		LD	D,(IY+dd)
ED 43nnnn		LD	(nnnn),BC	57	*	LD	D,A
ED 53nnnn		LD	(nnnn),DE	50	*	LD	D,B
22 nnnn	*	LD	(nnnn),HL	51	*	LD	D,C
DD 22nnnn		LD	(nnnn),IX	52	*	LD	D,D
FD 22nnnn		LD	(nnnn),IY	53	*	LD	D,E
ED 73nnnn		LD	(nnnn),SP	54	*	LD	D,H
0A	*	LD	A,(BC)	55	*	LD	D,L
1A	*	LD	A,(DE)	16 nn	*	LD	D, nn
7E	*	LD	A,(HL)	ED 5Bnnnn		LD	DE, (nnnn)
DD 7Edd		LD	A,(IX+dd)	11 nnnn	*	LD	DE, nnnn
FD 7Edd		LD	A,(IY+dd)	5E	*	LD	E,(HL)
3A nnnn	*	LD	A, (nnnn)	DD 5Edd		LD	E,(IX+dd)
7F	*	LD	A,A	FD 5Edd		LD	E,(IY+dd)
78	*	LD	A,B	5F	*	LD	E,A
79	*	LD	A,C	58	*	LD	E,B
7A	*	LD	A,D	59	*	LD	E,C
7B	*	LD	A,E	5A	*	LD	E,D
7C	*	LD	A,H	5B	*	LD	E,E
ED 57		LD	A,I	5C	*	LD	E,H
7D	*	LD	A,L	5D	*	LD	E,L
3E nn	*	LD	A, nn	1E nn	*	LD	E, nn
ED 5F		LD	A,R	66	*	LD	H,(HL)
46	*	LD	B,(HL)	DD 66dd		LD	H,(IX+dd)
DD 46dd		LD	B,(IX+dd)	FD 66dd		LD	H,(IY+dd)
FD 46dd		LD	B,(IY+dd)	67	*	LD	H,A
47	*	LD	B,A	60	*	LD	H,B
40	*	LD	B,B	61	*	LD	H,C
41	*	LD	B,C	62	*	LD	H,D
42	*	LD	B,D	63	*	LD	H,E
43	*	LD	B,E	64	*	LD	H,H
44	*	LD	B,H	65	*	LD	H,L
45	*	LD	B,L	26 nn	*	LD	H, nn
06 nn	*	LD	B, nn	2A nnnn	*	LD	HL, (nnnn)
ED 4Bnnnn		LD	BC, (nnnn)	21 nnnn	*	LD	HL, nnnn
01 nnnn	*	LD	BC, nnnn	ED 47		LD	I,A
4E	*	LD	C,(HL)	DD 2Annnn		LD	IX, (nnnn)
DD 4Edd		LD	C,(IX+dd)	DD 21nnnn		LD	IX, nnnn
FD 4Edd		LD	C,(IY+dd)	FD 2Annnn		LD	IY, (nnnn)
4F	*	LD	C,A	FD 21nnnn		LD	IY, nnnn
48	*	LD	C,B	6E	*	LD	L,(HL)
49	*	LD	C,C	DD 6Edd		LD	L,(IX+dd)
4A	*	LD	C,D	FD 6Edd		LD	L,(IY+dd)

Hex		Mnemonic		Hex		Mnemonic	
6F	*	LD	L,A	F5	*	PUSH	AF
68	*	LD	L,B	C5	*	PUSH	BC
69	*	LD	L,C	D5	*	PUSH	DE
6A	*	LD	L,D	E5	*	PUSH	HL
6B	*	LD	L,E	DD E5		PUSH	IX
6C	*	LD	L,H	FD E5		PUSH	IY
6D	*	LD	L,L	CB 86		RES	0,(HL)
2E nn	*	LD	L, nn	DD CBdd86		RES	0,(IX+dd)
ED 4F		LD	R,A	FD CBdd86		RES	0,(IY+dd)
ED 7Bnnnn		LD	SP, (nnnn)	CB 87		RES	0,A
F9	*	LD	SP,HL	CB 80		RES	0,B
DD F9		LD	SP,IX	CB 81		RES	0,C
FD F9		LD	SP,IY	CB 82		RES	0,D
31 nnnn	*	LD	SP, nnnn	CB 83		RES	0,E
ED A8		LDD		CB 84		RES	0,H
ED B8		LDDR		CB 85		RES	0,L
ED A0		LDI		CB 8E		RES	1,(HL)
ED B0		LDIR		DD CBdd8E		RES	1,(IX+dd)
ED 44		NEG		FD CBdd8E		RES	1,(IY+dd)
00	*	NOP		CB 8F		RES	1,A
B6	*	OR	(HL)	CB 88		RES	1,B
DD B6dd		OR	(IX+dd)	CB 89		RES	1,C
FD B6dd		OR	(IY+dd)	CB 8A		RES	1,D
B7	*	OR	A	CB 8B		RES	1,E
B0	*	OR	B	CB 8C		RES	1,H
B1	*	OR	C	CB 8D		RES	1,L
B2	*	OR	D	CB 96		RES	2,(HL)
B3	*	OR	E	DD CBdd96		RES	2,(IX+dd)
B4	*	OR	H	FD CBdd96		RES	2,(IY+dd)
B5	*	OR	L	CB 97		RES	2,A
F6 nn	*	OR	nn	CB 90		RES	2,B
ED BB		OTDR		CB 91		RES	2,C
ED B3		OTIR		CB 92		RES	2,D
ED 79		OUT	(C),A	CB 93		RES	2,E
ED 41		OUT	(C),B	CB 94		RES	2,H
ED 49		OUT	(C),C	CB 95		RES	2,L
ED 51		OUT	(C),D	CB 9E		RES	3,(HL)
ED 59		OUT	(C),E	DD CBdd9E		RES	3,(IX+dd)
ED 61		OUT	(C),H	FD CBdd9E		RES	3,(IY+dd)
ED 69		OUT	(C),L	CB 9F		RES	3,A
D3 nn	*	OUT	(nn),A	CB 98		RES	3,B
ED AB		OUTD		CB 99		RES	3,C
ED A3		OUTI		CB 9A		RES	3,D
F1	*	POP	AF	CB 9B		RES	3,E
C1	*	POP	BC	CB 9C		RES	3,H
D1	*	POP	DE	CB 9D		RES	3,L
E1	*	POP	HL	CB A6		RES	4,(HL)
DD E1		POP	IX	DD CBddA6		RES	4,(IX+dd)
FD E1		POP	IY	FD CBddA6		RES	4,(IY+dd)

Hex		Mnemonic		Hex		Mnemonic	
CB A7		RES	4,A	DD CBdd16		RL	(IX+dd)
CB A0		RES	4,B	FD CBdd16		RL	(IY+dd)
CB A1		RES	4,C	CB 17		RL	A
CB A2		RES	4,D	CB 10		RL	B
CB A3		RES	4,E	CB 11		RL	C
CB A4		RES	4,H	CB 12		RL	D
CB A5		RES	4,L	CB 13		RL	E
CB AE		RES	5,(HL)	CB 14		RL	H
DD CBddAE		RES	5,(IX+dd)	CB 15		RL	L
FD CBddAE		RES	5,(IY+dd)	17	*	RLA	
CB AF		RES	5,A	CB 06		RLC	(HL)
CB A8		RES	5,B	DD CBdd06		RLC	(IX+dd)
CB A9		RES	5,C	FD CBdd06		RLC	(IY+dd)
CB AA		RES	5,D	CB 07		RLC	A
CB AB		RES	5,E	CB 00		RLC	B
CB AC		RES	5,H	CB 01		RLC	C
CB AD		RES	5,L	CB 02		RLC	D
CB B6		RES	6,(HL)	CB 03		RLC	E
DD CBddB6		RES	6,(IX+dd)	CB 04		RLC	H
FD CBddB6		RES	6,(IY+dd)	CB 05		RLC	L
CB B7		RES	6,A	07	*	RLCA	
CB B0		RES	6,B	ED 6F		RLD	
CB B1		RES	6,C	CB 1E		RR	(HL)
CB B2		RES	6,D	DD CBdd1E		RR	(IX+dd)
CB B3		RES	6,E	FD CBdd1E		RR	(IY+dd)
CB B4		RES	6,H	CB 1F		RR	A
CB B5		RES	6,L	CB 18		RR	B
CB BE		RES	7,(HL)	CB 19		RR	C
DD CBddBE		RES	7,(IX+dd)	CB 1A		RR	D
FD CBddBE		RES	7,(IY+dd)	CB 1B		RR	E
CB BF		RES	7,A	CB 1C		RR	H
CB B8		RES	7,B	CB 1D		RR	L
CB B9		RES	7,C	1F	*	RRA	
CB BA		RES	7,D	CB 0E		RRC	(HL)
CB BB		RES	7,E	DD CBdd0E		RRC	(IX+dd)
CB BC		RES	7,H	FD CBdd0E		RRC	(IY+dd)
CB BD		RES	7,L	CB 0F		RRC	A
C9	*	RET		CB 08		RRC	B
D8	*	RET	C	CB 09		RRC	C
F8	*	RET	M	CB 0A		RRC	D
D0	*	RET	NC	CB 0B		RRC	E
C0	*	RET	NZ	CB 0C		RRC	H
F0	*	RET	P	CB 0D		RRC	L
E8	*	RET	PE	0F	*	RRCA	
E0	*	RET	PO	ED 67		RRD	
C8	*	RET	Z	C7	*	RST	0
ED 4D		RETI		CF	*	RST	8
ED 45		RETN		D7	*	RST	10H
CB 16		RL	(HL)	DF	*	RST	18H

Hex		Mnemonic		Hex	Mnemonic	
E7	*	RST	20H	CB D5	SET	2,L
EF	*	RST	28H	CB DE	SET	3,(HL)
F7	*	RST	30H	DD CBddDE	SET	3,(IX+dd)
FF	*	RST	38H	FD CBddDE	SET	3,(IY+dd)
9E	*	SBC	A,(HL)	CB DF	SET	3,A
DD 9Edd		SBC	A,(IX+dd)	CB D8	SET	3,B
FD 9Edd		SBC	A,(IY+dd)	CB D9	SET	3,C
9F	*	SBC	A,A	CB DA	SET	3,D
98	*	SBC	A,B	CB DB	SET	3,E
99	*	SBC	A,C	CB DC	SET	3,H
9A	*	SBC	A,D	CB DD	SET	3,L
9B	*	SBC	A,E	CB E6	SET	4,(HL)
9C	*	SBC	A,H	DD CBddE6	SET	4,(IX+dd)
9D	*	SBC	A,L	FD CBddE6	SET	4,(IY+dd)
DE nn	*	SBC	A, nn	CB E7	SET	4,A
ED 42		SBC	HL,BC	CB E0	SET	4,B
ED 52		SBC	HL,DE	CB E1	SET	4,C
ED 62		SBC	HL,HL	CB E2	SET	4,D
ED 72		SBC	HL,SP	CB E3	SET	4,E
37	*	SCF		CB E4	SET	4,H
CB C6		SET	0,(HL)	CB E5	SET	4,L
DD CBddC6		SET	0,(IX+dd)	CB EE	SET	5,(HL)
FD CBddC6		SET	0,(IY+dd)	DD CBddEE	SET	5,(IX+dd)
CB C7		SET	0,A	FD CBddEE	SET	5,(IY+dd)
CB C0		SET	0,B	CB EF	SET	5,A
CB C1		SET	0,C	CB E8	SET	5,B
CB C2		SET	0,D	CB E9	SET	5,C
CB C3		SET	0,E	CB EA	SET	5,D
CB C4		SET	0,H	CB EB	SET	5,E
CB C5		SET	0,L	CB EC	SET	5,H
CB CE		SET	1,(HL)	CB ED	SET	5,L
DD CBddCE		SET	1,(IX+dd)	CB F6	SET	6,(HL)
FD CBddCE		SET	1,(IY+dd)	DD CBddF6	SET	6,(IX+dd)
CB CF		SET	1,A	FD CBddF6	SET	6,(IY+dd)
CB C8		SET	1,B	CB F7	SET	6,A
CB C9		SET	1,C	CB F0	SET	6,B
CB CA		SET	1,D	CB F1	SET	6,C
CB CB		SET	1,E	CB F2	SET	6,D
CB CC		SET	1,H	CB F3	SET	6,E
CB CD		SET	1,L	CB F4	SET	6,H
CB D6		SET	2,(HL)	CB F5	SET	6,L
DD CBddD6		SET	2,(IX+dd)	CB FE	SET	7,(HL)
FD CBddD6		SET	2,(IY+dd)	DD CBddFE	SET	7,(IX+dd)
CB D7		SET	2,A	FD CBddFE	SET	7,(IY+dd)
CB D0		SET	2,B	CB FF	SET	7,A
CB D1		SET	2,C	CB F8	SET	7,B
CB D2		SET	2,D	CB F9	SET	7,C
CB D3		SET	2,E	CB FA	SET	7,D
CB D4		SET	2,H	CB FB	SET	7,E

