Christopher L. Morgan

Christopher L. Morgan is a professor at California State University, Hayward, where he teaches mathematics and computer science, including computer graphics, assembly language programming, computer architecture, and operating systems. Dr. Morgan has given talks and authored papers in pure mathematics and on representations of higher-dimensional objects on computers. He is director of the computer graphics lab at Hayward and is a member of a number of professional associations, including the American Mathematical Society, the National Council of Teachers of Mathematics, and the Association for Computing Machinery. He is coauthor along with Mitchell Waite of *8086/8088 16-Bit Microprocessor Primer* and *Graphics Primer for the IBM® PC*. He is the author of *Bluebook of Assembly Language Routines for the IBM® PC and XT*.

HIDDEN POWERS OF THE TRS-80® MODEL 100

by Christopher L. Morgan

A Plume/Waite Book
New American Library
New York and Scarborough, Ontario

SIGNET, SIGNET CLASSIC, MENTOR, PLUME, MERIDIAN and NAL BOOKS are published *in the United States* by New American Library, 1633 Broadway, New York, New York 10019, *in Canada* by The New American Library of Canada Limited, 81 Mack Avenue, Scarborough, Ontario M1L1M8

Cover design by Michael Manwaring
Illustrations by Winston and Karen Sin
Typography by Walker Graphics

First Printing, November, 1984

1 2 3 4 5 6 7 8 9

PRINTED IN THE UNITED STATES OF AMERICA

Contents

Acknowledgments

There are a number of people I would like to thank for their valuable help in making this book possible.

At the Waite Group, Robert Lafore has provided support, encouragement, guidance, and feedback which made my job at least an order of magnitude easier; Lyn Cordell has done a superb job in managing the production of the book; and Mitchell Waite initiated and oversaw the entire operation.

I would like to thank my wife Carol and my children Elizabeth and Thomas for their patience with me while I worked on this book.

Preface

This book is for anyone who wants to understand the inner secrets of the TRS-80® Model 100 portable computer. It will be equally useful to those who need to learn about the Model 100 in particular and to those who want to increase their general understanding of computers. This book and the Model 100 will provide a new world of fascinating study, and they'll both fit in your briefcase!

The Model 100 is the first of a new breed of lap-size computers that have launched a revolution in the way computing is done. No longer are we chained to our desks by a heavy machine requiring a wall plug and cumbersome peripherals. Now we can slip a full-fledged computer into a briefcase and take it with us on the airplane, to a client's office, or to a construction site.

Hidden Powers of the TRS-80® Model 100 reveals how this amazing machine works, on a level seldom glimpsed by the casual user or programmer. Using simple, down-to-earth language, this book explains the computer's hardware and the built-in software that make the Model 100 so powerful for its diminutive size.

Who Can Profit from This Book

If you are a programmer looking for ways to enhance your programs — to give them professional polish and speed — this book will show you how to use the powerful *undocumented routines* built into the Model 100's Read Only Memory. These routines can be accessed from either assembly language or BASIC; this book shows you how. Accessing these routines directly will make your programs more powerful and versatile. For instance, you will learn how to perform selective scrolling (scrolling only a few lines of the display), how to shift the display to "reverse video" (white on black) for special effects, and how to dial the telephone directly from BASIC.

If you are a programmer, learning how such hardware devices as the Liquid Crystal Display and keyboard work will enable you to perform tasks that are ordinarily impossible, such as setting up real-time displays of complex data or detecting any number of keys pressed simultaneously on the keyboard. Uses for such techniques include the simulation of instrument panels and other machines, both for games and for more serious programs.

If you are a hardware designer who would like to market a peripheral device for the Model 100, you will find it essential to know how the Model 100's hardware works and how it interfaces with the outside world and with the built-in software. This book will tell you what you need to know in order to design equipment that will work successfully in the Model 100 environment.

Finally, you may simply be curious about how computers work. *Hidden Powers* is written in a simple, jargon-free style. Anyone who has some familiarity with programming should be able to understand it, and it can open up a whole new area of exploration for someone who is learning about computers. You'll find that the computer is not a simple dumb beast waiting for your typed-in program. Instead, it is a surprisingly intelligent and complex machine that will reward your study and investigation.

Because many of the principles and chips used in the Model 100 are common to other computers as well, this in-depth investigation will help you to acquire an understanding of computers in general. The techniques and information presented in this book are similar, except in detail, to those you would use to investigate most other modern computers.

What You Need to Know to Use This Book

Any programmer who is familiar with a higher-level language such as BASIC should be able to profit from this book. The general explanations of the hardware and of the built-in ROM routines require no knowledge of assembly language. Many of the routines described can be called directly from BASIC programs. Also, example programs are given in BASIC so that you can experiment with the operation of the hardware.

Use of the disassembler program for further investigation of the Model 100's built-in ROM routines will require some knowledge of 8085 assembly language and of the hexadecimal numbering system. If you have access to a CP/M® system, a good book for learning about 8085 assembly language is *Soul of CP/M*, by Mitchell Waite and Robert Lafore (Indianapolis: Howard Sams & Co. Inc., 1983).

What's in This Book

The first few chapters in *Hidden Powers* provide some tools and background. You'll need these to understand the more detailed explanations of the individual components of the Model 100 that follow in later chapters.

Chapter 1 gives an overview of the Model 100, a quick once-over of the various elements that make up its hardware and software. Then some programming tools are introduced. The most important of these is a disassembler. This program, written in BASIC, permits you to dig into the Model 100's ROM, to explore its every nuance and detail. In subsequent chapters, *Hidden Powers* provides entry points to and explains the use of all the most common ROM routines. With these as starting points, you can use the disassembler to investigate individual routines and determine exactly how they work. Other BASIC programs are provided to display key areas in the ROM routines and to search for specific patterns in the Model 100's memory.

Chapter 2 provides a detailed look at the architecture of the Model 100. The 80C85 Central Processing Unit is examined first, followed by the bus structure and the chips that handle input/output operations. If you are unfamiliar with computer hardware, you will find that this chapter opens the door to a new and fascinating world.

Chapters 3 through 9 investigate individual components of the Model 100: memory, display, real-time clock, keyboard, communications devices, sound, and cassette system. For each of these subjects the hardware is explained first. Then the ROM routines that control the hardware are explored in detail. Entry addresses and parameters are summarized for these routines so that BASIC and assembly language programs can call them directly.

Throughout these chapters short BASIC programs allow you to gain easy "hands on" experience with particular parts of the machine. These example programs should make many aspects of the Model 100's operation accessible even to those who are not assembly language programmers.

When you finish *Hidden Powers*, you'll have a thorough, detailed understanding of the Model 100 and its operation. If you have never investigated a computer in such depth before, you'll find that a fascinating new world has been opened to you. No longer will you be at the mercy of the incomprehensible "mysteries" in your computer: you'll have the technique and the knowledge to dig in, explore, and understand what's *really* going on in the Model 100 or almost any other computer system.

1

Exploring the Model 100

Concepts
Overview of the Model 100
Some tools for probing the Model 100
Notes on using this book

This chapter introduces the TRS-80 Model 100 Portable Computer, points you to the appropriate written material, and provides the essential software tools that will start you toward understanding the operation of the Model 100. There is also a discussion of the major features of this book, including its boxes describing the ROM routines and its programs written in BASIC.

The techniques and much of the information presented in this chapter and in the rest of the book will carry over to other computers, not only from Radio Shack but from other companies as well. This is because most microcomputers are organized on the same basic principles and use many of the same chips.

Overview of the Model 100

Let's take a quick look at the major features of the Model 100 to get our bearings before we make our first explorations into its inner structure.

The TRS-80 Model 100 Portable Computer is a truly remarkable piece of engineering. It is the size of a notebook, yet it is a full-fledged computer with facilities to perform the three major functions of modern information-handling systems: computing, text entry, and communications.

The 64K bytes of memory space in the Model 100 are divided into two halves. The lower 32K bytes consist of ROM (Read Only Memory), and the upper 32K bytes contain RAM (Random Access Memory). You may have less RAM than this in your particular Model 100.

The Model 100's ROM houses its built-in programs. In Chapter 3 we will explore the entire contents of the ROM. Remarkably, the entire 32K of ROM is located on one chip.

The RAM contains user files (programs and text files) and the data needed to run the operating system. Not all 32K bytes of RAM need to be installed. The basic model comes with 8K bytes, and you can add more RAM in 8K increments until you fill the entire 32K bytes of available space.

The Model 100 uses a master menu to tie all its functions together. The master menu displays the names of the various programs and data files stored in the system. The file system resides entirely in the RAM and ROM memory of the computer, providing quick and easy access to a wide range of functions, features, and modes.

Unlike the situation with many computers, when you turn off the Model 100 in the normal manner (using the switch on the right side of the machine), the information in RAM is not lost. Thus both RAM and ROM behave like permanent memory, making secondary storage techniques such as cassette tape and disk not as vital as in many computers. You can store your favorite BASIC and machine-language programs in the main memory as long as you need them.

The file directory is stored in RAM, and the files it lists can be in either ROM or RAM. The Model 100 has five ROM files and as many as nineteen RAM files. Since the information in RAM does not disappear when the Model 100 is turned off, it makes sense to have this many RAM files active in the system at once. We will study the master menu and its directory in Chapter 3.

The programming power of the Model 100 comes from its Microsoft® BASIC interpreter. BASIC is the first item on the master menu. The BASIC in the Model 100 has the same syntax as the BASICs that come with most personal computers. However, because of size limitations, it is missing a few features. For example, there are no user-defined functions written in BASIC. Also, because of the memory-based file structure, saving and loading files is somewhat different — in some ways, better — than on disk-based systems, in that files are automatically updated as you work on them.

The code for the BASIC interpreter is contained in the first part of the Model 100's ROM. BASIC keeps its working data (variables, pointers, and buffers) in the highest part of memory, which is RAM. BASIC programs are stored throughout the rest of the Model 100's RAM along with other files (see Figure 1-1). In Chapter 3 we will explore the inner workings of the BASIC interpreter. You will see how BASIC interprets command lines, and you will learn where routines to perform all commands are located.

The Model 100 has its own built-in text editor, called TEXT. This is the second entry in the master menu. The code for TEXT is contained in ROM, above the code for BASIC. TEXT stores its variables and other kinds of working data in high memory in the same areas that BASIC uses for this purpose. In Chapter 3 we will also discuss how the TEXT program works.

A program called TELCOM allows you to move files conveniently between the Model 100 and other computers. The code for TELCOM resides in ROM, above the code for BASIC and below the code for TEXT. TELCOM uses the RS-232C serial port and the modem to send and receive files.

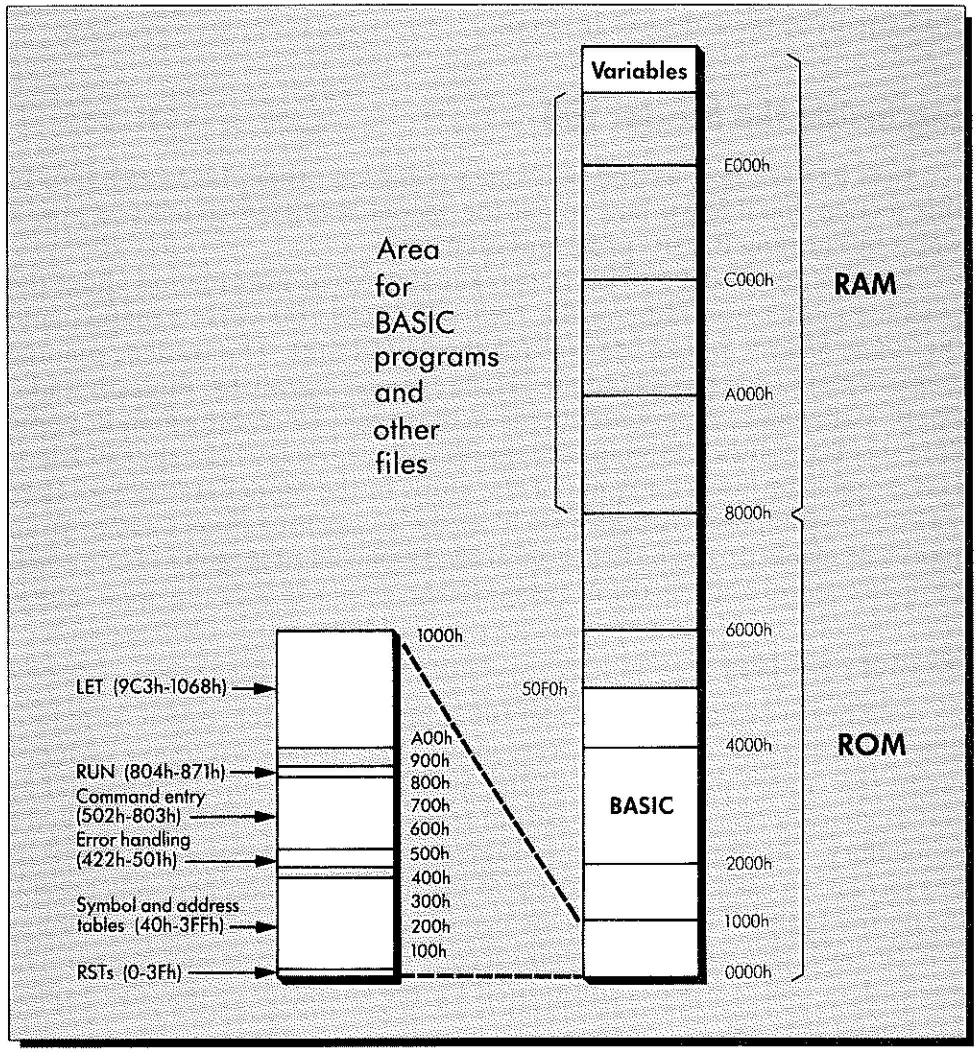


Figure 1-1. Memory map of BASIC

In Chapter 7 we will discuss these devices in detail, seeing how to program them directly, and in Chapter 3 we will examine the TELCOM program itself.

The Model 100 contains two other ROM programs, SCHEDL and ADDRSS. We will see how they fit into the scheme of things in Chapter 3.

The Model 100 has a wide variety of input and output devices, including a Liquid Crystal Display, a keyboard, a modem, an RS-232C serial port, a sound generator, a tape cassette interface, a printer interface, and a bar code reader interface. In Chapter 2 we will see how these various devices are arranged within the Model 100, and in Chapters 4 through 9 we will study many of these devices in detail.

Peeking Inside the Model 100

Now let's see how to find out what's going on inside the Model 100. In this section you'll learn how to get the Model 100 itself to tell you how it works by running BASIC programs on it.

Besides the Model 100 itself, two key sources of information about the machine's inner workings are the *TRS-80® Model 100* owner's manual and the *Radio Shack® Service Manual* for the Model 100. The owner's manual comes with the computer, and the *Service Manual* is available through Radio Shack stores and computer centers. A third source of information is the data sheets published by chip companies such as Intel, whose specifications and designs are used for some of the chips in the Model 100. These data sheets are compiled in books such as the *Component Data Catalog* from Intel®, available through their Literature Department as well as from some computer stores and electronic parts stores.

Memory Display

Let's begin exploring the Model 100 by writing a simple BASIC program that will display the contents of its memory. This display is a simple memory dump that shows the contents of the Model 100 as decimal values and as ASCII characters. Each line of output from this program displays six bytes of memory. This is the maximum number of bytes that can be displayed in this format. In each line the address of the first byte comes first, followed by the decimal value for each of the six bytes and finally by their corresponding ASCII characters. If a byte contains a control code such as carriage return, linefeed, or bell, a space is substituted for the character so that the display will not be affected.

```
100 ' DECIMAL/ASCII DUMP
110 '
120   FOR I = 0 TO 32767 STEP 6
130 '
140 ' ADDRESS
150    PRINT USING "#####";I;
160    PRINT "  ";
170 '
180 ' DECIMAL BYTES
190    FOR K = 0 TO 5
200     PRINT USING "####";PEEK(I+K);
210    NEXT K
220    PRINT " ";
230 '
240 ' ASCII CHARACTERS
250    FOR K = 0 TO 5
260     X = PEEK(I+K) AND 127
270     IF X<32 THEN X=32
280     PRINT CHR$(X);
290    NEXT K
300 '
310    PRINT
320   NEXT I
```

Looking at the program in detail, you can see that it consists of a FOR loop with index I that ranges from 0 to 32767, the entire length of the ROM. The STEP size for this loop is 6, corresponding to the fact that six bytes are displayed per line.

On line 140 the address of the first byte of the line is displayed. The PRINT USING statement formats the address to ensure that it always takes up the same number of spaces on the screen.

On lines 190-210, a FOR loop displays the decimal values of six bytes of memory. The PEEK function is used to fetch the contents of these bytes, and the PRINT USING command is used to ensure that their values appear evenly spaced on the screen.

On lines 250-290, a FOR loop displays the ASCII symbols corresponding to the same six bytes. On line 260, the PEEK function gets the value from memory. The value is ANDed with 127 to eliminate (make zero) the highest bit from the byte. Sometimes this bit is set (set to one) in the computer's memory to indicate when a character is the last letter of a name. We want to display characters as though the highest bit were not set; otherwise, we will get erroneous graphics characters when the highest bit is on. On line 270, the program checks for control codes and replaces them by the value 32, which is the ASCII code for a space. If we left the control codes alone, the display would get messed up every once in a while. On line 280, the character is printed out using the CHR$ function.

Run this program now to see what's in the Model 100's memory. Starting at location 128, you should see the keywords corresponding to BASIC's commands. This is the first step in understanding how and where BASIC works. Notice that each keyword begins with a byte that has a high value (see the decimal values for these bytes). In fact, the numerical code for the initial letter of each keyword is equal to the ASCII code of the letter plus the value 128. This is the result of turning on (setting to 1) the highest bit of the byte. You can guess that BASIC uses this fact when it searches through this list of keywords.

After you run this program as it is, you can change the limits on line 120 and run it to explore other areas of memory. For example, if you look at high memory, you might discover where the directory is stored. Try replacing line 120 by:

```
12Ø  FOR I = 63842 TO 64139 STEP 6
```

You should recognize the filenames in your directory as they go by. In Chapter 3, we'll show you a program that does a much better job at displaying the directory.

BASIC Keywords

Now let's look at BASIC's keywords in detail. The keywords form a complete guide to BASIC's commands and functions. Almost every BASIC command begins with a keyword that identifies the action to be done. Any command that doesn't start with a keyword is assumed to be a LET command; thus, even the commands that do not begin with a keyword have one assumed.

Here's a program that displays the keywords in tabular form. It lists all 127 BASIC keywords, numbered from 0 to 126.

```
1ØØ  ' COMMAND TABLE
11Ø  '
12Ø  FOR I = 128 TO 6Ø7
13Ø   X=PEEK(I)
14Ø   IF X<128 THEN 19Ø
15Ø   PRINT
16Ø   PRINT USING "###";K;
17Ø   K=K+1
18Ø   PRINT "  ";
19Ø   PRINT CHR$(X AND 127);
2ØØ  NEXT I
```

The program loops through all the characters from location 128 to 607 (see line 120). You can find these limits by running the previous memory dump program. On line 130, the ASCII code of the character is fetched from memory. On line 140, the program in effect looks for values greater than or equal to 128, indicating the beginning of a keyword. If it finds such values, lines 150-180 are executed. They number the output lines. The variable K keeps track of the numbering. On line 190, the individual characters of the keywords are printed out.

The table produced by this program does several things. First of all, it gives a complete list of the keywords, even the ones that are not documented in the manual because they are not supported by Radio Shack. Second, it provides an ordering or numbering for the keywords. In Chapter 3 we'll see how this numbering is used to index into tables that give the address of the routines to execute BASIC commands and functions.

The Disassembler

The next step in our investigation of the Model 100 is to write a *disassembler*. A disassembler is a program that converts the raw machine language in the computer into assembly language that we can read. In this section we give a listing for a disassembler that is written in BASIC. This disassembler provides a window into the ROM, allowing us to see exactly how the Model 100 handles each command and controls each of its devices. Each routine discussed in this book will be identified by its address and a name. When you enter the address, the disassembler program will display the assembly code that starts there. You can use the PAUSE key to stop and start the display from the disassembler as you read through the discussion in this book. You can use the SHIFT BREAK keys to terminate the program so that you can start it up at a new address as the discussion shifts to a new location in the ROM.

When you run this program, it will ask you for a starting address and then an ending address. Both of these addresses must be entered as four-digit hexadecimal numbers. Each address in this book will be given in hexadecimal as well as in decimal representation so that you can always enter it directly into the disassembler program.

Once the starting and ending addresses have been entered, the program starts displaying the assembly-language equivalents of the machine language stored in the memory between these addresses. For each machine-language instruction the program will display a single line on the screen. On this line you will see the address of the first byte of the instruction, then the name (mnemonic) of the instruction, and finally any operands (data or address of the data).

A few words of warning about using any disassembler are in order. Not all of a computer's memory is filled with machine language. Even the ROM contains tables of data. If you try to disassemble memory locations that do not contain machine language, you will get nonsensical assembly language as your output. Generally, this is pretty easy to recognize when it occurs, and you can run the disassembler to find out where the various tables are located in ROM. However, it is often difficult to determine the exact place where machine language ends and data begins or vice versa. In particular, when your first address falls in the middle of a CPU instruction, the first few lines of output will often be incorrect.

You should type in the disassembler program and save it in your Model 100. You can save it as a RAM file, but you should also back it up by saving it on cassette tape or on the optional disk drive, or by uploading to another computer where it can be saved on disk.

Here is the disassembler.

```
100 ' PROGRAM DISASSEMBLER
110 '
120 ' THIS PROGRAM DISASSEMBLES
130 ' MACHINE CODE.  WHEN THE
140 ' PROGRAM SIGNS ON,
150 ' SPECIFY A STARTING ADDRESS
160 ' AND AN ENDING ADDRESS.
170 '
180   CLEAR 2000
190   GOTO 470 ' GOTO MAIN PROGRAM
200 '
210 ' SUBROUTINE -- HEXADECIMAL BYTE
220 R2=INT(A/16) ' UPPER DIGIT
230 R1=A-R2*16   ' LOWER DIGIT
240 '
250 R2=R2+48:IF R2>57 THEN R2=R2+7
260 A$=CHR$(R2)
270 R1=R1+48:IF R1>57 THEN R1=R1+7
280 A$=A$+CHR$(R1)
290 RETURN
300 '
310 ' SUBROUTINE -- HEXADECIMAL WORD
320 J=I
330 Q2=INT(J/256):Q1=J-Q2*256
340 A=Q2:GOSUB 210:I$=A$
350 A=Q1:GOSUB 210:I$=I$+A$
360 RETURN
370 '
```

```
380 ' SUBROUTINE -- HEX WORD TO DEC
390 J=0
400 FOR K=1 TO 4
410 G=ASC(MID$(I$,K,1))-48
420 IF G>9 THEN G=G-7
430 J=16*J+G
440 NEXT K
450 RETURN
460 '
470 ' MAIN PROGRAM
480 '
490 DIM C$(256),L(256)
500 FOR I=0 TO 255:READ C$(I):NEXT I
510 FOR I=0 TO 255:READ L(I):NEXT I
520 '
530 INPUT "STARTING ADDRESS:";I$
540 GOSUB 380:I1=J
550 INPUT "ENDING ADDRESS  :";I$
560 GOSUB 380:I2=J:I=I1
570 '
580 ' TOP OF MAIN LOOP
590 GOSUB 310 ' HEX WORD
600 P$="H"+I$+"  "
610 '
620 ' GET FIRST BYTE
630 X=PEEK(I):I=I+1:P$=P$+C$(X)
640 IF L(X)=0 THEN 780
650 '
660 ' GET SECOND BYTE
670 A=PEEK(I):I=I+1
680 GOSUB 210:Y$=A$
690 IF L(X)=2 THEN 730
700 P$=P$+"BY"+Y$
710 GOTO 780
720 '
730 ' GET THIRD BYTE
740 A=PEEK(I):I=I+1
750 GOSUB 210:P$=P$+"H"+A$+Y$
760 '
770 ' PRINT THE LINE
780 PRINT P$
790 '
800 ' EXTRA LINE FOR JMP OR RET?
810 IF X<>195 AND X<>201 THEN 860
820 P$=";"
830 PRINT P$
840 '
```

```
850 ' LOOP BACK
860 IF I>I2 THEN STOP
870 GOTO 580
880 '
890 ' THE DATA SECTION
900 DATA "NOP",    "LXI B,"
910 DATA "STAX B", "INX B"
920 DATA "INR B",  "DCR B"
930 DATA "MVI B,", "RLC"
940 DATA "-",      "DAD B"
950 DATA "LDAX B", "DCX B"
960 DATA "INR C",  "DCR C"
970 DATA "MVI C,", "RRC"
980 DATA "-",      "LXI D,"
990 DATA "STAX D", "INX D"
1000 DATA "INR D",  "DCR D"
1010 DATA "MVI D,", "RAL"
1020 DATA "-",      "DAD D"
1030 DATA "LDAX D", "DCX D"
1040 DATA "INR E",  "DCR E"
1050 DATA "MVI E,", "RAR"
1060 DATA "RIM",    "LXI H,"
1070 DATA "SHLD ",  "INX H"
1080 DATA "INR H",  "DCR H"
1090 DATA "MVI H,", "DAA"
1100 DATA "-",      "DAD H"
1110 DATA "LHLD ",  "DCX H"
1120 DATA "INR L",  "DCR L"
1130 DATA "MVI L,", "CMA"
1140 DATA "SIM",    "LXI SP,"
1150 DATA "STA ",   "INX SP"
1160 DATA "INR M",  "DCR M"
1170 DATA "MVI M,", "STC"
1180 DATA "-",      "DAD SP"
1190 DATA "LDA ",   "DCX SP"
1200 DATA "INR A",  "DCR A"
1210 DATA "MVI A,", "CMC"
1220 DATA "MOV B,B","MOV B,C"
1230 DATA "MOV B,D","MOV B,E"
1240 DATA "MOV B,H","MOV B,L"
1250 DATA "MOV B,M","MOV B,A"
1260 DATA "MOV C,B","MOV C,C"
1270 DATA "MOV C,D","MOV C,E"
1280 DATA "MOV C,H","MOV C,L"
1290 DATA "MOV C,M","MOV C,A"
1300 DATA "MOV D,B","MOV D,C"
1310 DATA "MOV D,D","MOV D,E"
1320 DATA "MOV D,H","MOV D,L"
1330 DATA "MOV D,M","MOV D,A"
```

```
1340 DATA "MOV E,B","MOV E,C"
1350 DATA "MOV E,D","MOV E,E"
1360 DATA "MOV E,H","MOV E,L"
1370 DATA "MOV E,M","MOV E,A"
1380 DATA "MOV H,B","MOV H,C"
1390 DATA "MOV H,D","MOV H,E"
1400 DATA "MOV H,H","MOV H,L"
1410 DATA "MOV H,M","MOV H,A"
1420 DATA "MOV L,B","MOV L,C"
1430 DATA "MOV L,D","MOV L,E"
1440 DATA "MOV L,H","MOV L,L"
1450 DATA "MOV L,M","MOV L,A"
1460 DATA "MOV M,B","MOV M,C"
1470 DATA "MOV M,D","MOV M,E"
1480 DATA "MOV M,H","MOV M,L"
1490 DATA "HLT",    "MOV M,A"
1500 DATA "MOV A,B","MOV A,C"
1510 DATA "MOV A,D","MOV A,E"
1520 DATA "MOV A,H","MOV A,L"
1530 DATA "MOV A,M","MOV A,A"
1540 DATA "ADD B",  "ADD C"
1550 DATA "ADD D",  "ADD E"
1560 DATA "ADD H",  "ADD L"
1570 DATA "ADD M",  "ADD A"
1580 DATA "ADC B",  "ADC C"
1590 DATA "ADC D",  "ADC E"
1600 DATA "ADC H",  "ADC L"
1610 DATA "ADC M",  "ADC A"
1620 DATA "SUB B",  "SUB C"
1630 DATA "SUB D",  "SUB E"
1640 DATA "SUB H",  "SUB L"
1650 DATA "SUB M",  "SUB A"
1660 DATA "SBB B",  "SBB C"
1670 DATA "SBB D",  "SBB E"
1680 DATA "SBB H",  "SBB L"
1690 DATA "SBB M",  "SBB A"
1700 DATA "ANA B",  "ANA C"
1710 DATA "ANA D",  "ANA E"
1720 DATA "ANA H",  "ANA L"
1730 DATA "ANA M",  "ANA A"
1740 DATA "XRA B",  "XRA C"
1750 DATA "XRA D",  "XRA E"
1760 DATA "XRA H",  "XRA L"
1770 DATA "XRA M",  "XRA A"
1780 DATA "ORA B",  "ORA C"
1790 DATA "ORA D",  "ORA E"
1800 DATA "ORA H",  "ORA L"
1810 DATA "ORA M",  "ORA A"
1820 DATA "CMP B",  "CMP C"
```

```
1830 DATA "CMP D",  "CMP E"
1840 DATA "CMP H",  "CMP L"
1850 DATA "CMP M",  "CMP A"
1860 DATA "RNZ",    "POP B"
1870 DATA "JNZ ",   "JMP "
1880 DATA "CNZ ",   "PUSH B"
1890 DATA "ADI ",   "RST 0"
1900 DATA "RZ",     "RET"
1910 DATA "JZ ",    "-"
1920 DATA "CZ ",    "CALL "
1930 DATA "ACI ",   "RST 1"
1940 DATA "RNC",    "POP D"
1950 DATA "JNC ",   "OUT "
1960 DATA "CNC ",   "PUSH D"
1970 DATA "SUI ",   "RST 2"
1980 DATA "RC",     "-"
1990 DATA "JC ",    "IN "
2000 DATA "CC ",    "-"
2010 DATA "SBI ",   "RST 3"
2020 DATA "RPO",    "POP H"
2030 DATA "JPO ",   "XTHL"
2040 DATA "CPO ",   "PUSH H"
2050 DATA "ANI ",   "RST 4"
2060 DATA "RPE",    "PCHL"
2070 DATA "JPE ",   "XCHG"
2080 DATA "CPE ",   "-"
2090 DATA "XRI ",   "RST 5"
2100 DATA "RP",     "POP PSW"
2110 DATA "JP ",    "DI"
2120 DATA "CP ",    "PUSH PSW"
2130 DATA "ORI ",   "RST 6"
2140 DATA "RM",     "SPHL"
2150 DATA "JM ",    "EI"
2160 DATA "CM",     "-"
2170 DATA "CPI ",   "RST 7"
2180 DATA 0,2,0,0
2190 DATA 0,0,1,0
2200 DATA 0,0,0,0
2210 DATA 0,0,1,0
2220 DATA 0,2,0,0
2230 DATA 0,0,1,0
2240 DATA 0,0,0,0
2250 DATA 0,0,1,0
2260 DATA 0,2,2,0
2270 DATA 0,0,1,0
2280 DATA 0,0,2,0
2290 DATA 0,0,1,0
2300 DATA 0,2,2,0
```

```
2310 DATA 0,0,1,0
2320 DATA 0,0,2,0
2330 DATA 0,0,1,0
2340 DATA 0,0,0,0
2350 DATA 0,0,0,0
2360 DATA 0,0,0,0
2370 DATA 0,0,0,0
2380 DATA 0,0,0,0
2390 DATA 0,0,0,0
2400 DATA 0,0,0,0
2410 DATA 0,0,0,0
2420 DATA 0,0,0,0
2430 DATA 0,0,0,0
2440 DATA 0,0,0,0
2450 DATA 0,0,0,0
2460 DATA 0,0,0,0
2470 DATA 0,0,0,0
2480 DATA 0,0,0,0
2490 DATA 0,0,0,0
2500 DATA 0,0,0,0
2510 DATA 0,0,0,0
2520 DATA 0,0,0,0
2530 DATA 0,0,0,0
2540 DATA 0,0,0,0
2550 DATA 0,0,0,0
2560 DATA 0,0,0,0
2570 DATA 0,0,0,0
2580 DATA 0,0,0,0
2590 DATA 0,0,0,0
2600 DATA 0,0,0,0
2610 DATA 0,0,0,0
2620 DATA 0,0,0,0
2630 DATA 0,0,0,0
2640 DATA 0,0,0,0
2650 DATA 0,0,0,0
2660 DATA 0,0,2,2
2670 DATA 2,0,1,0
2680 DATA 0,0,2,0
2690 DATA 2,2,1,0
2700 DATA 0,0,2,1
2710 DATA 2,0,1,0
2720 DATA 0,0,2,1
2730 DATA 2,0,1,0
2740 DATA 0,0,2,0
2750 DATA 2,0,1,0
2760 DATA 0,0,2,0
2770 DATA 2,0,1,0
2780 DATA 0,0,2,0
```

```
2790 DATA 2,Ø,1,Ø
2800 DATA Ø,Ø,2,Ø
2810 DATA 2,Ø,1,Ø
2820 '
2830 END
```

Let's look at the program in detail. As you can see, most of the listing consists of DATA statements. Lines 900-2170 contain the mnemonics for all the CPU instructions, and lines 2180-2810 contain the number of additional bytes required for each instruction. These are the bytes needed for associated data or address information.