Hex	Mnemonic	
CB FC	SET	7,H
CB FD	SET	7,L
CB 26	SLA	(HL)
DD CBdd26	SLA	(IX+dd)
FD CBdd26	SLA	(IY+dd)
CB 27	SLA	A
CB 20	SLA	B
CB 21	SLA	C
CB 22	SLA	D
CB 23	SLA	E
CB 24	SLA	H
CB 25	SLA	L
CB 2E	SRA	(HL)
DD CBdd2E	SRA	(IX+dd)
FD CBdd2E	SRA	(IY+dd)
CB 2F	SRA	A
CB 28	SRA	B
CB 29	SRA	C
CB 2A	SRA	D
CB 2B	SRA	E
CB 2C	SRA	H
CB 2D	SRA	L
CB 3E	SRL	(HL)
DD CBdd3E	SRL	(IX+dd)
FD CBdd3E	SRL	(IY+dd)
CB 3F	SRL	A
CB 38	SRL	B

Hex		Mnemonic	
CB 39		SRL	C
CB 3A		SRL	D
CB 3B		SRL	E
CB 3C		SRL	H
CB 3D		SRL	L
96	*	SUB	(HL)
DD 96dd		SUB	(IX+dd)
FD 96dd		SUB	(IY+dd)
97	*	SUB	A
90	*	SUB	B
91	*	SUB	C
92	*	SUB	D
93	*	SUB	E
94	*	SUB	H
95	*	SUB	L
D6 nn	*	SUB	nn
AE	*	XOR	(HL)
DD AEdd		XOR	(IX+dd)
FD AEdd		XOR	(IY+dd)
AF	*	XOR	A
A8	*	XOR	B
A9	*	XOR	C
AA	*	XOR	D
AB	*	XOR	E
AC	*	XOR	H
AD	*	XOR	L
EE nn	*	XOR	nn

APPENDIX F

The Z-80 Instruction Set (Numeric)

The Zilog Z-80 instruction set is listed numerically with the corresponding hexadecimal values. The following representations apply.

nn	8-bit argument
nnnn	16-bit argument
dd	8-bit signed displacement
*	instructions common to the 8080

Hex			Mnemonic	
00		*	NOP	
01	nnnn	*	LD	BC, nnnn
02		*	LD	(BC),A
03		*	INC	BC
04		*	INC	B
05		*	DEC	B
06	nn	*	LD	B, nn
07		*	RLCA	
08			EX	AF,AF′
09		*	ADD	HL,BC
0A		*	LD	A,(BC)
0B		*	DEC	BC
0C		*	INC	C
0D		*	DEC	C
0E	nn	*	LD	C, nn
0F		*	RRCA	
10	dd		DJNZ	dd
11	nnnn	*	LD	DE, nnnn
12		*	LD	(DE),A
13		*	INC	DE
14		*	INC	D
15		*	DEC	D
16	nn	*	LD	D, nn
17		*	RLA	
18	dd		JR	dd
19		*	ADD	HL,DE
1A		*	LD	A,(DE)
1B		*	DEC	DE
1C		*	INC	E
1D		*	DEC	E
1E	nn	*	LD	E, nn
1F		*	RRA	
20	dd		JR	NZ, dd
21	nnnn	*	LD	HL, nnnn
22	nnnn	*	LD	(nnnn),HL
23		*	INC	HL
24		*	INC	H
25		*	DEC	H
26	nn	*	LD	H, nn
27		*	DAA	
28	dd		JR	Z, dd
29		*	ADD	HL,HL
2A	nnnn	*	LD	HL, (nnnn)
2B		*	DEC	HL
2C		*	INC	L
2D		*	DEC	L
2E	nn	*	LD	L, nn
2F		*	CPL	
30	dd		JR	NC, dd
31	nnnn	*	LD	SP, nnnn
32	nnnn	*	LD	(nnnn),A
33		*	INC	SP
34		*	INC	(HL)
35		*	DEC	(HL)
36	nn	*	LD	(HL), nn
37		*	SCF	

Hex			Mnemonic		Hex		Mnemonic	
38 dd		JR	C, dd		69	*	LD	L,C
39	*	ADD	HL,SP		6A	*	LD	L,D
3A nnnn	*	LD	A, (nnnn)		6B	*	LD	L,E
3B	*	DEC	SP		6C	*	LD	L,H
3C	*	INC	A		6D	*	LD	L,L
3D	*	DEC	A		6E	*	LD	L,(HL)
3E nn	*	LD	A, nn		6F	*	LD	L,A
3F	*	CCF			70	*	LD	(HL),B
40	*	LD	B,B		71	*	LD	(HL),C
41	*	LD	B,C		72	*	LD	(HL),D
42	*	LD	B,D		73	*	LD	(HL),E
43	*	LD	B,E		74	*	LD	(HL),H
44	*	LD	B,H		75	*	LD	(HL),L
45	*	LD	B,L		76	*	HALT	
46	*	LD	B,(HL)		77	*	LD	(HL),A
47	*	LD	B,A		78	*	LD	A,B
48	*	LD	C,B		79	*	LD	A,C
49	*	LD	C,C		7A	*	LD	A,D
4A	*	LD	C,D		7B	*	LD	A,E
4B	*	LD	C,E		7C	*	LD	A,H
4C	*	LD	C,H		7D	*	LD	A,L
4D	*	LD	C,L		7E	*	LD	A,(HL)
4E	*	LD	C,(HL)		7F	*	LD	A,A
4F	*	LD	C,A		80	*	ADD	A,B
50	*	LD	D,B		81	*	ADD	A,C
51	*	LD	D,C		82	*	ADD	A,D
52	*	LD	D,D		83	*	ADD	A,E
53	*	LD	D,E		84	*	ADD	A,H
54	*	LD	D,H		85	*	ADD	A,L
55	*	LD	D,L		86	*	ADD	A,(HL)
56	*	LD	D,(HL)		87	*	ADD	A,A
57	*	LD	D,A		88	*	ADC	A,B
58	*	LD	E,B		89	*	ADC	A,C
59	*	LD	E,C		8A	*	ADC	A,D
5A	*	LD	E,D		8B	*	ADC	A,E
5B	*	LD	E,E		8C	*	ADC	A,H
5C	*	LD	E,H		8D	*	ADC	A,L
5D	*	LD	E,L		8E	*	ADC	A,(HL)
5E	*	LD	E,(HL)		8F	*	ADC	A,A
5F	*	LD	E,A		90	*	SUB	B
60	*	LD	H,B		91	*	SUB	C
61	*	LD	H,C		92	*	SUB	D
62	*	LD	H,D		93	*	SUB	E
63	*	LD	H,E		94	*	SUB	H
64	*	LD	H,H		95	*	SUB	L
65	*	LD	H,L		96	*	SUB	(HL)
66	*	LD	H,(HL)		97	*	SUB	A
67	*	LD	H,A		98	*	SBC	A,B
68	*	LD	L,B		99	*	SBC	A,C

Hex	Mnemonic			Hex	Mnemonic	
9A	*	SBC	A,D	CB 00	RLC	B
9B	*	SBC	A,E	CB 01	RLC	C
9C	*	SBC	A,H	CB 02	RLC	D
9D	*	SBC	A,L	CB 03	RLC	E
9E	*	SBC	A,(HL)	CB 04	RLC	H
9F	*	SBC	A,A	CB 05	RLC	L
A0	*	AND	B	CB 06	RLC	(HL)
A1	*	AND	C	CB 07	RLC	A
A2	*	AND	D	CB 08	RRC	B
A3	*	AND	E	CB 09	RRC	C
A4	*	AND	H	CB 0A	RRC	D
A5	*	AND	L	CB 0B	RRC	E
A6	*	AND	(HL)	CB 0C	RRC	H
A7	*	AND	A	CB 0D	RRC	L
A8	*	XOR	B	CB 0E	RRC	(HL)
A9	*	XOR	C	CB 0F	RRC	A
AA	*	XOR	D	CB 10	RL	B
AB	*	XOR	E	CB 11	RL	C
AC	*	XOR	H	CB 12	RL	D
AD	*	XOR	L	CB 13	RL	E
AE	*	XOR	(HL)	CB 14	RL	H
AF	*	XOR	A	CB 15	RL	L
B0	*	OR	B	CB 16	RL	(HL)
B1	*	OR	C	CB 17	RL	A
B2	*	OR	D	CB 18	RR	B
B3	*	OR	E	CB 19	RR	C
B4	*	OR	H	CB 1A	RR	D
B5	*	OR	L	CB 1B	RR	E
B6	*	OR	(HL)	CB 1C	RR	H
B7	*	OR	A	CB 1D	RR	L
B8	*	CP	B	CB 1E	RR	(HL)
B9	*	CP	C	CB 1F	RR	A
BA	*	CP	D	CB 20	SLA	B
BB	*	CP	E	CB 21	SLA	C
BC	*	CP	H	CB 22	SLA	D
BD	*	CP	L	CB 23	SLA	E
BE	*	CP	(HL)	CB 24	SLA	H
BF	*	CP	A	CB 25	SLA	L
C0	*	RET	NZ	CB 26	SLA	(HL)
C1	*	POP	BC	CB 27	SLA	A
C2 nnnn	*	JP	NZ, nnnn	CB 28	SRA	B
C3 nnnn	*	JP	nnnn	CB 29	SRA	C
C4 nnnn	*	CALL	NZ, nnnn	CB 2A	SRA	D
C5	*	PUSH	BC	CB 2B	SRA	E
C6 nn	*	ADD	A, nn	CB 2C	SRA	H
C7	*	RST	0	CB 2D	SRA	L
C8	*	RET	Z	CB 2E	SRA	(HL)
C9	*	RET		CB 2F	SRA	A
CA nnnn	*	JP	Z, nnnn	CB 38	SRL	B

Hex	Mnemonic		Hex	Mnemonic	
CB 39	SRL	C	CB 6A	BIT	5,D
CB 3A	SRL	D	CB 6B	BIT	5,E
CB 3B	SRL	E	CB 6C	BIT	5,H
CB 3C	SRL	H	CB 6D	BIT	5,L
CB 3D	SRL	L	CB 6E	BIT	5,(HL)
CB 3E	SRL	(HL)	CB 6F	BIT	5,A
CB 3F	SRL	A	CB 70	BIT	6,B
CB 40	BIT	0,B	CB 71	BIT	6,C
CB 41	BIT	0,C	CB 72	BIT	6,D
CB 42	BIT	0,D	CB 73	BIT	6,E
CB 43	BIT	0,E	CB 74	BIT	6,H
CB 44	BIT	0,H	CB 75	BIT	6,L
CB 45	BIT	0L	CB 76	BIT	6,(HL)
CB 46	BIT	0,(HL)	CB 77	BIT	6,A
CB 47	BIT	0,A	CB 78	BIT	7,B
CB 48	BIT	1,B	CB 79	BIT	7,C
CB 49	BIT	1,C	CB 7A	BIT	7,D
CB 4A	BIT	1,D	CB 7B	BIT	7,E
CB 4B	BIT	1,E	CB 7C	BIT	7,H
CB 4C	BIT	1,H	CB 7D	BIT	7,L
CB 4D	BIT	1,L	CB 7E	BIT	7,(HL)
CB 4E	BIT	1,(HL)	CB 7F	BIT	7,A
CB 4F	BIT	1,A	CB 80	RES	0,B
CB 50	BIT	2,B	CB 81	RES	0,C
CB 51	BIT	2,C	CB 82	RES	0,D
CB 52	BIT	2,D	CB 83	RES	0,E
CB 53	BIT	2,E	CB 84	RES	0,H
CB 54	BIT	2,H	CB 85	RES	0,L
CB 55	BIT	2,L	CB 86	RES	0,(HL)
CB 56	BIT	2,(HL)	CB 87	RES	0,A
CB 57	BIT	2,A	CB 88	RES	1,B
CB 58	BIT	3,B	CB 89	RES	1,C
CB 59	BIT	3,C	CB 8A	RES	1,D
CB 5A	BIT	3,D	CB 8B	RES	1,E
CB 5B	BIT	3,E	CB 8C	RES	1,H
CB 5C	BIT	3,H	CB 8D	RES	1,L
CB 5D	BIT	3,L	CB 8E	RES	1,(HL)
CB 5E	BIT	3,(HL)	CB 8F	RES	1,A
CB 5F	BIT	3,A	CB 90	RES	2,B
CB 60	BIT	4,B	CB 91	RES	2,C
CB 61	BIT	4,C	CB 92	RES	2,D
CB 62	BIT	4,D	CB 93	RES	2,E
CB 63	BIT	4,E	CB 94	RES	2,H
CB 64	BIT	4,H	CB 95	RES	2,L
CB 65	BIT	4,L	CB 96	RES	2,(HL)
CB 66	BIT	4,(HL)	CB 97	RES	2,A
CB 67	BIT	4,A	CB 98	RES	3,B
CB 68	BIT	5,B	CB 99	RES	3,C
CB 69	BIT	5,C	CB 9A	RES	3,D