The first part of the program consists mainly of subroutines for converting numbers between decimal and hexadecimal notation. The main part of the program begins on line 470 and extends to line 870. The first few lines of the main part of the program dimension and load the arrays C$ and L, which hold the mnemonics and the instruction lengths as stored in the DATA statements. The next few lines (530-560) input the starting and ending addresses for the disassembly. The subroutine at line 380 is called to convert from hexadecimal to the normal decimal internal form for numbers. Upon return, the starting value is stored in the variables I1 and I, and the ending value is stored in the variable I2.

Line 580 marks the top of the main loop. Here the current address in the variable I is converted into a hexadecimal value stored in the string I$. On line 600, the string P$ is defined to contain the beginning of the output line. You can see that the output begins with the address of the instruction, prefixed by an "H". This makes each address into a label for the line of assembly language. We used "H" to stand for hexadecimal address.

Next, the first byte of instruction is fetched into the variable X and looked up in the lists C$ and L. The mnemonic in C$ is added to the output string P$. If the instruction requires only one byte, then the program jumps to the end of the loop, where the line is printed out.

If there is a second byte, it is picked up. If there is no third byte, the second byte is prefixed by a "BY" and added to the output string P$, and then the program jumps to the end of the loop, where P$ is printed. We used "BY" to indicate that the data is stored in a byte.

If there is a third byte, it is prefixed with an "H", packed with the second byte, and added to P$. This makes a label operand, perhaps matching one of the "H" labels at the beginning of one of the other output lines of this program. At the bottom of the loop, P$ is printed out in this case as well.

At the very bottom of the main loop, a check is made for JuMP or RETurn instructions. Following these, we place an extra blank comment line to make the code more readable.

This disassembler program can be used to examine each routine that we discuss in this book and to discover other routines as well. It can be modified in a number of ways. For example, it can be changed to produce a list of addresses referenced by the machine code together with the addresses of the instructions where those references occur. These references can be dumped into a file and sorted, giving a list of cross-references. From this list you can find the key entry points and variables of the machine-code programs.

Other BASIC programs can be written using the same hexadecimal conversion subroutines. For example, it is easy to write a program that displays the memory as a list of hexadecimal 16-bit words. This is useful for displaying tables of addresses. For example, such a table containing the addresses of the routines to handle all the BASIC commands starts at location 80h = 128d.

Searching for Special Patterns

From time to time it is important to find certain patterns in memory. For example, you will see on pages 4-7 that output ports F0h = 240d through FFh = 255d are used for the Liquid Crystal Display screen. Hence the key instruction for control of the LCD is:

```
OUT  port
```

where port is a number from F0h to FFh (240 to 255 decimal).

Let's write a program that searches for all instances of this instruction. The machine code for the OUT instruction is 211 decimal. Here is a short program that prints out the address of each occurrence of the pattern: 211, x, where x is a decimal number between 240 and 255.

```
100 ' PATTERN SEARCH
110 '
120   FOR I = Ø TO 32767
130    X = PEEK(I)
140    IF X<>211 THEN 180
150    Y = PEEK(I+1)
160    IF Y<240 THEN 180
170    PRINT I;
180  NEXT I
```

The program consists of a FOR loop that ranges through the entire ROM for address I from 0 to 32767 (see line 120). On line 130, the value of the byte is fetched, and on line 140 it is checked to see if it is equal to 211, the code for the OUT instruction. The next part of the FOR loop is skipped if a match is not made. If the match is made, the next byte is checked. If it is not between 240 and 255, the next part of the FOR loop is skipped. If this second byte value is between 240 and 255, the address I is printed out and the loop continues.

When you run this program, you will discover that there are six locations where this sequence occurs. In Chapter 4, when we study the LCD in more detail, we will discuss how the machine language at these locations actually programs the LCD.

This program can be easily modified to look for other patterns. For example, you could change it to look for specified three-byte sequences.

Notes on Using This Book

Chapters 1 and 2 of this book are introductory, providing you with an overview, guidance, and tools that you will need for reading and making use of the rest of the book.

Each of the remaining chapters (3 through 9) covers a major area of the operation of the Model 100. These chapters start out with a general discussion and then delve into a particular set of ROM routines.

The Boxes

Key information about major ROM routines is displayed in boxes. Each box shows the vital statistics for one particular routine or block of code.

Each box begins with the *name* of the routine. Some routines have a single word in capital letters as their name. These are the routines that are described in documents and articles published by Radio Shack (see Radio Shack publication 700-2245, *Model 100 ROM Functions*). Other routines have names consisting of several words. These are not mentioned in Radio Shack's literature.

After the name comes the *purpose* of the routine. This is a short description of the major function that the routine performs.

Following the purpose is the *entry point* for the routine. This is the address where execution of the routine begins. The entry address is given in both hexadecimal and decimal form, as described below.

The *input* and *output* for the routine are shown next. These describe how the registers and memory are used to pass data in and out of the routine. Here we also often mention the physical I/O devices that are involved.

Next, a BASIC example is given, if applicable. Normally this is in the form of a CALL command. The BASIC CALL command allows you to call a machine-language routine, passing values to it via the A register and the HL register pair. The CALL command does not allow you to read directly into your program any data that might be stored in a register, such as the A register. Because of these conditions, we have given BASIC examples just for those routines that input data through only the A and HL registers and do not output data through CPU registers.

Each box ends with a spot for *special comments*. Here we place general comments or warnings about using the routine.

Address Formats

As noted above, memory addresses are shown in both hexadecimal and decimal format. For example, 646h = 1606d. Here the small letter "h" following the 646 indicates a hexadecimal number, and the small "d" following 1606 indicates a decimal number. Entry points and other addresses are thus accessible both to the BASIC programmer, who needs addresses in decimal, and to the assembly-language programmer, who normally works in hexadecimal (as when using the disassembler).

Although some BASICs have built-in functions to use both decimal and hexadecimal forms of numbers, the Model 100's BASIC can work only with decimal notation. We have also provided simple routines for converting between decimal and hexadecimal, but you would not want to include these routines in every program that CALLed a ROM routine.

BASIC Programs

In this book there are more than two dozen BASIC programs that you can type in and run on the Model 100. You can store these programs as RAM files and/or as cassette files on tape. Many people store such programs by uploading them to a larger system, where they are saved on disk, or you can use the optional Model 100 disk. The programs fall into three main classes: *tools*, *demonstrations*, and *applications*.

Programs in the first category, *tools*, are presented in Chapter 1. Here we have programs that dump, disassemble, and search memory.

The second category, *demonstrations*, is designed to give you a better feel for the way particular features work. Sometimes we display key memory locations that are affected by a process. Sometimes we show the direct results of a software action upon some hardware. By running and modifying these programs yourself, you should gain the deeper understanding that is possible only through "hands on" experience.

The third category, *applications*, somewhat overlaps the last category. Applications programs demonstrate features and give useful and interesting applications for machine language and machine-level features of the Model 100. For example, we have included programs that interactively interpret formulas, scroll selected lines of the LCD display screen, and automatically dial a telephone number.

Summary

In this chapter we have discussed how this book is organized and how to use it. We have also presented some basic program tools that can be used in conjunction with this book. With these tools, you can better understand this book and also extend your knowledge to areas beyond the scope of these pages.

2

The Model 100's Hardware

Concepts

80C85 central processor
Bus structure
ROM and RAM
Port decoding
8155 parallel input/output interface controller
μPD 1990AC real-time clock
6402 universal asynchronous receiver transmitter
LCD screen
Keyboard
Printer interface

In this chapter we'll explore the organization of the Model 100's hardware. We will begin with an overview of the entire internal structure and then survey each of the major subsystems. Some of these subsystems are represented by chips such as the 80C85 CPU, the 8155 PIO, the μPD 1990 clock, and the 6402 UART. Other systems such as the memory, LCD, keyboard, and printer involve combinations of chips and other components. We will also study the buses that connect all of these subsystems together physically and logically to form the Model 100 computer.

Why Study Hardware?

Before we plunge into the hardware, let's talk about why it is useful for a programmer to know about such things. The answer is simple: the hardware is the stage upon which the software plays. The hardware physically houses the software and gives it meaning. More explicitly, the machine-language instructions of the CPU are part of the hardware, and without them, the programs stored in the ROM and RAM are merely sequences of numbers. This notion extends beyond the CPU. Each of the other major chips, including the 8155 PIO, the μPD 1990 clock, and the 6402 UART, can be programmed according to certain rules that involve sending bytes out certain I/O ports. Without these rules, these byte sequences are meaningless. To understand these rules, you need an understanding of the structure of the machine and its various subsystems. Thus, to understand the software, at least at the machine-language level, you must understand the hardware.

However, you should not take this to mean that you need to understand every detail in this chapter in order to profit from the rest of the book. Much of the material in this chapter will be relevant only in certain programming situations. Thus, while you are reading this chapter you should not worry if some aspects of the operation of the hardware are not completely clear. The detailed descriptions of the ROM routines in future chapters will clarify many points.

Each chip in the Model 100 is fully described in the documentation published by the company that designed it. For the programmable chips, this includes all their programming rules. Much of this information is also included in the service manual for the Model 100. We will present the essential details of this material in this chapter.

Hardware Overview

We start with a map of the Model 100 hardware system (see Figure 2-1).

The CPU is the nerve center of the system. It controls the main bus, which runs through the system like a spinal column, linking the various subsystems to the CPU. Two devices, the bar code reader and the tape cassette recorder/player interface, connect directly to the CPU.

The main system bus controls the serial communications lines through the 6402 UART chip, the RAM and ROM, and the 8155 PIO chip. It also helps to control and monitor the keyboard, clock, and LCD display. The main system bus is terminated in a socket in a recessed area on the bottom of the Model 100. A socket for the optional ROM is also here. This socket can be used for other purposes as well, such as connecting the disk drive and video interface.

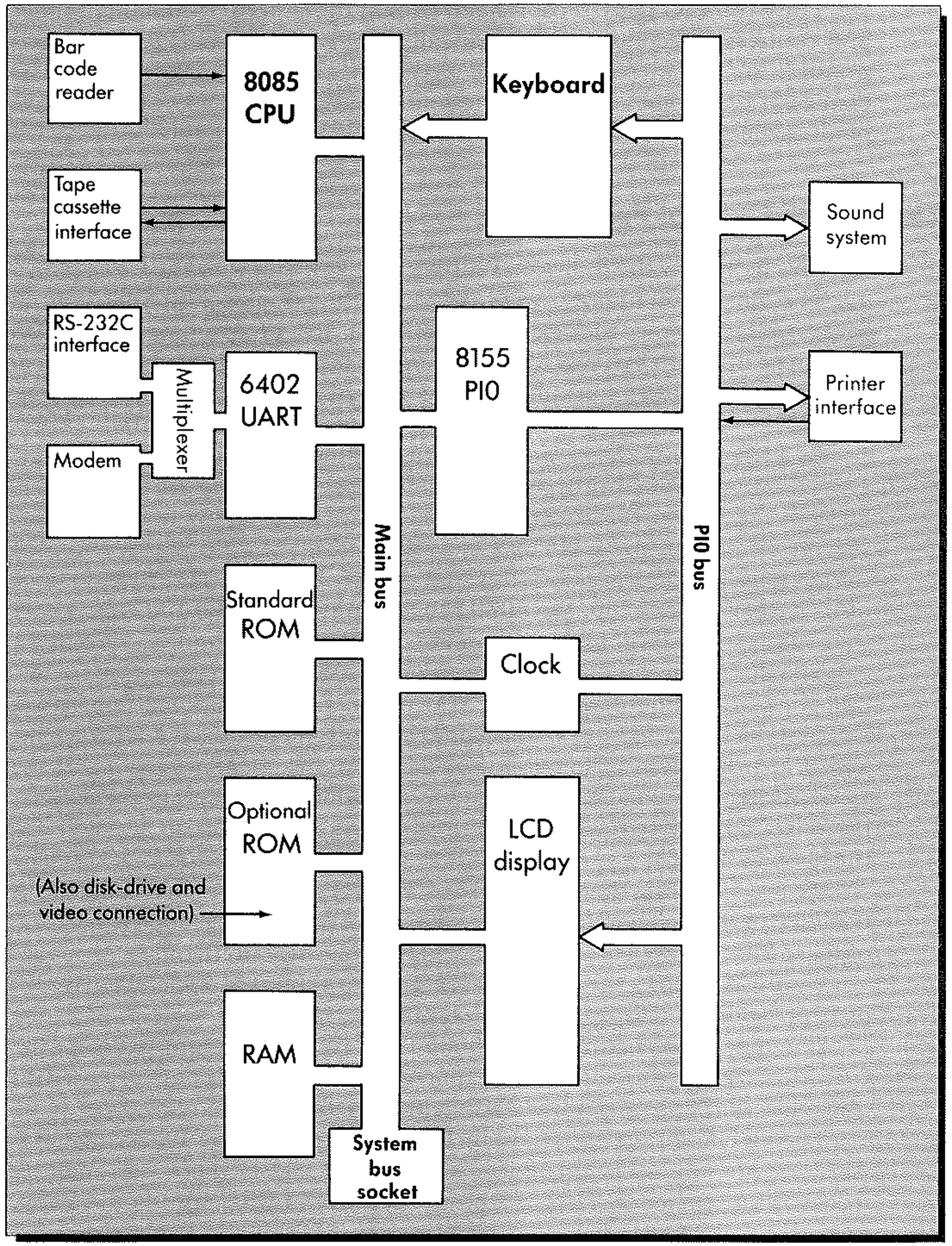


Figure 2-1. Model 100 block diagram

The 8155 PIO is the second main chip in the system. It contains a timer and controls a second bus that we call the PIO bus. The PIO bus feeds multiple signals in parallel to the keyboard, printer interface, and LCD display. It feeds other signals to the sound system and clock. Thus, control of all of these devices is channeled through the 8155 PIO.

The 80C85 Central Processor

Now let's study the individual subsystems, starting with the CPU.

The Model 100 uses a CMOS version of the popular 8-bit 8085 central processor chip (hence the designation 80C85). To a programmer the CMOS version of this chip behaves identically to the regular version. The only real difference is in power consumption. Because of its low power consumption, the CMOS version is able to run on battery power for many hours.

As a rule, low-power devices run slower than their high-power counterparts. However, much progress is being made in this regard. The CMOS 8085 in the Model 100 is driven at a clock speed of 2,457,600 cycles per second. Although this is not as fast as most of the high-power versions of the current crop of processors, it is significantly faster than the first versions of its earlier cousin chip, the 8080. As a result, the Model 100 runs faster than many of the larger microcomputers did a few years ago.

The 8085 chip differs from the older 8080 CPU chip in several important ways, including its RIM and SIM instructions, additional interrupts, internal clock circuits, and simpler power requirements. Except for the RIM and SIM instructions, the 8085 uses the same machine language and the same assembly-language mnemonics as the 8080. This overlap means that there is a large base of information, software, and expertise available for the 8085 CPU.

We are assuming that you already have access to books and manuals on the 8080 or 8085. Therefore, this book will not present a detailed discussion of these 8-bit microprocessor chips and how to program them.

Figure 2-2 shows the 8085 microprocessor's internal structure. We will discuss the major features of this diagram.

The internal structure of the 8085 CPU consists of eight major parts: CPU registers, Arithmetic Logic Unit (ALU), timing and control, interrupt control, serial control, bus interfacing, and instruction decoding.