Hex	Mnemonic		Hex	Mnemonic	
CB 9B	RES	3,E	CB CC	SET	1,H
CB 9C	RES	3,H	CB CD	SET	1,L
CB 9D	RES	3,L	CB CE	SET	1,(HL)
CB 9E	RES	3,(HL)	CB CF	SET	1,A
CB 9F	RES	3,A	CB D0	SET	2,B
CB A0	RES	4,B	CB D1	SET	2,C
CB A1	RES	4,C	CB D2	SET	2,D
CB A2	RES	4,D	CB D3	SET	2,E
CB A3	RES	4,E	CB D4	SET	2,H
CB A4	RES	4,H	CB D5	SET	2,L
CB A5	RES	4,L	CB D6	SET	2,(HL)
CB A6	RES	4,(HL)	CB D7	SET	2,A
CB A7	RES	4,A	CB D8	SET	3,B
CB A8	RES	5,B	CB D9	SET	3,C
CB A9	RES	5,C	CB DA	SET	3,D
CB AA	RES	5,D	CB DB	SET	3,E
CB AB	RES	5,E	CB DC	SET	3,H
CB AC	RES	5,H	CB DD	SET	3,L
CB AD	RES	5,L	CB DE	SET	3,(HL)
CB AE	RES	5,(HL)	CB DF	SET	3,A
CB AF	RES	5,A	CB E0	SET	4,B
CB B0	RES	6,B	CB E1	SET	4,C
CB B1	RES	6,C	CB E2	SET	4,D
CB B2	RES	6,D	CB E3	SET	4,E
CB B3	RES	6,E	CB E4	SET	4,H
CB B4	RES	6,H	CB E5	SET	4,L
CB B5	RES	6,L	CB E6	SET	4,(HL)
CB B6	RES	6,(HL)	CB E7	SET	4,A
CB B7	RES	6,A	CB E8	SET	5,B
CB B8	RES	7,B	CB E9	SET	5,C
CB B9	RES	7,C	CB EA	SET	5,D
CB BA	RES	7,D	CB EB	SET	5,E
CB BB	RES	7,E	CB EC	SET	5,H
CB BC	RES	7,H	CB ED	SET	5,L
CB BD	RES	7,L	CB EE	SET	5,(HL)
CB BE	RES	7,(HL)	CB EF	SET	5,A
CB BF	RES	7,A	CB F0	SET	6,B
CB C0	SET	0,B	CB F1	SET	6,C
CB C1	SET	0,C	CB F2	SET	6,D
CB C2	SET	0,D	CB F3	SET	6,E
CB C3	SET	0,E	CB F4	SET	6,H
CB C4	SET	0,H	CB F5	SET	6,L
CB C5	SET	0,L	CB F6	SET	6,(HL)
CB C6	SET	0,(HL)	CB F7	SET	6,A
CB C7	SET	0,A	CB F8	SET	7,B
CB C8	SET	1,B	CB F9	SET	7,C
CB C9	SET	1,C	CB FA	SET	7,D
CB CA	SET	1,D	CB FB	SET	7,E
CB CB	SET	1,E	CB FC	SET	7,H

Hex		Mnemonic		Hex		Mnemonic	
CB FD		SET	7,L	DD 9Edd		SBC	A,(IX+dd)
CB FE		SET	7,(HL)	DD A6dd		AND	(IX+dd)
CB FF		SET	7,A	DD AEdd		XOR	(IX+dd)
CC nnnn	*	CALL	Z, nnnn	DD B6dd		OR	(IX+dd)
CD nnnn	*	CALL	nnnn	DD BEdd		CP	(IX+dd)
CE nn	*	ADC	A, nn	DD CBdd06		RLC	(IX+dd)
CF	*	RST	8	DD CBdd0E		RRC	(IX+dd)
D0	*	RET	NC	DD CBdd16		RL	(IX+dd)
D1	*	POP	DE	DD CBdd1E		RR	(IX+dd)
D2 nnnn	*	JP	NC, nnnn	DD CBdd26		SLA	(IX+dd)
D3 nn	*	OUT	(nn),A	DD CBdd2E		SRA	(IX+dd)
D4 nnnn	*	CALL	NC, nnnn	DD CBdd3E		SRL	(IX+dd)
D5	*	PUSH	DE	DD CBdd46		BIT	0,(IX+dd)
D6 nn	*	SUB	nn	DD CBdd4E		BIT	1,(IX+dd)
D7	*	RST	10H	DD CBdd56		BIT	2,(IX+dd)
D8	*	RET	C	DD CBdd5E		BIT	3,(IX+dd)
D9		EXX		DD CBdd66		BIT	4,(IX+dd)
DA nnnn	*	JP	C, nnnn	DD CBdd6E		BIT	5,(IX+dd)
DB nn	*	IN	A, (nn)	DD CBdd76		BIT	6,(IX+dd)
DC nnnn	*	CALL	C, nnnn	DD CBdd7E		BIT	7,(IX+dd)
DD 09		ADD	IX,BC	DD CBdd86		RES	0,(IX+dd)
DD 19		ADD	IX,DE	DD CBdd8E		RES	1,(IX+dd)
DD 21nnnn		LD	IX, nnnn	DD CBdd96		RES	2,(IX+dd)
DD 22nnnn		LD	(nnnn),IX	DD CBdd9E		RES	3,(IX+dd)
DD 23		INC	IX	DD CBddA6		RES	4,(IX+dd)
DD 29		ADD	IX,IX	DD CBddAE		RES	5,(IX+dd)
DD 2Annnn		LD	IX, (nnnn)	DD CBddB6		RES	6,(IX+dd)
DD 2B		DEC	IX	DD CBddBE		RES	7,(IX+dd)
DD 34dd		INC	(IX+dd)	DD CBddC6		SET	0,(IX+dd)
DD 35dd		DEC	(IX+dd)	DD CBddCE		SET	1,(IX+dd)
DD 36ddnn		LD	(IX+dd), nn	DD CBddD6		SET	2,(IX+dd)
DD 39		ADD	IX,SP	DD CBddDE		SET	3,(IX+dd)
DD 46dd		LD	B,(IX+dd)	DD CBddE6		SET	4,(IX+dd)
DD 4Edd		LD	C,(IX+dd)	DD CBddEE		SET	5,(IX+dd)
DD 56dd		LD	D,(IX+dd)	DD CBddF6		SET	6,(IX+dd)
DD 5Edd		LD	E,(IX+dd)	DD CBddFE		SET	7,(IX+dd)
DD 66dd		LD	H,(IX+dd)	DD E1		POP	IX
DD 6Edd		LD	L,(IX+dd)	DD E3		EX	(SP),IX
DD 70dd		LD	(IX+dd),B	DD E5		PUSH	IX
DD 71dd		LD	(IX+dd),C	DD E9		JP	(IX)
DD 72dd		LD	(IX+dd),D	DD F9		LD	SP,IX
DD 73dd		LD	(IX+dd),E	DE nn	*	SBC	A, nn
DD 74dd		LD	(IX+dd),H	DF	*	RST	18H
DD 75dd		LD	(IX+dd),L	E0	*	RET	PO
DD 77dd		LD	(IX+dd),A	E1	*	POP	HL
DD 7Edd		LD	A,(IX+dd)	E2 nnnn	*	JP	PO, nnnn
DD 86dd		ADD	A,(IX+dd)	E3	*	EX	(SP),HL
DD 8Edd		ADC	A,(IX+dd)	E4 nnnn	*	CALL	PO, nnnn
DD 96dd		SUB	(IX+dd)	E5	*	PUSH	HL

Hex		Mnemonic		Hex		Mnemonic	
E6 nn	*	AND	nn	ED A2		INI	
E7	*	RST	20H	ED A3		OUTI	
E8	*	RET	PE	ED A8		LDD	
E9	*	JP	(HL)	ED A9		CPD	
EA nnnn	*	JP	PE, nnnn	ED AA		IND	
EB	*	EX	DE,HL	ED AB		OUTD	
EC nnnn	*	CALL	PE, nnnn	ED B0		LDIR	
ED 40		IN	B,(C)	ED B1		CPIR	
ED 41		OUT	(C),B	ED B2		INIR	
ED 42		SBC	HL,BC	ED B3		OTIR	
ED 43nnnn		LD	(nnnn),BC	ED B8		LDDR	
ED 44		NEG		ED B9		CPDR	
ED 45		RETN		ED BA		INDR	
ED 46		IM	0	ED BB		OTDR	
ED 47		LD	I,A	EE nn	*	XOR	N
ED 48		IN	C,(C)	EF	*	RST	28H
ED 49		OUT	(C),C	F0	*	RET	P
ED 4A		ADC	HL,BC	F1	*	POP	AF
ED 4Bnnnn		LD	BC, (nnnn)	F2 nnnn	*	JP	P, nnnn
ED 4D		RETI		F3	*	DI	
ED 4F		LD	R,A	F4 nnnn	*	CALL	P, nnnn
ED 50		IN	D,(C)	F5	*	PUSH	AF
ED 51		OUT	(C),D	F6 nn	*	OR	nn
ED 52		SBC	HL,DE	F7	*	RST	30H
ED 53nnnn		LD	(nnnn),DE	F8	*	RET	M
ED 56		IM	1	F9	*	LD	SP,HL
ED 57		LD	A,I	FA nnnn	*	JP	M, nnnn
ED 58		IN	E,(C)	FB	*	EI	
ED 59		OUT	(C),E	FC nnnn	*	CALL	M, nnnn
ED 5A		ADC	HL,DE	FD 09		ADD	IY,BC
ED 5Bnnnn		LD	DE, (nnnn)	FD 19		ADD	IY,DE
ED 5E		IM	2	FD 21nnnn		LD	IY, nnnn
ED 5F		LD	A,R	FD 22nnnn		LD	(nnnn),IY
ED 60		IN	H,(C)	FD 23		INC	IY
ED 61		OUT	(C),H	FD 29		ADD	IY,IY
ED 62		SBC	HL,HL	FD 2Annnn		LD	IY, (nnnn)
ED 67		RRD		FD 2B		DEC	IY
ED 68		IN	L,(C)	FD 34dd		INC	(IY+dd)
ED 69		OUT	(C),L	FD 35dd		DEC	(IY+dd)
ED 6A		ADC	HL,HL	FD 36ddnn		LD	(IY+dd), nn
ED 6F		RLD		FD 39		ADD	IY,SP
ED 72		SBC	HL,SP	FD 46dd		LD	B,(IY+dd)
ED 73nnnn		LD	(nnnn),SP	FD 4Edd		LD	C,(IY+dd)
ED 78		IN	A,(C)	FD 56dd		LD	D,(IY+dd)
ED 79		OUT	(C),A	FD 5Edd		LD	E,(IY+dd)
ED 7A		ADC	HL,SP	FD 66dd		LD	H,(IY+dd)
ED 7Bnnnn		LD	SP, (nnnn)	FD 6Edd		LD	L,(IY+dd)
ED A0		LDI		FD 70dd		LD	(IY+dd),B
ED A1		CPI		FD 71dd		LD	(IY+dd),C