The CPU registers each have their own "personalities". We assume that you have already been introduced to and understand these personalities from prior experience. Here is a quick overview of what you should know. Registers A, B, C, D, E, H, L, and Flags are all 8-bit memory cells within the CPU that can be combined into the following 16-bit register pairs: A/Flags, BC, DE, and HL. Some CPU instructions work with registers as single 8-bit temporary memory cells, while others work with these register pairs as 16-bit temporary memory cells. The A register is used as an "accumulator" for data; the HL register pair is used as a "pointer" into memory; the SP is the Stack Pointer for pointing to the stack for storing data and return addresses; and the PC is the Program Counter for pointing to the individual bytes of the machine language for the CPU.

The Arithmetic Logic Unit (ALU) performs the 8-bit arithmetic and logic operations such as addition, subtraction, complement, AND, and OR. It also performs shifting operations.

The timing and control section keeps the 8085 CPU humming along and also provides timing and control signals for the rest of the computer. The basic timing frequency is derived from an external crystal. The input from the external crystal comes into the timing and control section, where

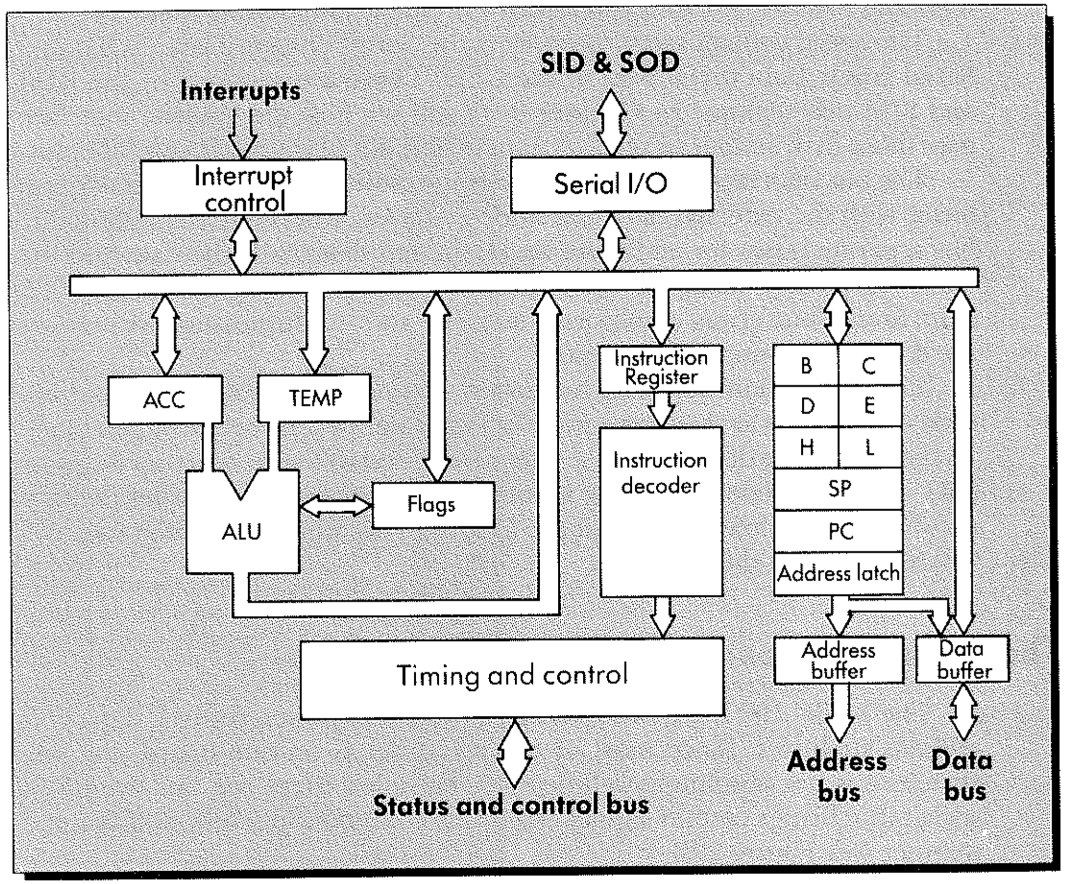


Figure 2-2. 8085 microprocessor (internal structure)

it is used to derive the basic timing signal for the operation of the microprocessor chip and the rest of the computer. For the Model 100, the crystal oscillates at 4,915,200 cycles per second. The timing and control circuits divide the crystal frequency by 2 to give the clock signal for the system. The same signal is sent out of the CPU to control memory and I/O access.

The interrupt control section interfaces the CPU to interrupt signal lines coming from various devices in the computer. These interrupt lines are called TRAP, RST 5.5, RST 6.5, and RST 7.5. In Chapter 3 we will see how these lines are serviced by software routines. For now, you need to understand only that if interrupts are enabled, a pulse on the TRAP line causes the CPU to automatically branch to location 24h = 36d as though a CALL 24h were suddenly executed. The RST 5.5, RST 6.5, and RST 7.5 lines operate in a similar manner, branching to locations 2Ch = 44d, 34h = 52d, and 3Ch = 60d respectively. The three RST interrupts behave somewhat differently from the TRAP interrupt, however, in that the 8085 RIM and SIM instructions can selectively enable and disable them.

The serial control section connects the Serial Output Data (SOD) and Serial Input Data (SID) pins to the CPU. These are serviced by the RIM and SIM instructions. In Chapter 9 we will see how the cassette interface uses these instructions to send its data in and out of the SOD and SID pins.

The bus interfacing section connects the main internal data and address buses with the external data and address buses. Internally the 8085 CPU has separate buses for addresses and data, but externally the data and the lower eight bits are shared by the same eight pins coming out of the 8085 chip. Some "buffering" (temporary storage) and "multiplexing" (switching) are done by the interfacing section.

The instruction decoding section is responsible for reading machine code instructions as they come into the computer byte by byte from memory. As they are being decoded, these bytes are stored in the Instruction Register (IR). The instruction decoding section analyzes these bytes, using the various bit fields to determine what the CPU is to do, what data it is to use, and where it is to put the results.

The System's Buses

The Model 100 has a bus system that includes power, control, data, and address subbuses. Since most of the computer is on one board, the bus structure is not absolutely well defined. However, there is a bus extension socket that brings certain signals out of the computer in a very well-defined manner (see Figure 2-3).

Power

The power bus carries the power needed to run the electrical components in the system. There are four separate lines that help conduct power: a ground line (GND) at 0 volts, a +5 volt line (VDD), a −5 volt line (VEE), and a line that carries voltage from the ni-cad battery (VB).

The voltage for the VDD and VEE lines is supplied by the four AA batteries or the AC adapter if it is used. The regular power switch on the right side of the Model 100 controls the power to these lines. The VDD line supplies the power for most of the chips in the Model 100. The VDD and VEE lines work together in certain circuits to produce voltages greater than five volts as needed. For example, the RS-232, MODEM, and LCD circuits use both the VDD and VEE lines.

As mentioned before, the voltage for the VB line is supplied by the rechargeable ni-cad battery. The memory power switch on the bottom of the Model 100 controls the power to this line. The VB line supplies power to the Model 100's RAM chips and to the real-time clock chip. It keeps the clock running and allows the contents of the RAM to be retained when the main power is shut off.

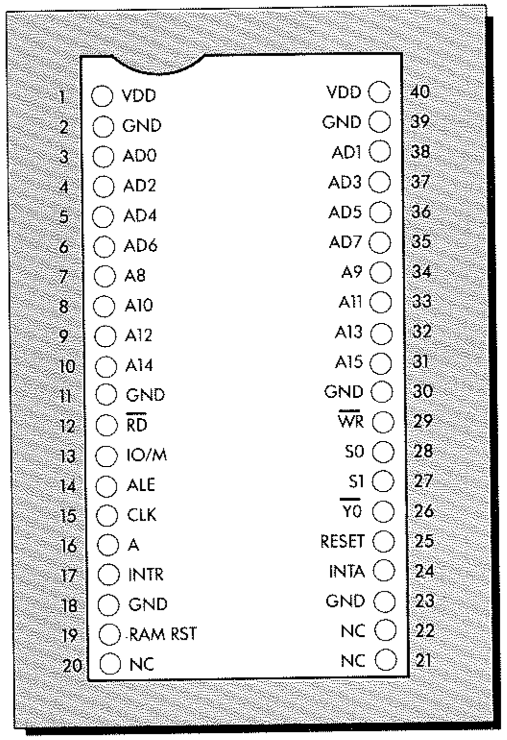


Figure 2-3. Bus extension socket

Of these power lines, only the ground and VDD lines appear on the bus extension socket.

Control

The control subbus contains a variety of signal lines, including read and write command lines for memory and I/O, interrupt control, timing signals, status, and reset control. We will not discuss these lines in much detail, since they are not of great concern to a programmer as long as they do their job properly.

Of the system's control lines, the following appear on the extension socket: RD, IO/M, ALE, CLK, A, INTR, RAM RST, WR, S0, S1, Y0, RESET, and INTA. All but the A, RAM RST, and Y0 are standard signals of the 8085 CPU. The A signal indicates when either reading or writing is happening and is used by external RAM, if present. The RAM RST signal is used to enable and disable any such external RAM. The Y0 signal is used to select an optional I/O controller unit. Y0 is generated when ports 80h = 128d through 8Fh = 143d are selected.

Data

There are eight data lines in the data subbus. They are labeled D0 through D7. On the 8085 CPU, the data lines share the same pins with the lower eight address lines (they are therefore labeled ADO through AD7 on Figure 2-3). On the main circuit board, the data lines are separated from the address lines.

The eight data lines appear on the bus extension socket.

Address

There are sixteen address lines in the address subbus. They are labeled A0 through A15. These lines are separated from the data lines on the main circuit board. All sixteen address lines appear on the bus extension socket.

ROM and RAM

The ROM is implemented by one chip that is called by its model number: LH-535618. This chip has fifteen address lines, three control lines, and eight data lines.

The fifteen address lines are connected to the lower fifteen lines of the address bus. The sixteenth address line of the address bus is routed to one of the control lines (Chip Select) of the ROM (see Figure 2-4). This ensures that the memory cells in this chip occupy the lower 32K bytes of the memory addressing space of the machine.

The data lines connect to the data bus, and the remaining two control lines control the timing of address selection and data output.

The RAM is implemented by four chip packs, each containing 8K bytes (see Figure 2-5). At least one of these chips must be installed for proper operation of the computer, for the various built-in ROM programs need some scratch storage. With just one chip pack installed, you have an 8K RAM machine. By installing more chips you can have a 16K, 24K, or 32K RAM machine. The chips are installed in high to low order in the addressing space of the 8085 CPU.

The RAM chip packs each contain four TC55188F-25 chips. Each of these chips has eight data lines, three control lines, and eleven address lines. This means that each chip contains two to the eleventh, or 2K, bytes of RAM. The Chip Enable control lines are used to determine which chip is

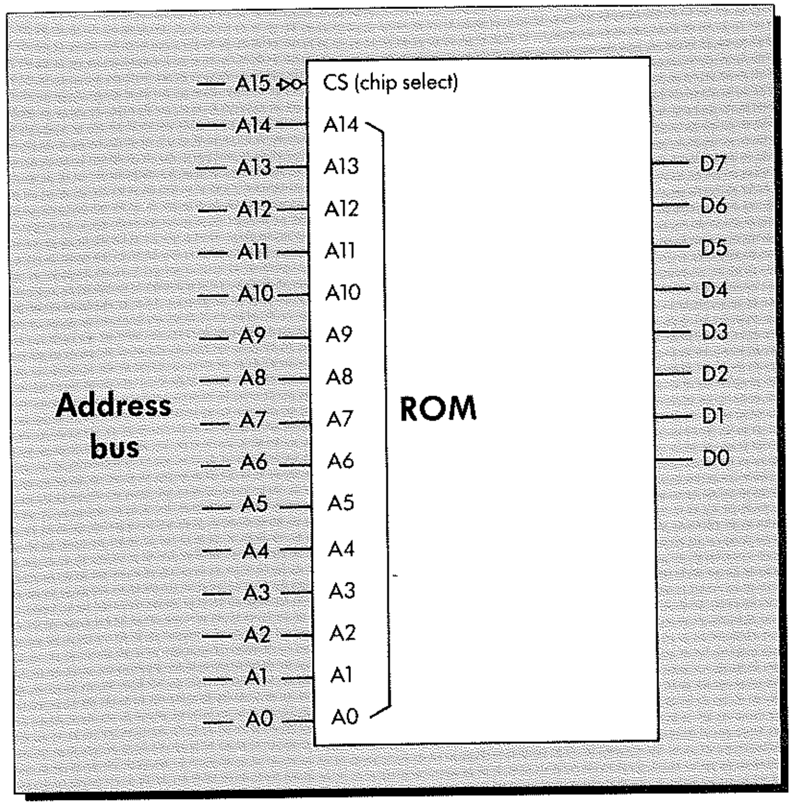


Figure 2-4. Addressing the ROM

selected for a particular memory address. Logic on the main circuit board converts address information from the address bus into the individual chip enable signals. In particular, bit 15 of the address bus determines whether or not the RAM is enabled at all, and bits 11 through 14 are decoded to determine which of the possible sixteen (four packs of four) chips is selected (see Figure 2-6).

Port Decoding

The 8085 CPU belongs to a family of processors that has two separate addressing spaces, one for memory and one for I/O. For the 8085 CPU, the memory addressing space has 64K bytes and the I/O addressing space has 256 bytes. In the former case, sixteen bits are used to generate addresses, and in the latter case only eight bits are used. For the I/O space, the lower eight lines of the address bus are used. A control signal in the control bus determines whether the CPU is trying to access a memory address or an I/O address. The programmer controls the CPU in this regard by selecting the appropriate machine-language instruction. IN and OUT instructions access the I/O space, whereas MOV, LDA, STA, LHLD, and SHLD instructions access the memory space.

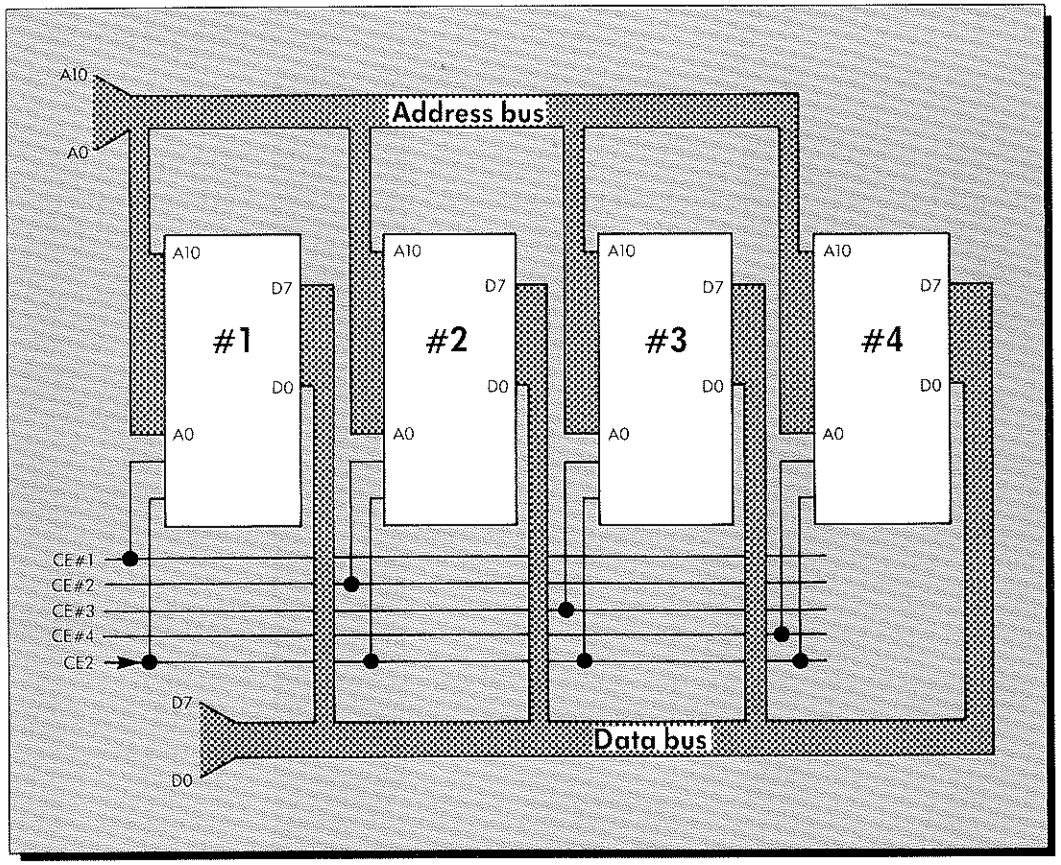


Figure 2-5. RAM chip pack

The Model 100 uses a *decoder chip* to divide up the I/O addressing space into ranges for various devices (see Figure 2-7). This decoder chip takes the upper four bits of the eight-bit I/O address and produces eight individual control lines, Y0 through Y7. These control lines are then used to selectively enable the various devices in the Model 100.