Hex	Mnemonic		Hex		Mnemonic	
FD 72dd	LD	(IY+dd),D	FD CBdd6E		BIT	5,(IY+dd)
FD 73dd	LD	(IY+dd),E	FD CBdd76		BIT	6,(IY+dd)
FD 74dd	LD	(IY+dd),H	FD CBdd7E		BIT	7,(IY+dd)
FD 75dd	LD	(IY+dd),L	FD CBdd86		RES	0,(IY+dd)
FD 77dd	LD	(IY+dd),A	FD CBdd8E		RES	1,(IY+dd)
FD 7Edd	LD	A,(IY+dd)	FD CBdd96		RES	2,(IY+dd)
FD 86dd	ADD	A,(IY+dd)	FD CBdd9E		RES	3,(IY+dd)
FD 8Edd	ADC	A,(IY+dd)	FD CBddA6		RES	4,(IY+dd)
FD 96dd	SUB	(IY+dd)	FD CBddAE		RES	5,(IY+dd)
FD 9Edd	SBC	A,(IY+dd)	FD CBddB6		RES	6,(IY+dd)
FD A6dd	AND	(IY+dd)	FD CBddBE		RES	7,(IY+dd)
FD AEdd	XOR	(IY+dd)	FD CBddC6		SET	0,(IY+dd)
FD B6dd	OR	(IY+dd)	FD CBddCE		SET	1,(IY+dd)
FD BEdd	CP	(IY+dd)	FD CBddD6		SET	2,(IY+dd)
FD CBdd06	RLC	(IY+dd)	FD CBddDE		SET	3,(IY+dd)
FD CBdd0E	RRC	(IY+dd)	FD CBddE6		SET	4,(IY+dd)
FD CBdd16	RL	(IY+dd)	FD CBddEE		SET	5,(IY+dd)
FD CBdd1E	RR	(IY+dd)	FD CBddF6		SET	6,(IY+dd)
FD CBdd26	SLA	(IY+dd)	FD CBddFE		SET	7,(IY+dd)
FD CBdd2E	SRA	(IY+dd)	FD E1		POP	IY
FD CBdd3E	SRL	(IY+dd)	FD E3		EX	(SP),IY
FD CBdd46	BIT	0,(IY+dd)	FD E5		PUSH	IY
FD CBdd4E	BIT	1,(IY+dd)	FD E9		JP	(IY)
FD CBdd56	BIT	2,(IY+dd)	FD F9		LD	SP,IY
FD CBdd5E	BIT	3,(IY+dd)	FE nn	*	CP	nn
FD CBdd66	BIT	4,(IY+dd)	FF	*	RST	38H

APPENDIX G

Cross-Reference of 8080 and Z-80 Instructions

The instructions are listed in alphabetic order according to the 8080 mnemonic. The following representations are used.

N	8-bit constant
NN	16-bit constant
R	single register
RR	double register
—	range of values expressed as one character
— —	range of values expressed as two characters

8080 code		HEX code	Z-80 code	
ACI	N	CE	ADC	A,N
ADC	M	8E	ADC	A,(HL)
ADC	R	8—	ADC	A,R
ADD	M	86	ADD	A,(HL)
ADD	R	8—	ADD	A,R
ADI	N	C6	ADD	A,N
ANA	M	A6	AND	(HL)
ANA	R	A—	AND	R
ANI	N	E6	AND	N
CALL	NN	CD	CALL	NN
CC	NN	DC	CALL	C,NN
CM	NN	FC	CALL	M,NN
CMA		2F	CPL	
CMC		3F	CCF	
CMP	M	BE	CP	(HL)
CMP	R	B—	CP	R
CNC	NN	D4	CALL	NC,NN
CNZ	NN	C4	CALL	NZ,NN
CP	NN	F4	CALL	P,NN
CPE	NN	EC	CALL	PE,NN
CPI	N	FE	CP	N
CPO	NN	E4	CALL	PO,NN
CZ	NN	CC	CALL	Z,NN

8080 code		HEX code	Z-80 code	
DAA		27	DAA	
DAD	B	09	ADD	HL,BC
DAD	D	19	ADD	HL,DE
DAD	H	29	ADD	HL,HL
DAD	SP	39	ADD	HL,SP
DCR	M	35	DEC	(HL)
DCR	R	— —	DEC	R
DCX	B	0B	DEC	BC
DCX	D	1B	DEC	DE
DCX	H	2B	DEC	HL
DCX	SP	3B	DEC	SP
DI		F3	DI	
EI		FB	EI	
HLT		76	HALT	
IN	N	DB	IN	A,(N)
INR	M	34	INC	(HL)
INR	R	— —	INC	R
INX	B	03	INC	BC
INX	D	13	INC	DE
INX	H	23	INC	HL
INX	SP	33	INC	SP
JC	NN	DA	JP	C,NN
JM	NN	FA	JP	M,NN
JMP	NN	C3	JP	NN
JNC	NN	D2	JP	NC,NN
JNZ	NN	C2	JP	NZ,NN
JP	NN	F2	JP	P,NN
JPE	NN	EA	JP	PE,NN
JPO	NN	E2	JP	PO,NN
JZ	NN	CA	JP	Z,NN
LDA	NN	3A	LD	A,(NN)
LDAX	B	0A	LD	A,(BC)
LDAX	D	26	LD	A,(DE)
LHLD	NN	2A	LD	HL,(NN)
LXI	B,NN	01	LD	BC,NN
LXI	D,NN	11	LD	DE,NN
LXI	H,NN	21	LD	HL,NN
LXI	SP,NN	31	LD	SP,NN
MOV	M,R	— —	LD	(HL),R
MOV	R,M	— —	LD	R,(HL)
MOV	R,R2	— —	LD	R,R2
MVI	M,N	36	LD	(HL),N
MVI	R,N	— —	LD	R,N
NOP		00	NOP	
ORA	M	B6	OR	(HL)
ORA	R	B—	OR	R
ORI	N	F6	OR	N
OUT	N	D3	OUT	(N),A

8080 code		HEX code	Z-80 code	
PCHL		E9	JP	(HL)
POP	B	C1	POP	BC
POP	D	D1	POP	DE
POP	H	E1	POP	HL
POP	PSW	F1	POP	AF
PUSH	B	C5	PUSH	BC
PUSH	D	D5	PUSH	DE
PUSH	H	E5	PUSH	HL
PUSH	PSW	F5	PUSH	AF
RAL		17	RLA	
RAR		1F	RRA	
RC		D8	RET	C
RET		C9	RET	
RLC		07	RLCA	
RM		F8	RET	M
RNC		D0	RET	NC
RNZ		C0	RET	NZ
RP		F0	RET	P
RPE		E8	RET	PE
RPO		E0	RET	PO
RRC		15	RRCA	
RST	0	C7	RST	0
RST	1	CF	RST	8
RST	2	D7	RST	10H
RST	3	DF	RST	18H
RST	4	E7	RST	20H
RST	5	EF	RST	28H
RST	6	F7	RST	30H
RST	7	FF	RST	38H
RZ		C8	RET	Z
SBB	M	9E	SBC	A,(HL)
SBB	R	9—	SBC	A,R
SBI	N	DE	SBC	A,N
SHLD	NN	22	LD	(NN),HL
SPHL		F9	LD	SP,HL
STA	NN	32	LD	(NN),A
STAX	B	02	LD	(BC),A
STAX	D	12	LD	(DE),A
STC		37	SCF	
SUB	M	96	SUB	(HL)
SUB	R	9—	SUB	R
SUI	N	D6	SUB	N
XCHG		EB	EX	DE,HL
XRA	M	AE	XOR	(HL)
XRA	R	A—	XOR	R
XRI	N	EE	XOR	N
XTHL		E3	EX	(SP),HL

APPENDIX H

Details of the Z-80 and 8080 Instruction Set

A summary of the Z-80 and 8080 instruction set is given in this appendix.* The instructions are listed alphabetically by the official Zilog mnemonic. If there is a corresponding 8080 instruction, the Intel mnemonic is shown in angle brackets; if not, "no 8080" is shown in the angle brackets. The Z-80 mnemonics are listed in numeric order in Appendix F. The Z-80 equivalent of an 8080 mnemonic can be found from the cross-reference given in Appendix G.

The letters A, B, C, D, E, H, I, L, IX, IY, R, and SP are used for the standard Z-80 register names. In addition, the symbols BC, DE, and HL are used for the register pairs. The following symbols are used for general arguments.

r, r2	8-bit CPU register
dd	8-bit signed displacement
nn	general 8-bit constant
nnnn	16-bit constant

The flag bits are represented by:

C	carry
H	half carry
N	add/subtract
P/O	parity/overflow
S	sign
Z	zero

Pointers to memory or input or output addresses are enclosed in parentheses.

---

*More details can be obtained from the Zilog programmer's manual, *Z-80 Assembly Language Programming Manual,* Zilog, Inc., 1977.

```
ADC     A,(HL)   <ADC    M>
```

Add the memory byte pointed to by the HL register pair to the accumulator and the carry flag. The result is placed in the accumulator.

Flags affected: C, H, O, S, Z
Flag reset: N

```
ADC     A,(IX+dd)        <no 8080>
ADC     A,(IY+dd)        <no 8080>
```

Add the memory byte referenced by the sum of the specified index register and the displacement to the accumulator and the carry flag. The result is placed in the accumulator.

Flags affected: C, H, O, S, Z
Flag reset: N

```
ADC     A,r      <ADC    r>
```

Add the value in register r to the accumulator and the carry flag. The result is placed in the accumulator.

Flags affected: C, H, O, S, Z,
Flags reset: N

```
ADC     A,nn     <ACI    nn>
```

Add the constant given in the second operand to the accumulator and the carry flag. The result is placed in the accumulator.

Flags affected: C, H, O, S, Z
Flag reset: N

```
ADC     HL,BC    <no 8080>
ADC     HL,DE    <no 8080>
ADC     HL,HL    <no 8080>
ADC     HL,SP    <no 8080>
```

Add the indicated double register to the HL register and the carry flag. The result is placed in HL.

Flags affected: C, H, O, S, Z,
Flag reset: N

```
ADD     A,(HL)           <ADD    M>
```

Add the memory byte pointed to by the HL pair to the accumulator. The result is placed in the accumulator.

Flags affected: C, H, O, S, Z
Flag reset: N

```
ADD     A,(IX+dd)   <no 8080>
ADD     A,(IY+dd)   <no 8080>
```

Add the memory byte pointed to by the sum of the specified index register and the displacement to the accumulator. The result is placed in the accumulator.

Flags affected: C, H, O, S, Z
Flag reset: N

```
ADD     A,r         <ADD     r>
```

Add the value in register r to the accumulator. The result is placed in the accumulator.

Flags affected: C, H, O, S, Z
Flag reset: N

```
ADD     A,nn        <ADI     nn>
```

Add the immediate byte (the second operand) to the accumulator. The result is placed in A.

Flags affected: C, H, O, S, Z
Flag reset: N

```
ADD     HL,BC       <DAD     B>
ADD     HL,DE       <DAD     D>
ADD     HL,HL       <DAD     H>
ADD     HL,SP       <DAD     SP>
```

Add the specified double register to the HL pair. The result is placed in HL. This is a double-precision addition. The carry flag is set if an overflow occurs. The instruction ADD HL,HL performs a 16-bit arithmetic shift left, effectively doubling the value in HL. The ADD HL,SP instruction can be used with the 8080 CPU to save an incoming stack pointer:

```
LXI     H,0
DAD     SP
SHLD    nnnn
```

Flags affected: C, H, O, S, Z
Flag reset: N

```
ADD     IX,BC            <no 8080>
ADD     IX,DE            <no 8000>
ADD     IX,IX            <no 8080>
ADD     IX,SP            <no 8080>
ADD     IY,BC            <no 8080>
ADD     IY,DE            <no 8080>
ADD     IY,IY            <no 8080>
ADD     IY,SP            <no 8080>
```

Add the indicated double register to the specified index register. The result is placed in the index register. The HL register pair does not participate in this group of instructions. Notice that there is no equivalent series of ADC instructions.

Flags affected: C, O, S, Z
Flag reset: N

```
AND     (HL)     <ANA    M>
```

Perform a logical AND with the accumulator and the memory location pointed to by the HL register pair. The result is placed in the accumulator.

Flags affected: P, S, Z
Flags reset: C, N
Flag set: H

```
AND     (IX+dd)         <no 8080>
AND     (IY+dd)         <no 8080>
```

Perform a logical AND with the accumulator and the memory byte referenced by the sum of the index register and displacement. The result is placed in the accumulator.

Flags affected: P, S, Z
Flags reset: C, N
Flag set: H

```
AND     r        <ANA    r>
```

Perform a logical AND with the accumulator and register r. The result is placed in the accumulator. An instruction AND A is an effective way to test the parity, sign, and zero flags.

Flags affected: P, S, Z
Flags reset: C, N
Flag set: H

```
AND     nn       <ANI    nn>
```

Perform a logical AND with the accumulator and the constant given in the argument. The result is placed in the accumulator. This instruction can be used to selectively reset bits of the accumulator. For example, the instruction AND 7FH will reset bit 7.

Flags affected: P, S, Z
Flags reset: C, N
Flag set: H

```
BIT     b,(HL)           <no 8080>
BIT     b,(IX+dd)        <no 8080>
BIT     b,(IY+dd)        <no 8080>
```

Test bit b of the memory byte referenced by the second operand. Bit b can range from zero through 7. The zero flag is set if the referenced bit is a logical 1; otherwise, it is reset. Thus, the zero flag becomes the complement of the selected bit.

Flag affected: Z
Flag set: H
Flag reset: N

```
BIT     b,r              <no 8080>
```

Test bit b of register r, where b can range from zero through 7. The zero flag is set if the referenced bit is a logical 1. It is reset otherwise.

Flag affected: Z
Flag set: H
Flag reset: N

```
CALL    nnnn     <CALL    nnnn>
```

Unconditional subroutine call to address nnnn. The address of the following instruction is pushed onto the stack.

Flags affected: none

```
CALL    C,nnnn      <CC       nnnn>
CALL    M,nnnn      <CM       nnnn>
CALL    NC,nnnn     <CNC      nnnn>
CALL    NZ,nnnn     <CNZ      nnnn>
CALL    P,nnnn      <CP       nnnn>
CALL    PE,nnnn     <CPE      nnnn>
CALL    PO,nnnn     <CPO      nnnn>
CALL    Z,nnnn      <CZ       nnnn>
```

Conditional subroutine call to address nnnn. The address of the following instruction is pushed onto the stack. The conditions are:

C	carry flag set	(carry)
M	sign flag set	(minus)
NC	carry flag reset	(not carry)
NZ	zero flag reset	(not zero)
P	sign flag reset	(plus)
PE	parity flag set	(parity even)
PO	parity flag reset	(parity odd)
Z	zero flag set	(zero)

```
CCF                  < CMC >
```

Complement the carry flag. This instruction can be given after a SCF command to reset the carry flag.

Flag affected: C
Flag reset: N

```
CP       (HL)       <CMP     M>
```

Compare the byte in memory pointed to by the HL pair to the accumulator (an implied operand). The zero flag is set if the accumulator is equal to the operand. The carry flag is set if the accumulator is smaller than the operand.

Flags affected: C, H, O, S, Z
Flag set: N

```
CP       (IX+dd)            <no 8080>
CP       (IY+dd)            <no 8080>
```

Compare the memory location referenced by the sum of the index register and displacement to the accumulator which is an implied operand. The zero flag is set if the accumulator is equal to the operand. The carry flag is set if the accumulator is smaller than the operand.

Flags affected: C, H, O, S, Z
Flag set: N

```
CP       r          <CMP     r>
```

Compare register r to the accumulator, which is an implied operand. The zero flag is set if the accumulator is equal to the operand. The carry flag is set if the accumulator is smaller than the operand.

Flags affected: C, H, O, S, Z
Flag set: N

```
CP       nn         <CPI     nn>
```

Compare the constant given in the operand to the accumulator, which is an implied operand. The zero flag is set if the accumulator is equal to the operand. The carry flag is set if the accumulator is smaller than the operand.

Flags affected: C, H, O, S, Z
Flag set: N

```
CPD      <no 8080>
CPDR     <no 8080>
CPI      <no 8080>
CPIR     <no 8080>
```

Compare the memory byte pointed to by HL to the accumulator. Decrement HL (if D) or increment HL (if I). Decrement the byte count in the BC register. Repeat the operation for CPDR and CPIR until a match is found or until the BC register pair has been decremented to zero. The zero flag is set if a match is found. The parity flag is set if BC is decremented to zero.

Flags affected: H, S
Flags set: N, Z if A = (HL), P if BC = 0

```
CPL       < CMA >
```

Complement the accumulator. This instruction performs a one's complement on the accumulator. (Compare to the instruction NEG.)

Flags set: H, N

```
DAA       < DAA >
```

Decimal adjust the accumulator. This instruction is used after each addition with BCD numbers. The Z-80 performs this operation properly for both addition and subtraction. The 8080, however, gives an incorrect result for subtraction.

Flags affected: O, S, Z, C, H

```
DEC       (HL)      <DCR     M>
```

Decrement the memory byte pointed to by the HL register pair.

Flags affected: H, O, S, Z
Flag set: N
Flag not affected: C

```
DEC       (IX+dd)            <no 8080>
DEC       (IY+dd)            <no 8080>
```

Decrement the memory byte pointed to by the sum of the index register and the displacement.

Flags affected: H, O, S, Z
Flag set: N
Flag not affected: C

```
DEC       r         <DCR     r>
```

Decrement the register r. Don't try to decrement a register past zero while executing a JP NC loop. The carry flag is not affected by this operation.

Flags affected: H, O, S, Z
Flag set: N
Flag not affected: C

```
DEC     BC      <DCX   B>
DEC     DE      <DCX   D>
DEC     HL      <DCX   H>
DEC     SP      <DCX   SP>
```

Decrement the indicated double register. Don't try to decrement a double register to zero in a JP NZ loop. It won't work since this operation does not affect any of the PSW flags. Instead, move one byte of the double register into the accumulator and perform a logical OR with the other byte.

```
LD   A,C
OR   B
JR   NZ,nnnn
```

Flags affected: none!!!!

```
DEC     IX      <no 8080>
DEC     IY      <no 8080>
```

Decrement the index register.

Flags affected: none

```
DI      < DI >
```

Disable maskable interrupt request.

```
DJNZ    dd      <no 8080>
```

Decrement register B and jump relative to displacement dd if B register is not zero.

Flags affected: none

```
EI      < EI >
```

Enable maskable interrupt request.

```
EX      (SP),HL         < XTHL >
```

Exchange the byte at the stack pointer with register L. Exchange the byte at the stack pointer + 1 with register H.

Flags affected: none

```
EX      (SP),IX         <no 8080>
EX      (SP),IY         <no 8080>
```

Exchange the 16 bits referenced by the stack pointer with the specified index register.

Flags affected: none

```
EX        AF,AF'        <no 8080>
```

Exchange the accumulator and flag register with the alternate set.

Flags affected: all

```
EX        DE,HL         < XCHG >
```

Exchange the double registers DE and HL.

Flags affected: none

```
EXX                   <no 8080>
```

Exchange BC, DE, and HL with the alternate set.

Flags affected: none

```
HALT          < HLT >
```

Suspend operation of the CPU until a reset or interrupt occurs. Dynamic memory refresh continues during a halt.

```
IM        0           <no 8080>
IM        1           <no 8080>
IM        2           <no 8080>
```

Sets interrupt mode 0, 1, or 2. Interrupt mode 0 is automatically selected when a Z-80 reset occurs. The result is the same as the 8080 interrupt response. Interrupt mode 1 performs an RST 38H instruction. Interrupt mode 2 provides for many interrupt locations.

```
IN        r,(C)         <no 8080>
```

9 Input a byte from the port address in register C to register r.

Flags affected: P, S, Z
Flags reset: H, N

```
IN        A,(nn)         <IN    nn>
```

Input a byte from the port address nn to the accumulator.

Flags affected: none

```
INC       (HL)        <INR      M>
```

Increment the memory byte pointed to by the HL pair.

Flags affected: H, O, S, Z
Flag set: N
Flag not affected. C!!!!

```
INC     (IX+dd)         <no 8080>
INC     (IY+dd)         <no 8080>
```

Increment the memory byte pointed to by the sum of the index register and the displacement.

Flags affected: H, O, S, Z
Flag set: N
Flag not affected: C!!!

```
INC     r       <INR    r>
```

Increment the 8-bit register r. Don't try to increment a register past zero while executing a JP NC loop. It won't work because the carry flag is unaffected by this instruction.

Flags affected: H, O, S, Z
Flag set: N
Flag not affected: C!!!

```
INC     BC      <INX    B>
INC     DE      <INX    D>
INC     HL      <INX    H>
INC     SP      <INX    SP>
```

Increment the specified double register.

Flags affected: none!!!!

```
INC     IX      <no 8080>
INC     IY      <no 8080>
```

Increment the specified index register.

Flags affected: none

```
IND         <no 8080>
INDR        <no 8080>
INI         <no 8080>
INIR        <no 8080>
```

Input a byte from the port address in register C to the memory byte pointed to by HL. Decrement register B. The HL register is incremented (if I) or decremented (if D). For INDR and INIR the process is repeated until the 8-bit register B becomes zero.

Flag affected: Z (if B = 0)
Flag set: N

```
JP      (HL)     < PCHL >
```

Copy the HL register pair into the program counter; then retrieve the next instruction from the address referenced by HL. This instruction causes a jump to the address referenced by HL.

Flags affected: none

```
JP      (IX)     <no 8080>
JP      (IY)     <no 8080>
```

Copy the contents of the specified index register into the program counter; then retrieve the next instruction from the address referenced by IX or IY.

Flags affected: none

```
JP      nnnn     <JMP     nnnn>
```

Unconditional jump to address nnnn.

Flags affected: none

```
JP      C,nnnn     <JC      nnnn>
JP      M,nnnn     <JM      nnnn>
JP      NC,nnnn    <JNC     nnnn>
JP      NZ,nnnn    <JNZ     nnnn>
JP      P,nnnn     <JP      nnnn>
JP      PE,nnnn    <JPE     nnnn>
JP      PO,nnnn    <JPO     nnnn>
JP      Z,nnnn     <JZ      nnnn>
```

Conditional jump to address nnnn where:

C	means carry flag set	(carry)
M	means sign flag set	(minus)
NC	means carry flag reset	(not carry)
NZ	means zero flag reset	(not zero)
P	means sign flag reset	(plus)
PE	means parity flag set	(parity even)
PO	means parity flag reset	(parity odd)
Z	means zero flag set	(zero)

```
JR      dd       <no 8080>
```

Unconditional relative jump with a signed displacement nn. The jump is limited to 129 bytes forward and 126 bytes backward in memory.

Flags affected: none

```
JR      C,dd          <no 8080>
JR      NC,dd         <no 8080>
JR      NZ,dd         <no 8080>
JR      Z,dd          <no 8080>
```

Conditional relative jump to address nn where:

C	means carry flag set	(carry)
NC	means carry flag reset	(not carry)
NZ	means zero flag reset	(not zero)
Z	means zero flag set	(zero)

```
LD      (BC),A        <STAX   B>
LD      (DE),A        <STAX   D>
```

Move the byte in the accumulator to the memory byte referenced by the specified register pair.

```
LD      (HL),r        <MOV    M,r>
```

Move the byte in register r to the memory byte pointed to by the HL pair.

```
LD      (HL),nn        <MVI    M,nn>
```

Move the immediate byte nn into the memory byte referenced by the HL pair.

```
LD      (IX+dd),r         <no 8080>
LD      (IX+dd),nn        <no 8080>
LD      (IY+dd),r         <no 8080>
LD      (IY+dd),nn
```

Move the byte in register r or the immediate byte nn into the memory byte referenced by the sum of the index register plus the displacement. These instructions can be used to load relocatable binary code.

```
LD      (nnnn),A          <STA     nnnn>
```

Store the accumulator in the memory location referenced by nnnn.

```
LD      (nnnn),BC         <no 8080>
LD      (nnnn),DE         <no 8080>
```

Store the low-order byte (C or E) of the specified double register at the memory location nnnn. Store the high-order byte (B or D) at nnnn + 1.

```
LD      (nnnn),HL         <SHLD    nnnn>
```

Store register L at the memory address nnnn. Store register H at the address nnnn + 1.

```
LD        (nnnn),IX        <no 8080>
LD        (nnnn),IY        <no 8080>
LD        (nnnn),SP        <no 8080>
```

Store the low-order byte of the specified register IX, IY, or SP at the location nnnn. Store the high-order byte at nnnn + 1. The instruction LD (nnnn),SP can be used to temporarily save an incoming stack pointer. It can later be restored by a LD SP,(nnnn) operation.

```
LD        A,(BC)        <LDAX    B>
LD        A,(DE)        <LDAX    D>
```

Move the memory byte referenced by the specified register pair BC or DE into the accumulator.

```
LD        A,I        <no 8080>
```

Load the accumulator from the interrupt-vector register. The parity flag reflects the state of the interrupt-enable flip-flop.