Let's look at the decoding process in more detail. As this decoder chip is configured, it has four input lines (consisting of three selection lines and one enable line) and eight output lines (Y0 through Y7). If the enable line is zero, then all outputs are zero. In this case, we say that the decoder is disabled. However, if the enable line is equal to one, then the chip converts a three-bit selection code placed on the selection lines into a pattern of signals on the output lines, turning "on" the output line corresponding to

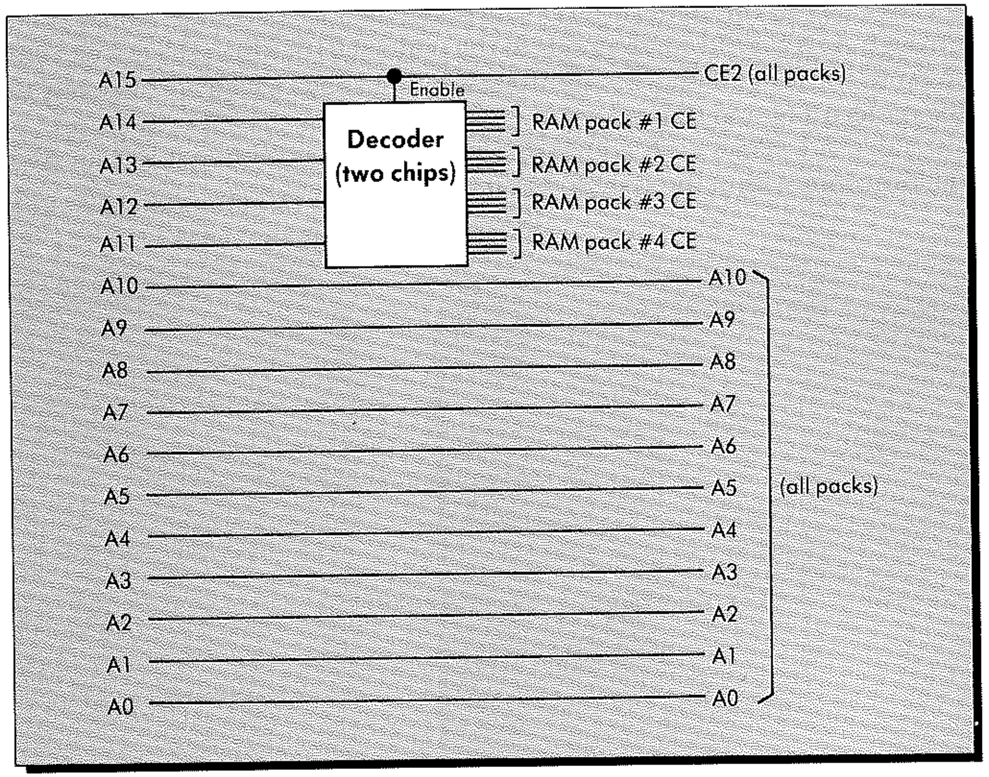


Figure 2-6. RAM addressing

the selection code and turning "off" all the rest. For example, if the three selection lines form the binary pattern 000, then Y0 is selected (has the value one) and the rest of the lines have value zero. If the three selection lines form the binary pattern 001, then Y1 is selected and the others are zero; and so on.

The port address lines A4, A5, and A6 are connected to the three selection lines, and the port address line A7 is connected to the enable line. Thus, if A7 is zero, all of the output lines Y0 through Y7 are zero; that is, none of the outputs is selected. This happens if the address is less than 80h = 128d.

If the address is greater than or equal to 80h = 128d, then the address line A7 has value one, enabling the decoder. In this case, the address bits A4, A5, and A6 determine which of the output lines Y0 through Y7 is selected.

The addresses from 80h to FFh divide into ranges depending upon the values for bit positions A4 through A6 of the address. For example, the range 80h = 128d through 8Fh = 143d corresponds to the pattern 000 for these bits, which selects Y0; the range 90h = 144d through 9Fh = 159d corresponds to the pattern 001, which selects Y1; the range A0h = 160d through AFh = 175d corresponds to the pattern 010, which selects Y2; and so on to the range F0h = 255d through FFh = 255d, which corresponds to the pattern 111 and thus selects Y7.

Table 2-1 shows each addressing range, the control line it selects, and the functions within the system that it enables.

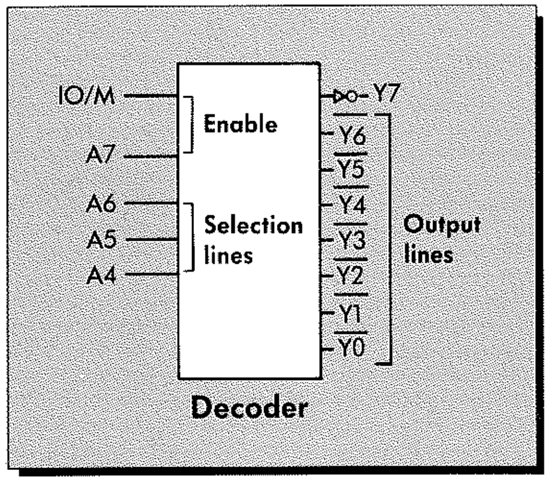


Figure 2-7. The port decoder

8155 Parallel Input/Output Interface Controller

The 8155 in the Model 100 is a CMOS version of the very powerful and flexible 8155 multifunction chip. This chip has 22 parallel I/O data lines, a timer, and 256 bytes of RAM (see Figure 2-8 for this chip's internal structure). The Model 100 uses all the data lines and the timer but does not use the RAM.

The 22 I/O data lines of the 8155 are divided into three I/O ports with eight lines in the first port (port A), eight in the second (port B), and six in the third (port C). We shall see in subsequent chapters how these ports are shared, in a very thrifty manner, by most of the input and output subsystems of the Model 100. In particular, the keyboard, real time clock, LCD screen, and printer all use some of these I/O data lines.

On the Model 100, the ports on the 8155 PIO chip are assigned the port addresses B0h = 176d through BFh = 191d. This is because the 8155 chip is enabled via the Y3 signal line from the main port decoder.

There are six ports used by the 8155 PIO. Each port appears twice within the sixteen-port range assigned to the 8155. This is because one bit (bit 3) of the port address is ignored.

Port 0 (at port addresses B0h = 176d and B8h = 184d) is used for control and status. When the port is written to, it is used for control. When it is read from, it is used for status.

When port 0 is used for control, bits 7 and 6 are used to set the timer mode as follows: 00 means no operation, 01 means stop the timer, 10 means stop after the count has expired, and 11 means start the timer. Bit 5 is used

Port Addressing Range	Functions
80h = 128d to 8Fh = 143d	Optional I/O controller
90h = 144d to 9Fh = 159d	Optional telephone answering unit
A0h = 160d to AFh = 175d	Output bit 0: on/off relay for telephone Output bit 1: enable Modem chip
B0h = 176d to BFh = 191d	8155 PIO
C0h = 192d to CFh = 207d	Data in/out for UART
D0h = 208dto DFh = 223d	UART program and read status
E0h = 224d to EFh = 239d	Output bit 0: select optional ROM Output bit 1: printer strobe Output bit 2: clock strobe Output bit 3: cassette motor on/off Input: keyboard matrix
F0h = 240d to FFh = 255d	LCD program, read, and write

Table 2-1. Port addressing ranges and functions

to enable or disable interrupts for port B; a value of 0 means disable. Bit 4 is used to enable or disable interrupts for port A. Bits 3 and 2 are used to define how the lines in port C are to be used as follows: 00 makes all six lines into input lines, 01 makes all six lines into output lines, 10 makes half the lines into output lines and half into control and status lines for port A, and 11 makes half the lines into control and status lines for port A and half the lines into control and status lines for port B. Bit 1 is used to control the direction of the data lines in port B, and bit 0 is used to control the direction of the data lines in port A. A value of 0 means that the lines are used for input, and a value of 1 means that the lines are used for output.

For the Model 100, bits 7 and 6 are changed to control the timer, but bits 5 through 0 always stay the same. These last six bits form a six-bit binary number, 000011b. This number specifies that all interrupts from ports A and B are disabled, ports A and B are used for output, and all six lines of port C are used for input. On the Model 100, ports A and B are used to send control signals to various devices in the computer, while port C is used to monitor the status of various devices.

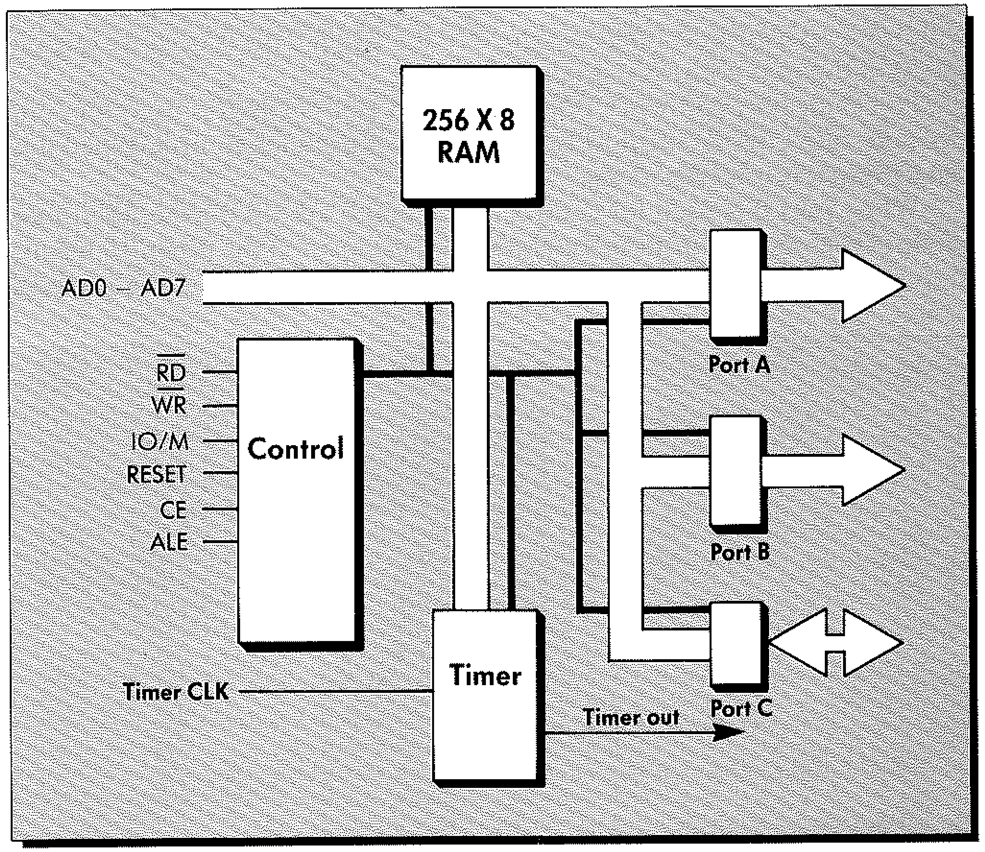


Figure 2-8. 8155 P10 (internal structure)

When port 0 is used for status, bit 7 is not used, bit 6 tells if the timer is running, and the other bits are used to monitor the interrupt and full/empty status of ports A and B. The Model 100 never reads port 0 (addresses B0h = 176d and B8h = 184d) and therefore never uses this status port.

Port 1 (addresses B1h = 177d and B9h = 185d) connects directly to the eight data lines of port A. Port 2 (addresses B2h = 178d and BAh = 186d) connects directly to the eight data lines of port B. Port 3 (addresses B3h = 179d and BBh = 187d) connects directly to port C.

Port 4 (addresses B4h = 180d and BCh = 188d) contains the lower eight bits of the count for initializing the timer. Port 5 (addresses B5h = 181d and BDh = 189d) contains the upper six bits of the count for the timer and two bits that are used to set the timer mode. See Chapter 8 on sound for more details on setting the timer.

Special Control Port

In addition to the control and status lines maintained by the 8155 PIO, there is a special control port that appears at addresses E0h = 224h through EFh = 239d (see Figure 2-9).

For this port, bit 0 is used to select an optional ROM, bit 1 is used to "strobe" data to the printer, bit 2 is used to "strobe" the real time clock, and bit 3 is used to control the remote motor control for the cassette recorder/player.

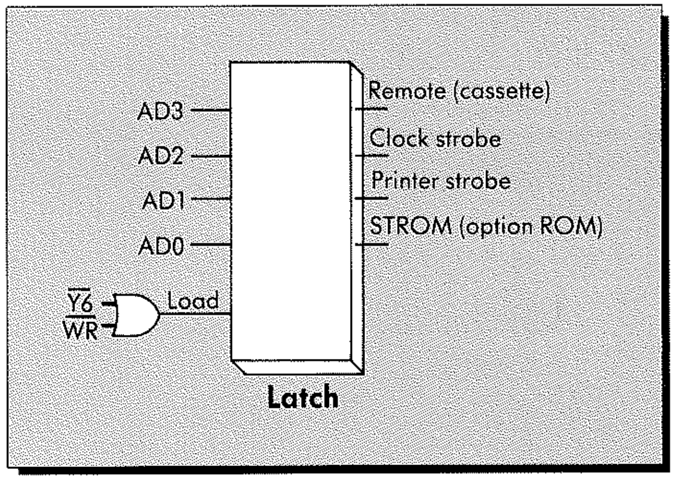


Figure 2-9. Special control port

The term "strobe" means that the signal line is used to send a pulse that locks the data being sent into the device to which it is being sent (printer, clock, or whatever). The strobe pulse is given once the data has been placed on the data lines and has settled down to a valid set of values.

µPD 1990AC Real Time Clock

The real time clock will be discussed in Chapter 5. We include its internal structure here (see Figure 2-10).

6402 Universal Asynchronous Receiver Transmitter

The 6402 UART (Universal Asynchronous Receiver Transmitter) will be discussed in Chapter 7. In that chapter we will see how the entire serial communications subsystem works. We include its internal structure here (see Figure 2-11).

Liquid Crystal Display Screen

The Liquid Crystal Display Screen will be discussed in Chapter 4. In that chapter we will see how it works and how it connects to the 8155 I/O data lines and to the main address and data buses.

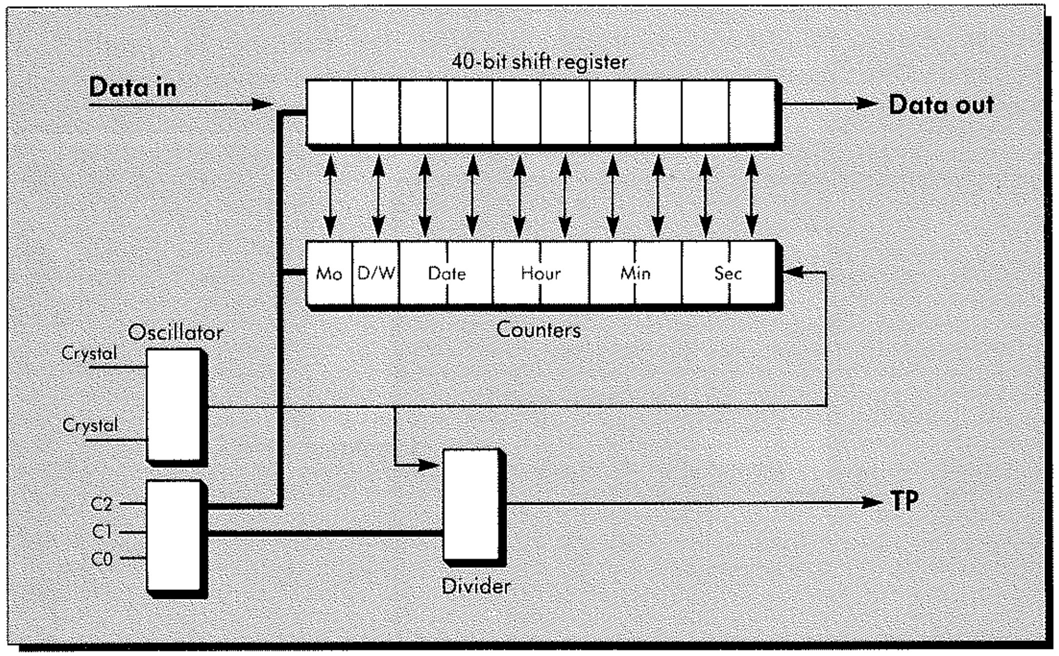


Figure 2-10. Real time clock (internal structure)

Keyboard

The keyboard will be discussed in Chapter 6. In that chapter we will see that the keys are laid out in a matrix that can be read using the data I/O lines from ports A and B of the 8155 and a special input port with addresses E0h = 224d through EFh = 239d.