Flags affected: S, Z, P
Flags reset: H, N

```
LD        A,R        <no 8080>
```

Load the accumulator from the memory-refresh register. The parity flag reflects the state of the interrupt-enable flip-flop. This is an easy way to obtain a fairly decent random number.

Flags affected: S, Z, P
Flags reset: H, N

```
LD        I,A        <no 8080>
```

Copy the accumulator into the interrupt-vector register.

Flags affected: none

```
LD        R,A        <no 8080>
```

Copy the accumular into memory-refresh register.

Flags affected: none

```
LD        r,(HL)        <MOV    r,M>
```

Move the byte in the memory location pointed to by the HL register pair into register r.

```
LD      r,(IX+dd)        <no 8080>
LD      r,(IY+dd)        <no 8080>
```

Move the byte at the memory location referenced by the sum of the index register and the displacement into register r.

```
LD      r,r2     <MOV     r,r2>
```

Move the byte from register r2 to r.

```
LD      r,nn     <MVI     r,nn>
```

Load register r with the 8-bit data byte nn.

```
LD      A,(nnnn)         <LDA     nnnn>
```

Load the accumulator from the memory byte referenced by the 16-bit pointer nnnn.

```
LD      BC,(nnnn)        <no 8080>
LD      DE,(nnnn)        <no 8080>
```

Load the low-order byte (C or E) from the location referenced by the 16-bit pointer nnnn. Load the high-order byte (B or D) from nnnn + 1.

```
LD      HL,(nnnn)        <LHLD    nnnn>
```

Load register L from the address referenced by the 16-bit value nnnn. Load register H from the address nnnn + 1.

```
LD      BC,nnnn          <LXI   B,nnnn>
LD      DE,nnnn          <LXI   D,nnnn>
LD      HL,nnnn          <LXI   H,nnnn>
LD      SP,nnnn          <LXI  SP,nnnn>
LD      IX,nnnn          <no 8080>
LD      IY,nnnn          <no 8080>
```

Load the specified double register with the 16-bit constant nnnn. Be careful not to confuse LD HL,(nnnn) with LD HL,nnnn.

```
LD      IX,(nnnn)        <no 8080>
LD      IY,(nnnn)        <no 8080>
LD      SP,(nnnn)        <no 8080>
```

Load the low byte of IX, IY, or SP from the memory location nnnn. Load the high byte from nnnn + 1. The LD SP,(nnnn) instruction can be used to retrieve a previously saved stack pointer.

```
LD      SP,HL          < SPHL >
LD      SP,IX          <no 8080>
LD      SP,IY          <no 8080>
```

Load the stack pointer from the specified 16-bit register. The SPHL instruction can be used to retrieve a previously saved stack pointer when the 8080 CPU is used.

```
LHLD    nnnn
SPHL
```

```
LDD             <no 8080>
LDDR            <no 8080>
LDI             <no 8080>
LDIR            <no 8080>
```

Move the byte referenced by the HL pair into the location pointed to by the DE register pair. Decrement the 16-bit byte counter in BC. Increment (if I) or decrement (if D) both HL and DE. Repeat the operation for LDDR and LDIR until the BC register has been decremented to zero.

```
NEG             <no 8080>
```

This instruction performs a two's complement on the accumulator. It effectively subtracts the accumulator from zero. To perform this task on an 8080, use a CMA command followed by an INR A command.

Flags affected: all

```
NOP                    < NOP >
```

No operation is performed by the CPU.

Flags affected: none

```
OR      (HL)           <ORA    M>
```

Perform a logical OR with the accumulator and the byte referenced by HL. The result is placed in the accumulator.

Flags affected: P, S, Z
Flags reset: C, H, N

```
OR      (IX+dd)        <no 8080>
OR      (IY+dd)        <no 8080>
```

Perform a logical OR with the accumulator and the byte referenced by the specified index register plus the displacement. The result is placed in the accumulator.

Flags affected: P, S, Z
Flags reset: C, H, N

```
OR      r           <ORA    r>
```

Perform a logical OR with the accumulator and the register r. The result is placed in the accumulator. An instruction of OR A is an efficient way to test the parity, sign, and zero flags.

Flags affected: P, S, Z
Flags reset: C, H, N

```
OR      nn          <ORI    nn>
```

Perform a logical OR with the accumulator and the byte nn. The result is placed in the accumulator. This instruction can be used to set individual bits of the accumulator. For example, OR 20H will set bit 5 to a logical 1.

Flags affected: P, S, Z
Flags reset: C, H, N

```
OTDR                <no 8080>
OTIR                <no 8080>
```

Output a byte from the memory location pointed to by the HL pair. The port address is contained in register C. Register B is decremented. The HL register pair is incremented (if I) or decremented (if D). The process is repeated until register B has become zero.

Flags set: N, Z

```
OUT     (C),r       <no 8080>
```

Output the byte in register r to the port address contained in register C.

Flags affected: none

```
OUT     (nn),A      <OUT   nn>
```

Output the byte in the accumulator to the port address nn.

Flags affected: none

```
OUTD                <no 8080>
OUTI                <no 8080>
```

Output a byte from the memory location pointed to by the HL pair. The port address is contained in register C. Register B is decremented. The HL register pair is incremented (if I) or decremented (if D).

Flag affected: Z
Flag set: N

```
POP     AF      <POP    PSW>
```

Move the byte at the memory location referenced by the stack pointer into the flag register (PSW), and increment the stack pointer. Then move the byte at the location referenced by the new stack-pointer value into the accumulator and increment the stack pointer a second time.

Flags affected: all

```
POP     BC      <POP    B>
POP     DE      <POP    D>
POP     HL      <POP    H>
```

Copy two bytes of memory into the appropriate double register as follows. The memory byte referenced by the stack pointer is moved into the low-order byte (C, E, or L), then the stack pointer is incremented. The memory byte referenced by the new stack pointer is then moved into the high-order byte (B, D, or H). The stack pointer is incremented a second time.

Flags affected: none

```
POP     IX      <no 8080>
POP     IY      <no 8080>
```

Copy the top of the stack into the specified index register. Increment the stack pointer twice.

Flags affected: none

```
PUSH    AF      <PUSH   PSW>
```

Store the accumulator and flag register in memory as follows. The stack pointer is decremented, then the value in the accumulator is moved to the memory byte referenced by the stack pointer. The stack pointer is decremented a second time. The flag register is copied to the byte at the memory address referenced by the current stack-pointer position.

Flags affected: none

```
PUSH    BC      <PUSH   B>
PUSH    DE      <PUSH   D>
PUSH    HL      <PUSH   H>
```

Store the referenced double register in memory as follows. The stack pointer is decremented, then the byte in the specified high-order register B, D, or H is copied to the memory location referenced by the stack pointer. The stack pointer is decremented a second time. The byte in the low-order position C, E, or L is moved to the byte referenced by the current value of the stack pointer.

Flags affected: none

```
PUSH    IX    <no 8080>
PUSH    IY    <no 8080>
```

The indicated index register is copied to the top of the stack. The stack pointer is decremented twice.

Flags affected: none

```
RES     b,(HL)       <no 8080>
RES     b,(IX+dd)    <no 8080>
RES     b,(IY+dd)    <no 8080>
```

Reset bit b of the memory byte referenced by the second operand. Bit b can range from zero through 7.

Flag reset: N
Other flags affected: none

```
RES     b,r          <no 8080>
```

Reset bit b of register r to a value of zero. Bit b can range from zero through 7.

Flag reset: N
Other flags affected: none

```
RET     < RET >
```

Return from a subroutine. The top of the stack is moved into the program counter. The stack pointer is incremented twice.

```
RET     C     < RC >
RET     M     < RM >
RET     NC    < RNC >
RET     NZ    < RNZ >
RET     P     < RP >
RET     PE    < RPE >
RET     PO    < RPO >
RET     Z     < RZ >
```

Conditional return from a subroutine. If the condition is met, the top of the stack is moved into the program counter. The stack pointer is incremented twice.

C	means carry flag set	(carry)
M	means sign flag set	(minus)
NC	means carry flag reset	(not carry)
NZ	means zero flag reset	(not zero)
P	means sign flag reset	(plus)
PE	means parity flag set	(parity even)
PO	means parity flag reset	(parity odd)
Z	means zero flag set	(zero)

```
RETI          <no 8080>
```

Return from maskable interrupt.

```
RETN          <no 8080>
```

Return from nonmaskable interrupt.

The following RL and RLA instructions rotate bits to the left through carry.

```
!->----------->------------->---------------->------!
!                                                   !
!   !---!           !---!---!---!---!---!---!---!---!      !
!-  !   ! <------<-  <-  <-  <-  <-  <-  <-  <-   ! <-!
    !---!           !---!---!---!---!---!---!---!---!
    carry                          register
```

```
RL        (HL)        <no 8080>
```

The memory byte referenced by the HL pair is rotated left through carry. The carry flag moves into bit zero. Bit 7 moves to the carry flag.

Flags affected: C, P, S, Z
Flags reset: H, N

```
RL        (IX+dd)             <no 8080>
RL        (IY+dd)             <no 8080>
```

The memory byte referenced by the sum of the index register and the displacement is rotated left through carry. The carry flag moves into bit zero. Bit 7 moves to the carry flag.

Flags affected: C, P, S, Z
Flags reset: H, N

```
RL        r           <no 8080>
```

The byte in register r is rotated left through carry. The carry flag moves into bit zero. Bit 7 moves to the carry flag. Note: the instruction RL A performs the same task that the separate instruction RLA does, but the former takes twice as long as the latter.

Flags affected: C, P, S, Z
Flags reset: H, N

```
RLA       < RAL >
```

The byte in the accumulator is rotated left through carry. The carry flag moves to bit zero. Bit 7 of the accumulator moves to the carry flag.

Flag affected: C
Flags reset: H, N

The following RLC and RLCA instructions rotate bits to the left.

```
                 !->------------->----------------->-------!
                 !                                          !
 !---!           !    !---!---!---!---!---!---!---!---!     !
 !   <------!---<-   <-   <-   <-   <-   <-   <-   <-----!
 !---!                !---!---!---!---!---!---!---!---!
 carry                         register
```

```
RLC     (HL)     <no 8080>
```

The byte referenced by the HL pair is rotated left circularly. Bit 7 moves to both the zero bit and to the carry flag.

Flags affected: C, P, S, Z
Flags reset: H, N

```
RLC     (IX+dd)        <no 8080>
RLC     (IY+dd)        <no 8080>
```

The byte referenced by the specified index register plus the displacement is rotated left circularly. Bit 7 moves to both bit zero and the carry flag.

Flags affected: C, P, S, Z
Flags reset: H, N

```
RLC     r        <no 8080>
```

The byte in register r is rotated left circularly. Bit 7 moves to both bit zero and the carry flag. Note: RLC A performs the same task that another instruction RLCA does, but the former instruction takes twice as long as the latter.

Flags affected: C, P, S, Z
Flags reset: H, N

```
RLCA    < RLC >
```

The accumulator is rotated left circularly. Bit 7 moves to both bit zero and the carry flag.

Flag affected: C
Flags reset: H, N

```
RLD            <no 8080>
```

A 4-bit rotation over 12 bits. The low 4 bits of A move to the low 4 bits of the memory location referenced by the HL pair. The original low 4 bits of memory move to the high 4 bits. The original high 4 bits move to the low 4 bits of A. This instruction is used for BCD operations.