Printer Interface

Let's finish this chapter with a quick discussion of the printer interface, since it will not be discussed later.

The printer interface uses port A of the 8155 PIO, the special control port (addresses E0h = 224h through EFh = 239d), and bits 1 and 2 of port C of the 8155 PIO chip (see Figure 2-12).

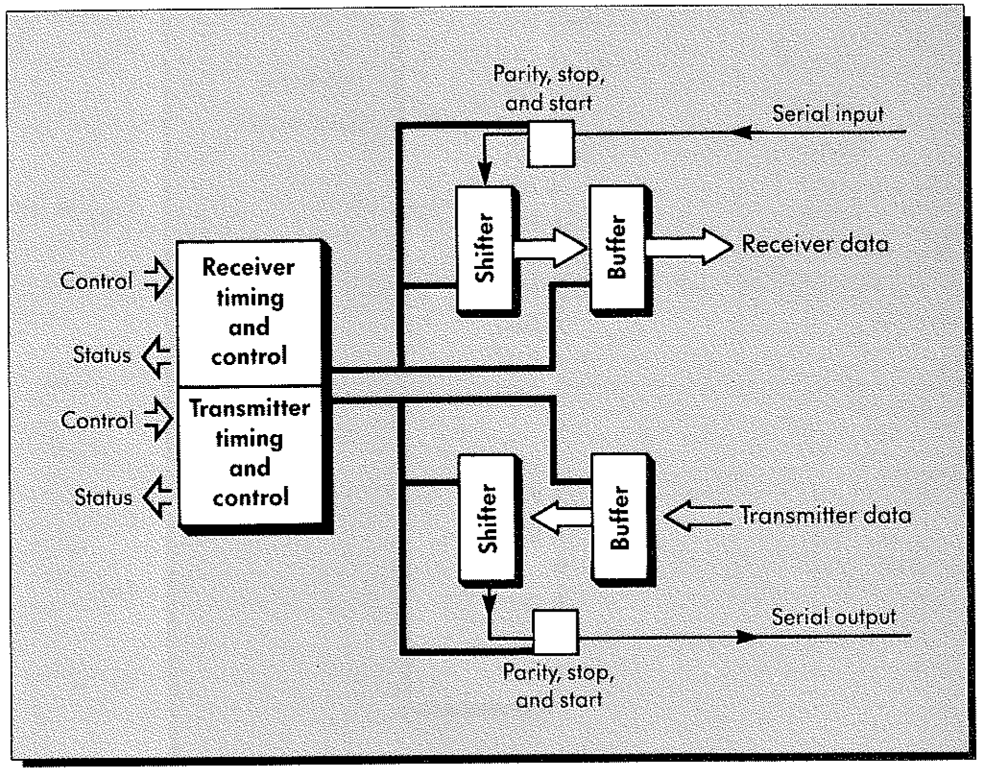


Figure 2-11. 6402 UART (internal structure)

Bits 1 and 2 of port C of the 8155 carry the busy/ready signal (noninverted and inverted) from the printer. To send data once the printer is ready, you send the data out port A of the 8155 and then change bit 1 of the special control port from a value of zero to a value of one and finally back to zero. This last action "strobes" the data to the printer. As we mentioned previously, a "strobe" is a control signal that is used to load data into a hardware buffer. In this case, the hardware buffer is in the printer.

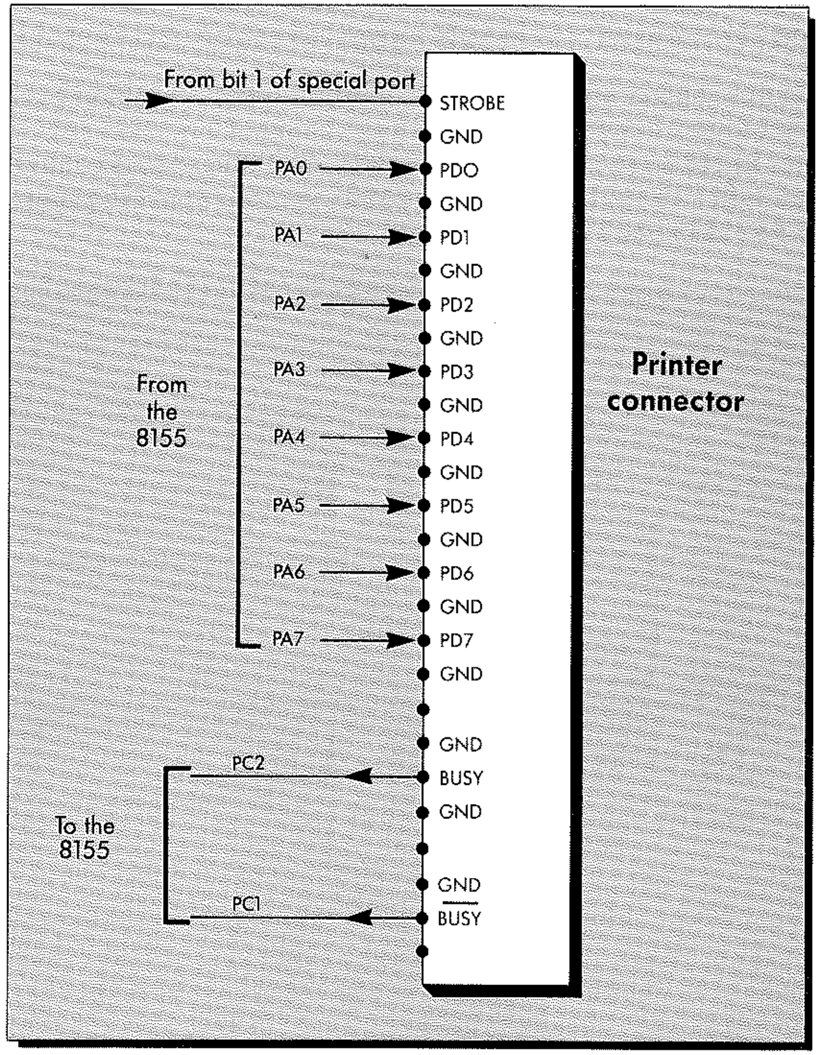


Figure 2-12. Printer interface

Summary

In this chapter we have explored the major hardware features of the Model 100. We have seen that it uses a CMOS version of the 8085 CPU, a member of a very popular family of chips. We have seen how the ROM, RAM, and ports are constructed and controlled, we have explored the multifunction 8155 chip, and we have quickly surveyed various subsystems such as the LCD, keyboard, and real time clock. Many of these subsystems have chapters devoted to them later in the book.

3

Hidden Powers of the ROM

Concepts

- Overall organization of ROM
- Interrupt entry points
- BASIC
- TELCOM
- MENU
- ADDRSS
- SCHEDL
- TEXT
- Cold and warm starts

In this chapter we will explore the secrets of the Model 100's ROM. We will study the overall organization of the ROM and then look at specific routines and tables that help run the BASIC, TELCOM, ADDRSS, SCHDL, TEXT, and MENU programs. The routines we study here will be those that perform general management tasks rather than controlling specific devices. The remaining chapters of this book will cover specific devices and their associated control routines.

There are 32 kilobytes of ROM in the Model 100. In these bytes are hundreds of useful routines that make the Model 100 a very powerful computer. We will start from the beginning (address 0) and work our way to the top of this ROM (address 7FFFh = 32767d) (see Figure 3-1).

At the lowest addresses of memory we find interrupt entry points. These are a series of hardware and software entry points to perform fundamental I/O and syntactical tasks.

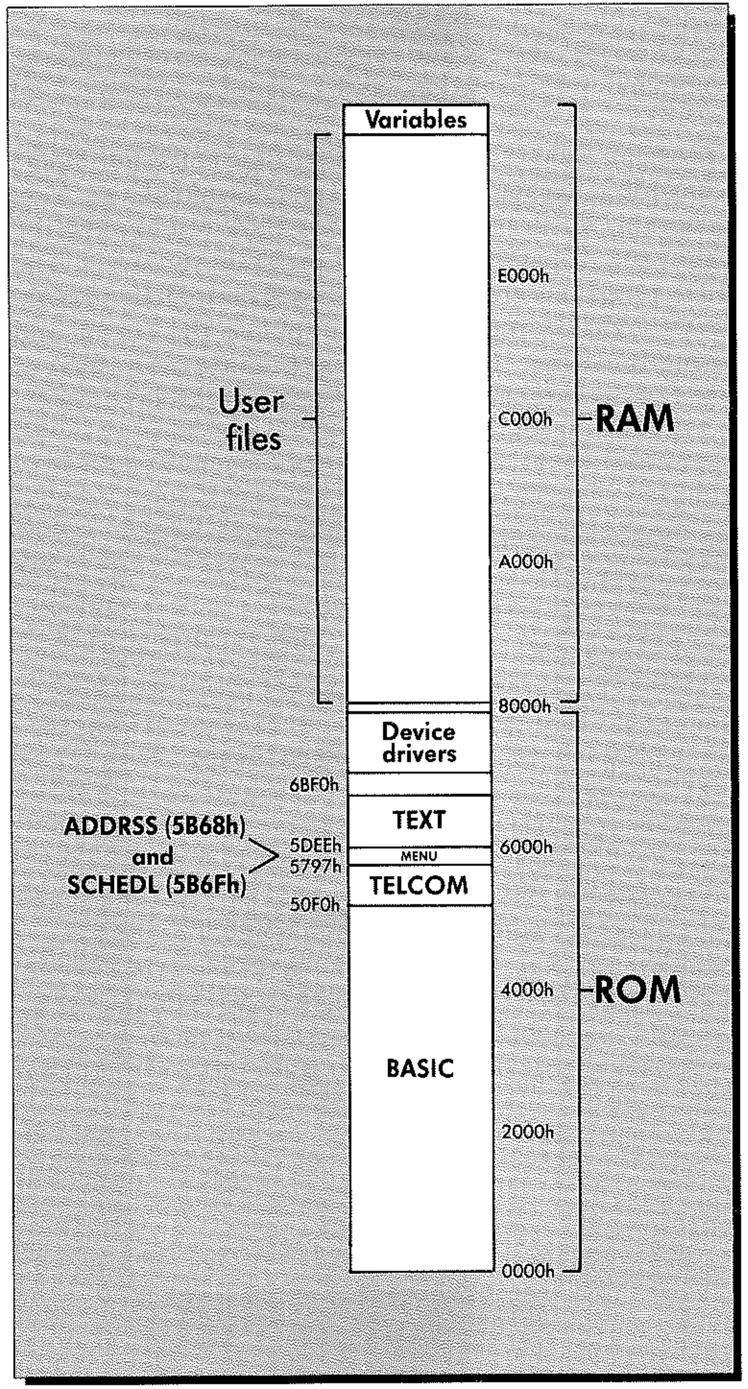


Figure 3-1. Layout of the Model 100's ROM

Next comes the BASIC interpreter. BASIC extends through a large area of ROM and includes areas for address and symbol tables, error handling, command input, command interpretation, and execution of individual commands.

The TELCOM program follows. It has routines that make the Model 100 into a terminal, including the ability to upload and download files. Next comes the MENU program, which has routines that allow you to move the cursor around the menu and dispatch to whatever program you select. The ADDRSS and SCHEDL programs are next. These programs blend into the TEXT program, which contains routines to manipulate the cursor and perform various editing functions. The final areas (highest addresses) of ROM contain initialization routines and primitive device control routines.

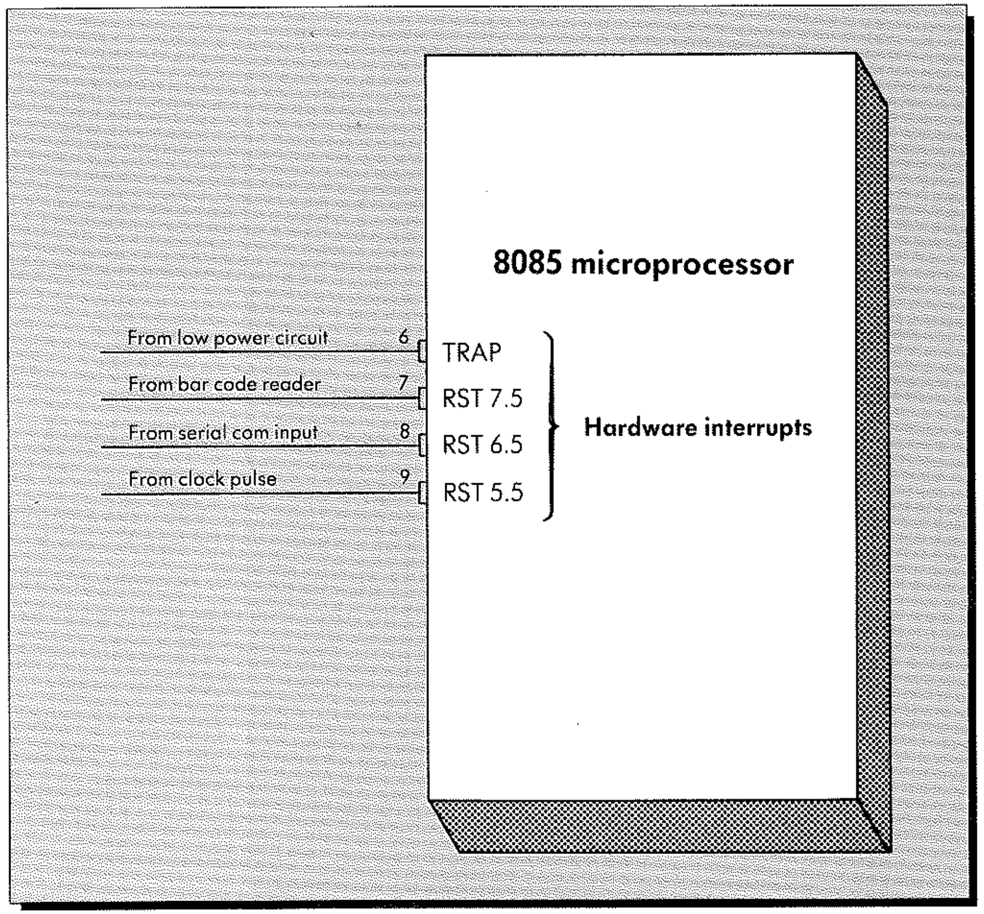


Figure 3-2. The hardware interrupt signal lines

We have strictly followed the physical organization of the ROM memory in this chapter in order to make the chapter easy to use as a reference guide to the ROM. However, this leads to a "bottom up" approach, particularly in studying BASIC. When you first read this chapter, you may want to start with the section on running BASIC programs and then go back and scan through the earlier sections as you need them.

The Interrupt Entry Points

The very lowest locations of ROM contain entry points for the 8085 CPU's software and hardware interrupt routines. The Model 100 uses all twelve of these special entry points. Eight of these are "called" by the one-byte RST (ReStarT) instructions. The other four are activated by special interrupt signal lines coming into the CPU from the low power detection circuit, the bar code reader interface, the serial communications line, and the real-time clock (see Figure 3-2).