Flags affected: P, S, Z
Flags reset: H, N

```
                    !--<---<-!    !-<----<-!
                    !        !    !        !
!--------!--------!     !--------!--------!
! 4 bits ! 4 bits !     ! 4 bits ! 4 bits !
!--------!--------!     !--------!--------!
  accumulator     !          memory       !
                  !-->--------->------>-!
```

The following RR and RRA instructions rotate bits to the right through carry.

```
!-------<-------------<----------------<---------<-!
!                                                  !
!    !---!---!---!---!---!---!---!---!       !---! !
!->  !  ->  ->  ->  ->  ->  ->  ->  ->-----> !   !-!
     !---!---!---!---!---!---!---!---!       !---!
              register                       carry
```

```
RR      (HL)     <no 8080>
```

The memory byte pointed to by the HL pair is rotated right through carry. Carry moves to bit 7. Bit zero moves to the carry flag.

Flags affected: C, P, S, Z
Flags reset: H, N

```
RR      (IX+dd)          <no 8080>
RR      (IY+dd)          <no 8080>
```

The memory byte pointed to by the specified index register plus the offset is rotated right through carry. The carry flag moves to bit 7. Bit zero moves to the carry flag.

Flags affected: C, P, S, Z
Flags reset: H, N

```
RR      r          <no 8080>
```

The byte in register r is rotated right through carry. Carry moves to bit 7. Bit zero moves to the carry flag. Note: RR A performs the same task that another instruction RRA does, but the former instruction takes twice as long as the latter.

Flags affected: C, P, S, Z
Flags reset: H, N

```
RRA             < RAR >
```

The accumulator is rotated right through carry. The carry flag moves to bit 7. Bit zero moves to the carry flag.

Flag affected: C
Flags reset: H, N

The following RRC and RRCA instructions rotate bits to the right.

```
!-------<------------<---------------<-!
!                                      !
!     !---!---!---!---!---!---!---!---!  !     !---!
!-->  !  ->  ->  ->  ->  ->  ->  ->  ->--!----->  !
      !---!---!---!---!---!---!---!---!        !---!
                  register                     carry
```

```
RRC     (HL)     <no 8080>
```

The memory byte pointed to by the HL pair is rotated right circularly. Bit zero moves to both the carry flag and bit 7.

Flags affected: C, P, S, Z
Flags reset: H, N

```
RRC     (IX+dd)          <no 8080>
RRC     (IY+dd)          <no 8080>
```

The memory byte pointed to by the index register plus the offset is rotated right circularly. Bit zero moves to both the carry flag and bit 7.

Flags affected: C, P, S, Z
Flags reset: H, N

```
RRC     r        <no 8080>
```

The byte in register r is rotated right circularly. Bit zero moves to both the carry flag and bit 7. Note: RRC A performs the same task that another instruction RRCA does, but the former instruction takes twice as long as the latter.

Flags affected: C, P, S, Z
Flags reset: H, N

```
RRCA            < RRC >
```

The accumulator is rotated right circularly. Bit zero moves to both the carry flag and bit 7.

Flag affected: C
Flags reset: H, N

```
RRD       <no 8080>
```

A 4-bit rotation over 12 bits. The low 4 bits of A move to the high 4 bits of the memory location referenced by the HL pair. The original high 4 bits of memory move to the low 4 bits. The original low 4 bits move to the low 4 bits of A. This instruction is used for BCD operations.

Flags affected: P, S, Z
Flags reset: H, N

```
                     !-->--->-!      !->---->-!
                     !        !      !        !
!--------!--------!      !--------!--------!
! 4 bits ! 4 bits !      ! 4 bits ! 4 bits !
!--------!--------!      !--------!--------!
 accumulator      !          memory       !
                  !--<--------<------<-!
```

```
RST 00H         <RST 0>
RST 08H         <RST 1>
RST 10H         <RST 2>
RST 18H         <RST 3>
RST 20H         <RST 4>
RST 28H         <RST 5>
RST 30H         <RST 6>
RST 38H         <RST 7>
```

These restart instructions generate one-byte subroutine calls to address given in the Z-80 operand.

```
SBC       A,(HL)   <SBB     M>
```

Subtract the carry flag and the memory byte pointed to by the HL pair from the accumulator. The result is placed in the accumulator. Some Z-80 assemblers allow the 8080 mnemonic of SBB to be used as an alternate Z-80 mnemonic for this instruction.

Flags affected: C, H, O, S, Z
Flag set: N

```
SBC       A,(IX+dd)       <no 8080>
SBC       A,(IY+dd)       <no 8080>
```

Subtract the carry flag and the memory byte pointed to by the sum of the index register and displacement from the accumulator. The result is placed in the accumulator.

Flags affected: C, H, O, S, Z
Flag set: N

```
SBC     A,r      <SBB    r>
```

Subtract the carry flag and the specified CPU register from the accumulator. The result is placed in the accumulator. Some Z-80 assemblers allow the 8080 mnemonic of SBB to be used as an alternate Z-80 mnemonic for this instruction.

Flags affected: C, H, O, S, Z
Flag set: N

```
SBC     A,nn     <SBI    nn>
```

Subtract the immediate byte (the second operand) and the carry flag from the accumulator. The result is placed in the accumulator. Some Z-80 assemblers allow the 8080 mnemonic of SBB to be used as an alternate Z-80 mnemonic for this instruction.

Flags affected: C, H, O, S, Z
Flag set: N

```
SBC     HL,BC    <no 8080>
SBC     HL,DE    <no 8080>
SBC     HL,HL    <no 8080>
SBC     HL,SP    <no 8080>
```

Subtract the specified CPU double register and the carry flag from the HL register pair. The result is placed in HL. You may need to reset the carry flag with an OR A operation before using these instructions.

Flags affected: C, H, O, S, Z
Flag set: N

```
SCF              < STC >
```

Set the carry flag. There is no equivalent reset command. However, the carry flag can be reset with the XOR A instruction, or with the pair of instructions SCF and CCF.

Bit set: C
Bits reset: H, N

```
SET     b,(HL)          <no 8080>
SET     b,(IX+dd)       <no 8080>
SET     b,(IY+dd)       <no 8080>
```

Set bit b of the memory byte referenced by the second operand. Bit b can range from zero through 7.

Flag reset: N
Other flags affected: none

```
SET     b,r                <no 8080>
```

Set bit b of register r. Bit b can range from zero through 7.

Flag reset: N
Other flags affected: none

The following SLA instructions shift bits to the left.

```
!---!                  !---!---!---!---!---!---!---!---!
!   ! <-------<-  <-  <-  <-  <-  <-  <-  <-   ! <-- 0
!---!                  !---!---!---!---!---!---!---!---!
carry                           register
```

```
SLA     (HL)     <no 8080>
```

Perform an arithmetic shift left on the memory byte pointed to by the HL pair. Bit 7 is moved to the carry flag. A zero is moved into bit zero. This operation doubles the original value.

Bits affected: C, P, S, Z
Bits reset: H, N

```
SLA     (IX+dd)        <no 8080>
SLA     (IY+dd)        <no 8080>
```

Perform an arithmetic shift left on the memory byte pointed to by the index register plus the displacement. Bit 7 is moved to the carry flag. A zero is moved into bit zero. This operation doubles the original value.

Bits affected: C, P, S, Z
Bits reset: H, N

```
SLA     r        <no 8080>
```

Perform an arithmetic shift left on register r. Bit 7 is moved to the carry flag. A zero is moved into bit zero. This operation doubles the original value. Note: SLA A performs the same task that another instruction ADD A,A does, but the former instruction takes twice as long as the latter.

Bits affected: C, P, S, Z
Bits reset: H, N

The following SRA instructions shift bits to the right.

```
       !---!---!---!---!---!---!---!---!                    !---!
!-->   !   ->  ->  ->  ->  ->  ->  ->  ->------>            !   !
!      !---!---!---!---!---!---!---!---!                    !---!
!          !          register                              carry
!---<---!
```

```
SRA      (HL)     <no 8080>
```

Perform an arithmetic shift right on the memory byte pointed to by the HL pair. Bit zero moves to the carry flag. Bit 7 is duplicated in bit 6.

Bits affected: C, P, S, Z
Bits reset: H, N

```
SRA      (IX+dd)           <no 8080>
SRA      (IY+dd)           <no 8080>
```

Perform an arithmetic shift right on the byte pointed to by the index register plus the displacement. Bit zero moves to carry and bit 7 is duplicated into bit 6.

Bits affected: C, P, S, Z
Bits reset: H, N

```
SRA      r        <no 8080>
```

Perform an arithmetic shift right on register r. Bit zero moves to carry and bit 7 is duplicated into bit 6. The operation effectively halves the register value. The carry flag represents the remainder. The carry flag is set if the original number was odd.

Bits affected: C, P, S, Z
Bits reset: H, N

The following SRL instructions shift bits to the right.

```
         !---!---!---!---!---!---!---!---!          !---!
0 -->    !   ->  ->  ->  ->  ->  ->  ->  ->------>  !   !
         !---!---!---!---!---!---!---!---!          !---!
                    register                        carry
```

```
SRL      (HL)     <no 8080>
```

Perform a logical shift right on the byte pointed to by the HL register pair. A zero bit is moved into bit 7. Bit zero moves to the carry flag.

Bits affected: C, P, Z
Bits reset: H, N, S

```
SRL      (IX+dd)           <no 8080>
SRL      (IY+dd)           <no 8080>
```

Perform a logical shift right on the byte pointed to by the index register plus the displacement. A zero bit is moved into bit 7. Bit zero moves to the carry flag.

Bits affected: C, P, Z
Bits reset: H, N, S

```
SRL     r       <no 8080>
```

Perform a logical shift right on register r. A zero bit is moved into bit 7. Bit zero moves to the carry flag.

Bits affected: C, P, S, Z
Bits reset: H, N

```
SUB     (HL)    <SUB    M>
```

Subtract the memory byte referenced by the HL pair from the accumulator. The result is placed in the accumulator.

Flags affected: C, H, O, S, Z
Flag set: N

```
SUB     (IX+dd)         <no 8080>
SUB     (IY+dd)         <no 8080>
```

Subtract the memory byte referenced by the index register plus the displacement from the value in the accumulator. The result is placed in A.

Flags affected: C, H, O, S, Z
Flag set: N

```
SUB     r       <SUB    r>
```

Subtract the specified CPU register from the accumulator. The result is placed in the accumulator. Notice that Z-80 SUB mnemonic has only one operand, whereas there are two operands in the corresponding 8-bit ADC, ADD, and SBC mnemonics. The destination operand for all four of these instructions is always the accumulator. Consequently, the first operand is optional with some assemblers.

Flags affected: C, H, O, S, Z
Flag set: N

```
SUB     nn      <SUI    nn>
```

Subtract the immediate byte nn from the accumulator. The result is placed in the accumulator.

Flags affected: C, H, O, S, Z
Flag set: N

```
XOR     (HL)          <XRA    M>
```

Perform a logical exclusive OR with the accumulator and the byte referenced by the HL pair. The result is placed in the accumulator.

Flags affected: P, S, Z
Flags reset: C, H, N

```
XOR     (IX+dd)          <no 8080>
XOR     (IY+dd)          <no 8080>
```

Perform a logical exclusive OR with the accumulator and the byte referenced by the sum of specified index register and the displacement. The result is placed in the accumulator.

Flags affected: P, S, Z
Flags reset: C, H, N

```
XOR     r             <XRA    r>
```

Perform a logical exclusive OR with the accumulator and register r. The result is placed in the accumulator. The XOR A instruction is an efficient way to zero the accumulator, although all the flags are then reset. XOR A is also used to reset the carry flag.

Flags affected: P, S, Z
Flags reset: C, H, N

```
XOR     nn            <XRI    nn>
```

Perform a logical exclusive OR with the accumulator and the byte nn. The result is placed in the accumulator.