Software Interrupts

Let's start with the software interrupts. These are eight locations with addresses that are even multiples of eight. Each one is called by one of the RST instructions. The Model 100 has placed special routines in these RST locations that in effect extend the instruction set of its 8085 CPU. For example, RST 3 compares the DE and HL register pairs (which would have been a handy CPU instruction if Intel had included it in the design of the 8085 CPU).

The RST 0 entry point is at location 0 and can be considered both a software and a hardware entry point. It is activated by the RST 0 instruction (software) and the RESET switch (hardware) on the back of the computer. The code at location 0 is a jump to location 7D33h = 32,051d, where there is a procedure to restart the Model 100 after it gets hung up.

Routine: RST 0

Purpose: To restart the Model 100 and return to main menu

Entry Point: 0h = 0d

Input: None

Output: The machine is reset.

BASIC Example:

```
   CALL 0
```

Special Comments: None

The RST 1 entry point is located at 8h = 8d. Here there is a routine that is useful in analyzing the syntax of BASIC commands. This routine compares the next byte from the BASIC command line with the byte immediately following the RST 1. If the two bytes agree, it returns, continuing execution right after the byte following the RST 1 instruction. If the two disagree, it jumps to a routine that declares a syntax error and halts execution of the BASIC program.

Routine: RST 1

Purpose: To compare next byte of machine-language program with next byte of BASIC command line

Entry Point: 8h = 8d

Input: Upon entry, the HL register pair points to a byte in memory.

Output: If the byte in memory matches the next byte after the RST 1 instruction, then the routine returns, continuing execution right after the byte following the RST 1 instruction. If the two bytes disagree, a syntax error is declared and BASIC returns to its input mode.

BASIC Example: Not directly applicable

Special Comments: None

The RST 2 entry point is at location 10h = 16d. This jumps to location 858h = 2,136d, where there is a routine that is also used in analyzing the syntax of BASIC commands. This routine examines the next byte of the command line. If that byte contains the ASCII code of a digit from 0 to 9, the routine returns with the carry flag set. If not, it skips over spaces, tabs, and linefeeds in the command line. It returns once it finds a byte that is not one of these. If it finds a byte that is zero, it sets the Z flag. This usually means "end of command line".

Routine: RST 2

Purpose: To search for next byte of command, checking for zero byte or decimal digit

Entry Point: 10h = 16d

Input: Upon entry, the HL register pair points to a byte position in a BASIC command line.

Output: The routine skips over spaces and tabs in the command line. It stops when it finds a byte that is not a space or a tab. It returns with the value of this byte in the A register. If this byte contains the ASCII code of a decimal digit (0 through 9), the routine returns with the carry flag set. If this byte is zero, it returns with the Z flag set.

BASIC Example: Not directly applicable

Special Comments: None

The RST 3 entry point is at location 18h = 24d. Here there is a routine that compares the contents of the DE with the contents of the HL register. If they are equal, it returns with the Z flag set. If HL is less than DE, then the carry is set. Otherwise it is clear.

Routine: RST 3

Purpose: To compare two 16-bit integers

Entry Point: 18h = 24d

Input: Upon entry, the register pairs HL and DE each contain 16-bit integers.

Output: When the routine returns, the carry and zero flags are set as follows:

Condition	Flags
HL < DE	C and NZ
HL = DE	NC and Z
HL > DE	NC and NZ

BASIC Example: Not applicable

Special Comments: None

The RST 4 entry point is at location 20h = 32d. This jumps to location 4B44h = 19,268d, where there is a routine to print a character on the LCD screen (see Chapter 4).

Routine: RST 4

Purpose: To print a character on the LCD screen

Entry Point: 20h = 32d

Input: Upon entry, the A register contains the ASCII code of the character to be printed.

Output: The character is printed on the screen.

BASIC Example:

```
CALL 32,X
```

where the value of X is the ASCII code of the character.

Special Comments: None

The RST 5 entry point is at location 28h = 40d. This jumps to location 1069h = 4201d, where there is a routine that checks the data type of the expression currently being processed by BASIC.

Routine: RST 5

Purpose: To check the data type of the expression currently being processed

Entry Point: 28h = 40d

Input: Upon entry, the data type of the expression must be in location FB65h = 64,357d.

Output: Upon exit, the flags indicate the data type according to the following rules:

Data Type	Flag Condition
Integer	Zero flag clear (NZ)
String	Sign flag clear (P)
Single-precision real	Carry flag set (C)
Double-precision real	Parity flag set (E)

BASIC Example: Not directly applicable

Special Comments: None

The RST 6 entry point is at location 30h = 48d. This jumps to location 33DCh = 13,276d, where there is a routine that checks the sign of the number currently being processed by BASIC.

Routine: RST 6

Purpose: To get the sign of the real expression currently being evaluated

Entry Point: 30h = 48d

Input: Upon entry, a valid single- or double-precision real number must be in BASIC's accumulator (locations FC18h = 64,536d through FC1Fh = 64,543d).

Output: The A register is used to return a result. If the real number in BASIC's accumulator is zero, the result in A is zero. If the real number is negative, this result is − 1. If the real number is positive, the result is + 1.

BASIC Example: Not directly applicable

Special Comments: None

The RST 7 entry point is at location 38h = 56d. This jumps to location 7FD6h = 32,726d, where there is a dispatch routine that uses a special table of addresses stored in RAM (see Figure 3-3). This table is called the "hook" table because it gives programmers a chance to insert their own routines in strategic places in the ROM programs. In Chapter 4 we will see an example of the way the RST 7 instruction is used in the ROM programs.

The RST 7 routine uses the byte following the RST 7 instruction to look up an address in a table starting at location FADAh = 64,218d in RAM. The routine calls this subroutine and returns to continue execution right after the byte following the RST 7 instruction. The first 57 addresses point to a subroutine consisting of just a RETurn instruction, and the last 37 locations each point to a routine that declares an illegal function error.

Routine: RST 7

Purpose: To dispatch to routine in hook table

Entry Point: 38h = 56d

Input: The byte following the RST 7 instruction is used as input and must be between 0 and 56.

Output: The routine dispatches to the location whose address is at location FADAh = 64,218d plus twice the value of the byte following the RST 7 instruction.

BASIC Example: Not directly applicable

Special Comments: None

Hardware Interrupts

The four hardware interrupt entry points are located at 24h = 36d, 2Ch = 44d, 34h = 52d, and 3Ch = 60d. In each case there is a signal line coming into the CPU that can be used to cause the CPU to interrupt its normal activity and immediately branch to one of these entry points. The routines pointed to by these entry points are the key to understanding how the corresponding devices work.

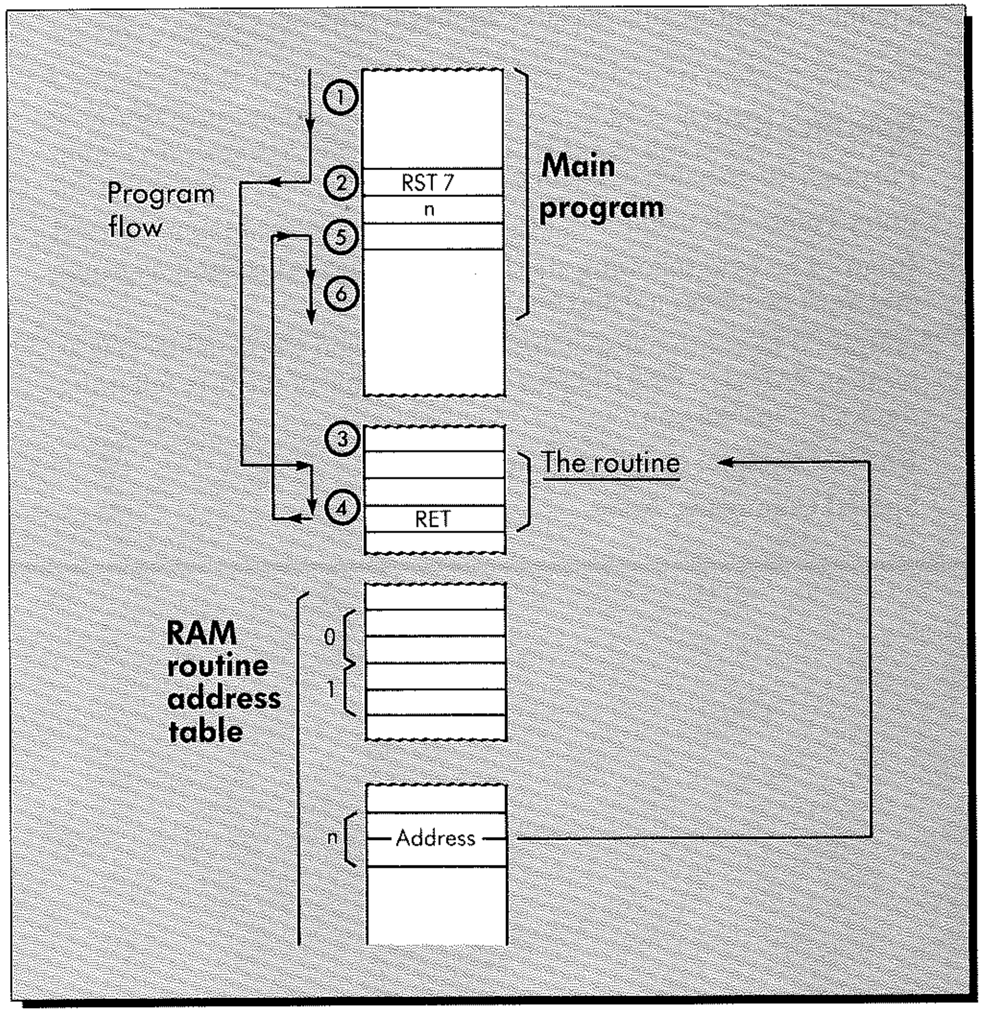


Figure 3-3. RAM routine table

The TRAP entry point is at location 24h = 36d. It is activated when the power circuit indicates a low-power condition. This entry point jumps to location F602h, where the low-power condition is handled.

Routine: TRAP

Purpose: To handle low-power condition

Entry Point: 24h = 36d

Input: None

Output: Shuts off the power.

BASIC Example:

```
CALL 36
```

Special Comments: To restore the power, turn the power switch (on right side of machine) off and then on again.

The RST 5.5 entry point is at location 2Ch = 44d. It is activated by the bar code reader. Here, interrupts are disabled with the DI instruction. Then there is a jump to location F5F9h = 62,969d, where the bar code reader interrupt service routine resides.

Routine: RST 5.5

Purpose: To receive data from bar code reader

Entry Point: 2Ch = 44d

Input: From the bar code reader

Output: To the Model 100

BASIC Example: Not directly applicable

Special Comments: Not implemented in the machine as it comes from the factory.

The RST 6.5 entry point is at location 34h = 52d. It is activated by input from the serial communications line. Here, interrupts are disabled with the DI instruction. Then there is a jump to 6DACh = 28,076d, where the interrupt service routine for accepting input from the serial communications line resides.

Routine: RST 6.5

Purpose: To input byte from serial communications line

Entry Point: 34h = 52d

Input: A byte from the serial communications line must be ready.

Output: To the serial communications input buffer

BASIC Example: Not applicable

Special Comments: None

The RST 7.5 entry point is at location 3Ch = 60d. It is activated by a pulse from the clock every 4 milliseconds. At this entry point, interrupts are disabled with the DI instruction. Then there is a jump to 1B32h = 6,962d, where the background task begins. We will discuss the background task in more detail in the next few chapters.

Routine: RST 7.5

Purpose: To initiate one cycle of the background task

Entry Point: 3Ch = 60d

Input: None

Output: Affects LCD display, real-time clock, and keyboard (see Chapters 4, 5, and 6).

BASIC Example: Not applicable

Special Comments: See chapters 4, 5, and 6 for a full discussion of the background task.

The BASIC Interpreter

The BASIC interpreter provides a friendly programming environment for the Model 100. BASIC is a popular language because it is powerful, has simple and direct syntax rules, and can immediately run programs or direct commands as they are entered or modified. In this section you will see exactly how all this works. We will also show you a simple program that makes BASIC even more powerful and friendly.

BASIC extends from about 40h = 64d to about 50F0h = 20,720d in the Model 100's ROM. However, many of the routines in this area are shared by other programs and BASIC uses some routines located in other areas.

BASIC Symbol and Address Tables

The address and symbol tables used by BASIC start at location 40h = 64d, right after the interrupt entry points. These tables are described briefly below. Refer to the appendices for their complete contents.

The first table (locations 40h = 64d through 7Fh = 127d) is a list of addresses of routines to handle various BASIC functions such as INT, ABS, SIN, and PEEK (see Appendix A). This table is used by the BASIC interpreter to help it find these routines.

The second table (locations 80h = 128d through 25Fh = 607d) contains all the BASIC keywords (see Appendix B). This table is used by the BASIC interpreter to convert its keywords into special one-byte codes called tokens. The tokens range in numerical value from 128 to 255. The first keyword in this table is assigned to the token value 128, the second is assigned to 129, and so on through the rest of the table.

The third table (locations 262h = 610d through 2E1h = 737d) contains the addresses of the routines to handle BASIC commands. These are given by the BASIC *initial* keywords — that is, keywords that appear at the beginning of a BASIC command line, such as FOR, LET, GOTO, and CLEAR (see Appendix C).

The fourth table (locations 2E2h = 738d through 2EDh = 749d) contains operator priorities for the binary operations: +, −, *, and so on (see Appendix D). These priorities allow BASIC to recognize which operations to do first in a complex algebraic expression. For example, in the expression A + B * C, the * operation should be performed first and the + second.

The fifth table (locations 2EEh = 750d through 2F7h = 759d) contains addresses of some numerical conversion routines (see Appendix E).

These convert numbers to double-precision, integer, and single-precision formats.

The sixth table (locations 2F8h = 760d through 303h = 771d) contains the addresses of the routines for binary operations of +, −, *, /, and comparison for double-precision floating-point numbers (see Appendix F).

The seventh table (locations 304h = 772d through 30Fh = 783d) contains the addresses of the routines for binary operations of +, −, *, /, and comparison for single-precision floating-point numbers (see Appendix G).

The eighth table (locations 310h = 784d through 31Bh = 795d) contains the addresses of the routines for binary operations of +, −, *, /, and comparison for integers (see Appendix H).

The ninth table (locations 31Ch = 796d through 359h = 857d) contains a list of all the two-character error designators (see Appendix I).

Area Mapped to High Memory

The contents of the next area of ROM (locations 35Ah = 858d through 3E9h = 1001d) are moved to RAM (locations F5F0h = 62,960d through F67Fh = 63,103d) when the Model 100 is first initialized. This area contains various variables and routines, including those that help control the keyboard, LCD screen, and BASIC itself. Their initial values are stored in ROM. When this area is mapped to RAM, these initial values are put in place.

Among the routines in this area are ones that are called at the beginning of certain interrupt service routines. When this area is moved to RAM, these routines consist of just a RETurn instruction followed by two NO oPeration instructions. However, because they are placed in RAM, they can be changed to JuMPs to your own interrupt routines.

Other routines in this area implement the INP and OUT commands (see boxes) and key steps of the line-drawing algorithm used in the LINE command (see Chapter 4).

Routine: INP — BASIC Command

Purpose: To input from port

Entry Point: F66Ah = 63,082d

Input: The address of the desired port must be in location F66Bh = 63,083d.

Output: The routine returns with the data from the port in the A register.

BASIC Example: Not directly applicable

Special Comments: When you use the INP function, BASIC automatically puts the port address in location F66Bh = 63,083d.

Routine: OUT — BASIC Command

Purpose: To output to a port

Entry Point: F667h = 63,079d

Input: Upon entry, the A register contains the data to be output, and location F668h = 63,080d contains the port address.

Output: The contents of the A register are sent out the port.

BASIC Example:

```
POKE 978,PORT
CALL 977,DATA
```

where PORT is the port address and DATA is the value of the data byte.

Special Comments: None

Messages and Errors

The next area of ROM (locations 3EAh = 1002d through 400h = 1024d) contains messages used by BASIC. These include "Error", "?", "Ok", and "Break".

A set of routines that handles various errors runs from about 422h = 1058d to 501h = 1281d.

The main entry point for handling errors is at location 45Dh = 1117d (see box). Upon entry, the E register must contain an error code. When this routine is called, it displays the corresponding error message and aborts execution of a BASIC program. The error codes are given on page 217 of the Model 100 owner's manual and in Appendix I.