Flags affected: P, S, Z
Flags reset: C, H, N

APPENDIX I

Abbreviations and Acronyms

A/D	Analogue to Digital
APL	A Programming Language
ASCII	American Standard Code for Information Interchange
BCD	Binary-Coded Decimal
BDOS	Basic Disk-Operating System (CP/M)
BIOS	Basic Input/Output System (CP/M)
Bit	BInary digiT
BJT	Bipolar Junction Transistor
CBIOS	Customized Bios (CP/M)
CCD	Charge-Coupled Device
CCP	Console Command Processor (CP/M)
CMOS	Complementary Metal-Oxide Semiconductor
COBOL	COmmon Business-Oriented Language
CP/M	Control Program for Microcomputers
CPU	Central Processing Unit
CRC	Cyclical-Redundancy Check
CRT	Cathode Ray Tube
CTS	Clear To Send
D/A	Digital to Analogue
DCD	Data-Carrier Detect
DDT	Dynamic Debugging Tool (CP/M)
DMA	Direct Memory Access
DOS	Disk-Operating System
ECL	Emitter-Coupled Logic
EOF	End Of File
EPROM	Erasable PROM
FCB	File Control Block (CP/M)
FDOS	Full DOS (CP/M)
FET	Field-Effect Transistor
FIFO	First-In, First-Out
FORTRAN	FORmula TRANslation
Hz	Hertz (cycles per second)
IC	Integrated Circuit
IGFET	Insulated-Gate FET

IIL	Integrated Injection Logic
I/O	Input/Output
JFET	Junction FET
LCD	Liquid Crystal Display
LED	Light-Emitting Diode
LIFO	Last-In, First-Out
LSB	Least-Significant Bit (or Byte)
LSI	Large-Scale Integration
MHz	Megahertz (million Hz)
MODEM	MODulator/DEModulator
MOS	Metal-Oxide Semiconductor
MOSFET	MOS FET
MSB	Most Significant Bit (or Byte)
NAND	Not AND
NOR	Not OR
OP AMP	OPerational AMPlifier
PIP	Peripheral Interchange Program (CP/M)
PPL	Phase-Locked Loop
PROM	Programmable ROM
PSW	Program-Status Word
R/W	Read/Write
RAM	Random-Access Memory
ROM	Read-Only Memory
RTS	Request To Send
SCR	Silicon-Controlled Rectifier
SID	Symbolic Instruction Debugger (CP/M)
TPA	Transient Program Area (CP/M)
TTL	Transistor-Transistor Logic
TTY	Teletype
UART	Universal Asynchronous Receiver/Transmitter
UJT	UniJunction Transistor
XOR	Exclusive OR

APPENDIX J

Undocumented Z-80 Instructions

The Z-80 CPU contains two 16-bit index registers called IX and IY. All of the official Zilog instructions which refer to IX and IY are 16-bit operations. For example, the IX register can be assigned the constant value 1234 with the LD instruction

```
LD        IX,1234H
```

And the contents of the IY register can be saved on the stack with a PUSH operation.

```
PUSH      IY
```

A perusal of the numerically ordered Z-80 instructions, given in Appendix F, reveals that sequences beginning with the byte DD hex refer to operations involving the IX register. Similarly, the byte FD hex signifies an IY operation.

In addition to the 16-bit operations, there are many 8-bit operations that can be performed with the Z-80 index registers. These instructions are not shown in Appendix F because they have not been officially documented by Zilog. All of these 8-bit IX and IY instructions follow the same pattern. The first byte of the IX operation codes is DD hex and the first byte of the IY operation codes is FD hex. The remaining byte of the instruction is a regular Z-80 operation which refers to the H or L register. But, in this case, H now refers to the high 8 bits of the index register and L refers to the low 8 bits.

As an example, consider the Z-80 instruction:

```
LD        H,B
```

which moves the byte in register B into the H register. If this operation code is preceded by a DD hex byte, then the B register is moved into the high 8 bits of the IX register. An assembler that incorporated these undocumented instructions might generate the following source code.

```
60        LD        H,B       ;MOVE B TO H
DD60      LD        XH,B      ;MOVE B TO HIGH IX
DD6B      LD        XL,E      ;MOVE E TO LOW IX
FD67      LD        YH,A      ;MOVE A TO HIGH IY
FD69      LD        YL,C      ;MOVE C TO LOW IY
```

But, except for Allen Ashley's PDS assembler, these XH, XL, YH, and YL symbols have not actually been incorporated into Z-80 assemblers. The undocumented instructions encompass all of the 8-bit LD op codes involving the H and L registers. These include the register-to-register moves

```
LD      H,r
LD      L,r
LD      r,H
LD      r,L
```

where r is one of the general-purpose registers A, B, C, D, and E. The instructions

```
LD      L,H   and
LD      H,L
```

are also included. In this case the H and L refer respectively to the high 8 bits and the low 8 bits of the same index register. Thus, the instruction

```
LD      L,H
```

preceded by a DD byte will move the high 8 bits of the IX register into the low 8 bits. The regular H and L registers cannot be operands for these moves. Move-immediate instructions are not included in the new instructions. Furthermore, 8-bit transfers cannot be made from one index register to the other.

Many of the other 8-bit register operations can be utilized in this way. In addition to the above LD instructions, these include the following.

```
ADC     A,H     ADC     A,L
ADD     A,H     ADD     A,L
AND     H       AND     L
CP      H       CP      L
DEC     H       DEC     L
INC     H       INC     L
OR      H       OR      L
SBC     A,H     SBC     A,L
SUB     H       SUB     L
XOR     H       XOR     L
```

The rotations and shifts, the bit manipulations that set, reset and test, and the input and output operations are not included.

All of these undocumented operations can be produced with a regular Z-80 assembler. One method is to generate the initial DD or FD hex with the DEFB directive. Then, the corresponding regular Z-80 instruction follows. For example, the following two lines will generate the instruction needed to move the accumulator into the high half of IX.

```
DEFB    0DDH    ;SET FOR IX
LD      H,A     ;MOVE A TO XH
```

Alternately, a macro can be used. The definition could be

```
LDX        MACRO      REG1,REG2
           DEFB       0DDH
           LD         REG1,REG2
           ENDM
```

Then the macro call

```
LDX        H,A
```

will generate the desired code.

Index

Accumulator, 3
Acoustical coupler, 65
Acronyms, 311
Active high, 93
Analog-to-digital, 188
Arithmetic shift, 76, 155, 180, 307
ASCII, 20, 66, 150
    character set, 253
Assembler, 2, 86
Autostart tape, 201

Base conversion, 150
BCD, 19, 168
Binary, 17
Binary coded decimal, 19, 168
BIOS (CP/M), 91, 216
Block move, 13, 111, 297
Branch table, 107
Buffered input, 55
Bus, 1

Carry flag, 5, 10
Checksum, 68, 188
Clock routine, 241
Code, operation, 1
Cold start, 103
Compare 7, 123
Complement, 21
Concatination, 72
Conditional branch, 12, 287, 293
Control character, 93, 99, 100
Conversion, number base, 150
CP/M, 51, 86, 91, 123, 175, 201, 214
CPU, 1
Crazy octal, 164, 182
Cross reference
    8080 to Z-80, 280
    Z-80 to 8080, 283

DAA, 179
Data port, 49
Data ready, 91
Debugger, 94
Decimal adjust accumulator, 179
Decrement, 8, 10
Delay, after carriage return, 148
Delimiter, 151
De Morgan's theorem, 28
Digital-to-analog, 188
Directive, 87
DOS, 214
Double precision arithmetic, 22, 180
Double register, 3, 10, 13
Doubling, 15, 155, 307
Dummy variable, 70
Dump, memory, 95

Echo, 95
Editor, 86, 143
Exclusive OR, 26

FCB (CP/M), 226, 242
File control block, 226, 242
Filename, 204
Fixed entry, 152
Flags, 3, 6, 10
    I/O, 50, 71, 93
Free entry, 152
Full duplex, 65
Function number (CP/M), 216

Gates, 21
GO program, 226

Halving, 15
Handshake, 49
Hex arithmetic, 119
Hex format, 201
Hexadecimal numbers, 17
High-level language, 2, 214

Increment, 8, 10
Index register, 313
Input, 14, 48, 117, 146
Instruction set
    alphabetic, 8080, 258
    alphabetic, Z-80, 264
    numeric, 8080, 261
    numeric, Z-80, 272
    details, 283

Intel hex format, 201
Interrupt-driven keyboard, 55
Interrupt register, 11
Interrupts, 52
IOBYTE (CP/M), 217, 220

Jump, relative, 12, 144

Label, 2
  local, 82
Labels, readable, 203
Leading-zero suppression, 173
LIFO stack, 30
List routine, 229
Loader routine, 125
Logical AND, 23, 224
Logical devices, 215
Logical exclusive OR, 26
Logical operations, 7, 20
Logical OR, 23
Logical shift, 15
Looping, 50

Macros, 69, 143
Magnetic tape, 187
Masking AND, 25, 50, 178, 220
Memory allocation (CP/M), 217
Memory
  map, 255
  pointer, 2, 55, 113
  test, 120
Memory-mapped port, 48
Mnemonic, 2
Modem, 64
Monitor, 85, 128
Movement of data, 111
Multiplication, 15, 157

NAND gate, 26
Nibble, 178
NOR gate, 26
Nulls, carriage return,
    77, 93, 219
Number-base conversion, 150

Object program, 2
Octal, 16, 163, 180
Offset, 202
One's complement, 21
Opcode, 1
Operand, 2
Operation code, 1
Output, 14, 48, 71, 117, 146, 215

Packed BCD, 19, 168
Page, memory, 105
Paper tape, 187
Parallel port, 49
Parity, 66
Passing data, 37, 39
Peripheral, 1, 14
  port, 48, 117, 146
Physical devices, 218
PIP (CP/M), 143, 230
Pointer, 2, 12, 55
Polling, 52
POP, 31
Port, I/O, 48, 117, 146
  initialization, 146
Printer routine, 147
Program counter, 3, 11, 36
Program status word, 3, 34
PSW, 3, 34
PUSH, 31

Register
  flag, 3
  memory, 5
  refresh, 11
  saving, 33
  status, 2
Register pair, 3
Registers
  8080, 3
  Z-80, 11
Relative jump, 12, 144
Resetting a bit, 25
Rotation, 9
RST instruction, 53

Scroll, 64, 99
Searching, 113, 115
Serial port, 49
Setting a bit, 24
Shift, 14, 76, 155, 307
Signed number, 6, 15
Source program, 2, 214
Split octal, 164, 182
Spooling, 52
Stack, 30
  automatic placement, 45
  saving registers on, 33
  setting up new, 40
  storing data on, 31
Stack pointer, 31, 109
STAT (CP/M), 225
Status port, 49
String, 150
Subroutine call, 35, 287
Subtraction, 22, 177

Table, command branch, 107
Tape recording, 187
Tape, test, 212
Telephone modem, 64
Top-down method, 85
Truth table, 21
Two's complement, 21, 74, 75, 177, 201

Unconditional branch, 12
Undocumented Z-80 instructions, 313
Unsigned number, 6

Vectors, 103, 219
Vectored interrupt, 53

Warm start, 103

Z-80 instruction macros, 73
Z-80 registers, 11

COMPUTERS $12.95

8080/Z80 Assembly Language

Techniques For Improved Programming

By Alan R. Miller

The reveiwers are unanimous:

"Just about everything you'll ever want to know about assembly language programming,... a reference work you'll use for years to come." — *Microcomputing*

"This book is for intermediate and advanced programmers, but its clarity and organization can provide even a beginning programmer with an avenue to advanced techniques." —*Interface Age*

"A truly definitive work on 8080/Z80 Assembly Language Programming—it should be on every programmer's bookshelf! Whether you are a CP/M expert or just a rank beginner at assembly language programming, this book represents a combined tutorial and reference that you will return to often." — *Lifelines*

"Bravo, Alan Miller!" — *80 Micro*

For both intermediate and advanced programmers, this complete guide to programming the 8080 and Z80 microprocessors lets you get every response your computer is capable of generating, and enables you to perform much more complex and sophisticated operations.

Learn the details of assembly language programming—easily and quickly—as you develop a powerful system monitor in a step-by-step, top-down approach. Over 100 pages of programs are included to let you develop, write, and test your own routines.

You'll start with number bases and logical operations, then move on to branching, rotation and shifting, one's and two's complement arithmetic, and stack operations. You'll find out how your assembly language programs can utilize the CP/M operating system for all input and output. There's an entire chapter devoted to assembler macros, and another whole chapter on input and output operations. Ten indispensable appendices contain all of the reference materials needed to write 8080 or Z80 assembly language programs.

Alan R. Miller, Professor of Metallurgy at the New Mexico Institute of Technology, is a Contributing Editor to *Interface Age.* He holds a Ph.D. in Engineering from the University of California, Berkeley.

Photo: Courtesy Intel Corporation

JOHN WILEY & SONS
605 Third Avenue, New York, N.Y. 10158
New York • Chichester • Brisbane • Toronto • Singapore ISBN 0 471 08124-8