Routine: Main BASIC Error Routine

Purpose: To display an error message

Entry Point: 45Dh = 1117d

Input: Upon input, the E register contains the error code.

Output: The routine displays the error message corresponding to the error code in the E register. See the error code table on page 217 of the owner's manual and Appendix I of this book.

BASIC Example: Not directly applicable

Special Comments: None

There are special error entry points in which the E register is loaded with a specific error code and then the main entry point processes the error. These special entry points are located in two areas of the ROM. The first area goes from 446h = 1094d to 45Ch = 1116d, and the second area goes from 504Eh = 20,558d to 506Ah = 20,586d (see Appendix J).

Command Entry

The main command entry loop for BASIC begins at location 501h = 1281d. We will discuss some of its highlights.

The code at location 50Bh = 1291d displays the message "Ok" on the screen. The message is stored at location 3F6h = 1014d, as discussed previously. The routine that displays a message is located at 27B1h = 10,161d. It expects the address of the message in the HL register pair.

At location 511h = 1297d, the value FFFFh (minus one in 2's complement arithmetic) is placed in the current BASIC line number F67Ah = 63,098d. This tells BASIC that it is in command entry or immediate mode rather than running a program.

Next the INLIN routine at 4644h = 17,988d is called (see box). This routine waits for the next line from the keyboard. If the line is nonempty, the first level of interpretation, called tokenizing, takes place.

Routine: INLIN

Purpose: To input a line from the keyboard

Entry Point: 4644h = 17,988d

Input: From the keyboard

Output: When the routine returns, the line is stored in memory starting at location F685h = 63,109d. The line is terminated in memory with a byte whose value is zero.

BASIC Example:

```
CALL 17988
```

Special Comments: You can pick up the line by using the PEEK statement.

During tokenizing, all the keywords in the BASIC command line are converted to their one-byte tokens (see BASIC keyword table — Appendix B). The command line is then in a very compact form, making it easier to run and to store.

The tokenizing routine is located at 646h = 1606d and is called from 54Ah = 1354d. The tokenizing routine has to be careful not to tokenize REMarks and quoted material, and it has to translate lowercase characters in variable names to uppercase.

Routine: Tokenize

Purpose: To tokenize a BASIC command line

Entry Point: 646h = 1606d

Input: Upon entry, the HL register pair points to (contains the address of) the untokenized line.

Output: When the routine returns, the tokenized line is stored in memory starting at location F681h = 63,105d. The tokenized line is terminated by a byte whose value is zero.

BASIC Example:

```
AØ=VARPTR(A$)
A1=PEEK(AØ+1)+256*PEEK(AØ+2)
CALL 16Ø6,Ø,A1
```

where A$ is a string containing the line to be tokenized, A0 is the address of the string's length and location parameters, and A1 is the address of the actual bytes of the string.

Special Comments: You can use PEEK to get the bytes of the tokenized string from where they are stored in memory.

The routine works by matching character strings in the command line against character strings in the token table at 80h = 128d (see Appendix B). When it is looking for a match, it sweeps through the table, counting all the mismatches until it finds a match. This count then is used to compute the value of the token.

After tokenizing the line, the BASIC interpreter tries to insert the newly created line into the program. It must first check whether the new line is an immediate command. This is done with the RST 2 instruction, which is invoked at 523h = 1315d, and the conditional jump at 552h = 1362d. If the line contains an immediate command, the command is executed immediately by jumping to location 4F1Ch = 20,252d.

If the line is to be inserted into the program, then the code from 555h = 1365d to 5ECh = 1516d is used to place it in the program. If a line with the same line number already exists in the program, the new line replaces the old; otherwise, it is simply inserted in the program. The routine to search for the line number in the program is located at 628h = 1576d (see box).

Routine: Search for Line Number

Purpose: To search a BASIC program for a line with a specified line number

Entry Point: 628h = 1576d

Input: Upon entry, the DE register pair has the line number in binary.

Output: When the routine returns, the HL register pair contains the address of the proper location to insert a line with the specified line number into the program. If the specified line number matches an existing line number in the program, the carry flag is clear; otherwise it is set.

BASIC Example: Not directly applicable

Special Comments: None

Running BASIC Programs

The next major area of memory contains the code that runs BASIC programs. This is the heart of the BASIC interpreter. This section extends from about 804h = 2052d to 871h = 2161d.

This code has to check several conditions as it keeps running your program. The first thing it checks is the communications line, by calling a routine located at 6D6Dh = 28,013d (see Chapter 7). Then it checks for an interrupt from the real-time clock by calling a routine at 4028h = 16,424d (see Chapter 5). Next it checks for a break from the keyboard by calling a routine at 13F3h = 5107d (see Chapter 6).

The BASIC interpreter then looks for a colon indicating the next statement of a multiple statement on a BASIC command line. Finally, it checks for the end of the program. It returns to the command loop via a jump to 428h = 1064d if the program has indeed ended.

If the program has not ended, interpretation continues. The current line number is stored in location F67Ah = 63,098d, and then the interpreter dispatches to the routine to handle the keyword for the BASIC command currently under consideration. It points to the address table for BASIC commands. It is interesting to note that the last part of the dispatching routine (locations 858h = 2136d through 871h = 2161d) uses the same code as is used by the RST 2 routine to check for numbers and skip over spaces.

BASIC Keyword Routines

The next area of ROM contains routines to handle the various BASIC keywords. This area extends from about 872h = 2162d to about 50F0h = 20,720d. It also includes some isolated sections of code before and after this. For example, the code for the FOR statement extends from 726h = 1830d to about 803h = 2051d, which is between the tokenizer routine and the command dispatcher.

The LET Statement

Among the most fundamental parts of the code for the Model 100 is the code for handling the LET command (see box). Each LET command statement consists of a variable name or identifier followed by an equal sign, followed by a BASIC expression. Executing the LET command involves locating variables and interpreting BASIC expressions. In a sense it is the key to understanding how BASIC works in the Model 100.

Routine: LET — BASIC Command

Purpose: To evaluate BASIC expressions and store the results in the indicated BASIC variables

Entry Point: 9C3h = 2499d

Input: Upon entry, the HL register pair points to (contains the address of) the rest of the BASIC LET command line (starting with the variable identifier on the left of the "=").

Output: When the routine returns, the value of the BASIC expression is stored in the indicated variable (on the left side of the "=" in the LET command line).

BASIC Example:

```
CALL 2499,0,63105
```

where the input buffer at F681h = 63,105d contains a tokenized BASIC LET command line starting with the name of the variable on the left side of the "=". Call the tokenizer routine at 646h = 1606d before using this example.

Special Comments: The input buffer at 63,105 is also used by the INPUT statement.

The code for the LET routine extends from 9C3h = 2499d to A2Eh = 2606d.

We have included a BASIC program that calls this LET routine. It also calls the tokenizer routine at 646h = 1606d. This program turns your Model 100 into a fancy calculator. It can also be modified to turn the Model 100 into an interactive function grapher.

When you run this program, you are asked for the formula of the function. You should enter it as a BASIC formula F(X) in the single variable X. Next you are asked to give a starting value, an ending value, and a step size for X. Once you have entered these, the program displays the values of X and F(X) for the range and step size that you selected.

```
100 ' INTERACTIVE FUNCTION EVALUATOR
110 '
120   CLEAR 100
130 '
140   PRINT "INPUT A BASIC FORMULA";
150   PRINT " IN X"
160   INPUT "F(X)=";B$
170   A$= "Y="+B$+CHR$(0)
180 '
190   INPUT "STARTING VALUE OF X";X0
200   INPUT "ENDING VALUE OF   X";X1
210   INPUT "STEP SIZE OF X     ";X2
220 '
230 ' TOKENIZE THE FORMULA
240   A0 = VARPTR(A$)
250   A1 = PEEK(A0+1)+256*PEEK(A0+2)
260   CALL 1606,0,A1
270 '
280 ' DISPLAY LOOP
290   FOR X = X0 TO X1 STEP X2
300    CALL 2499,0,63105
310    PRINT X,Y
320   NEXT X
```

Let's look at the program more closely. On lines 140-170, the formula is input into the string B$ and packed into the string A$. The string A$ includes a prefix of "Y=" to supply the variable to the left of the "=" as well as the "=". There is also a zero character packed into the end of A$ to terminate the string for the ROM routines that we will call.

On lines 190-210, the starting step, ending step, and step size are input as variables X0, X1, and X2. Lines 230-250 compute the address where the actual bytes of the command string are stored in memory. The tokenizer routine at 646h = 1606d is called in line 260. Here, HL points to the command string for our custom LET statement. The display loop for the values of X and F(X) runs from lines 280-320. It consists of a FOR loop indexed by the variable X with starting step, ending step, and step size determined by X0, X1, X2. Inside the FOR loop, the LET routine at 9C3h = 2499d is called to evaluate the formula Y = F(X), and then the values of X and Y are printed on the LCD screen.

To modify this program into an interactive function grapher, replace the PRINT command in the FOR loop by a line plotting command. Of course, you will have to add more controls to keep the screen looking good for the display.

Now let's return to our general discussion of the LET command and see how it is implemented in ROM.

The LET statement is the only BASIC command that doesn't require a keyword, although it also accepts lines that begin with the token for the keyword LET. As part of the command interpreter, BASIC must be able to detect commands that don't begin with a token. This function occurs at 840h = 2112d, just before dispatch to the individual routines for keywords. Here 80h = 128d is subtracted from the initial character on the line. If the result is negative, the character is not a token, and a jump is made to the code for LET.

Let's now look at the code for LET in more detail. It first calls a routine at 4790h = 18,320d, which returns the address of the variable on the left side of the equals sign (see box). Then it checks for the equals sign, aborting the program and declaring a syntax error if the equals sign is not present. It next calls a routine at DABh = 3499d to evaluate the right-hand side of the equals sign. Finally, it moves the computed value into the location reserved for the variable on the left of the equals sign.

Routine: Address Finder for BASIC Variables

Purpose: To locate the address of a BASIC variable

Entry Point: 4790h = 18,320d

Input: Upon entry, the HL register pair points to (contains the address of) the name of a variable.

Output: When the routine returns, the DE register pair points to the location of the variable in BASIC's table of variables. Information such as its data type and value is stored here.

BASIC Example: Not directly applicable

Special Comments: None

The address finder routine at 4790h = 18,320d expects the address of the name of the variable in the HL register pair, and it returns the address of the variable in the DE register pair.

The address finder routine is used by the VARPTR function as well as the LET command. VARPTR is a BASIC function that returns the address of a variable whose name is specified or a file whose number is specified. The code for the VARPTR function starts at F7Eh = 3966d. The part of the code that deals with the address of a variable begins at F92h = 3986d. It calls a routine at 482Ch = 18,476d that directly calls the address finder routine.

Routine: VARPTR — BASIC Function

Purpose: To return the location of a variable or a file

Entry Point: F7Eh = 3966d

Input: Upon input, the HL register pair points to (contains the address of) the name of a BASIC variable or a BASIC file number.

Output: When the routine returns, the address of the variable or the file buffer is in the DE register pair.

BASIC Example:

```
A = VARPTR(X)
```

where X is a BASIC variable and A is the BASIC variable where the address of X will be stored.

Special Comments: Cannot be called from BASIC because the result is returned in DE.

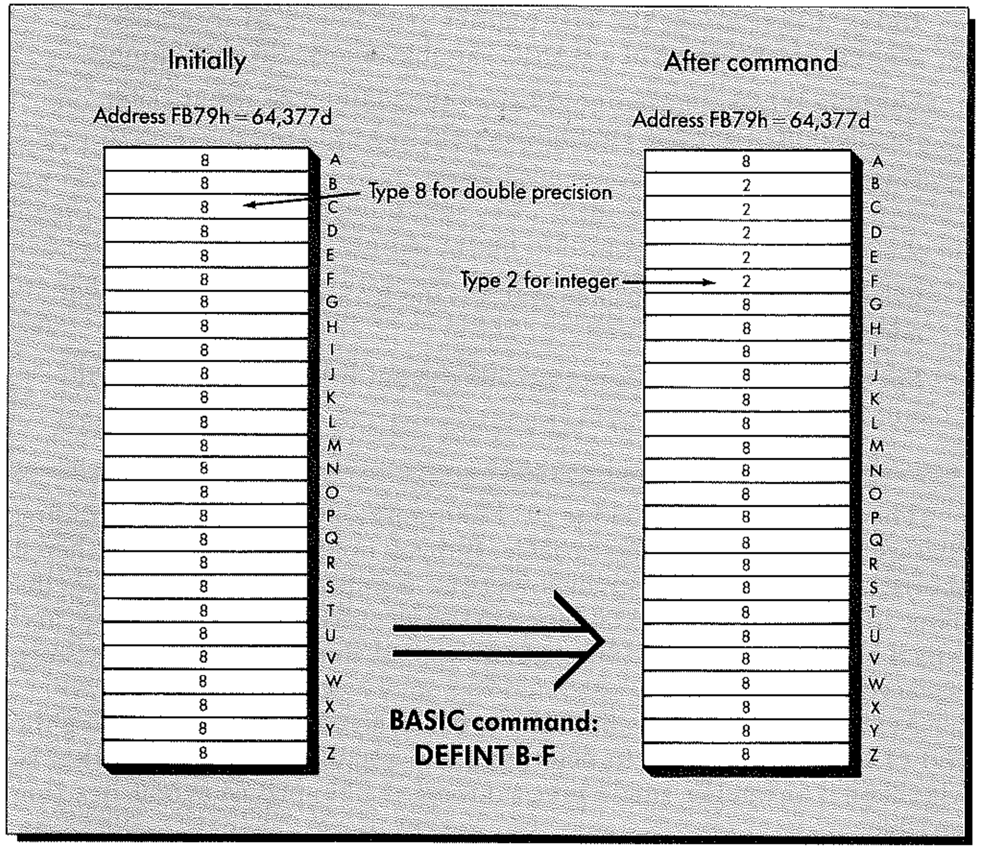
The address finder routine at 4790h = 18,320d first checks the first character in the name of the variable. If this is not between A and Z, then it aborts, declaring a syntax error; otherwise it puts the first character of the name in the C register. Next it picks up the second character, if present, and puts it into the B register. If there is no second character in the name, zero is placed in the B register.

Next, the routine skips through the rest of the name. After reaching the end of the name, it tries to determine the *type* of the variable — that is, whether it is an integer, single-precision real, double-precision real, or string variable.

There are two ways that the type can be determined. One is by means of special symbols such as %, $, !, and #. The other is by declaration through the DEFINT, DEFSNG, or DEFDBL statements. This last method specifies declared default data types according to the first character of the name. If one of the special symbols (%, $, !, or #) is not found, the address finder routine looks in a table at FB79h = 64,377d for the declared type (see Figure 3-4). No matter which way the type is determined, it is stored in location FB65h = 64,357d.

The routine then checks for the telltale parentheses of a variable that is an array entry. If the parentheses are present, it has to compute the position of the entry within the array. We won't describe this process, but it begins at 488Dh = 18,573d.

Next, the address finder routine searches for the name of the variable among the existing variables. Only the first two characters of the name are used. As we mentioned above, the C register contains the first character, and the B register contains the second character, if it is present. The code for this search runs from 4801h = 18,433d to 4828h = 18,472d.



	Initially	After command
A	8	8
B	8	2
C	8	2
D	8	2
E	8	2
F	8	2
G	8	8
H	8	8
I	8	8
J	8	8
K	8	8
L	8	8
M	8	8
N	8	8
O	8	8
P	8	8
Q	8	8
R	8	8
S	8	8
T	8	8
U	8	8
V	8	8
W	8	8
X	8	8
Y	8	8
Z	8	8

Figure 3-4. Declared default types

BASIC maintains its variables in a table whose address is stored at FBB2h = 64,434d (see Figure 3-5). If the address finder routine cannot find the variable in this table, it makes a place for the variable in the table. The code for doing this runs from 4835h = 18,485d to about 4875h = 18,549d.

The different types of variables require different amounts of storage in this table. Integer variables require 5 bytes, string variables require 6 bytes, single-precision real numbers require 7 bytes, and double-precision real numbers require 11 bytes. In each case, the first byte of its entry in the table contains a code for the type: 2 for integers, 3 for string variables, 4 for

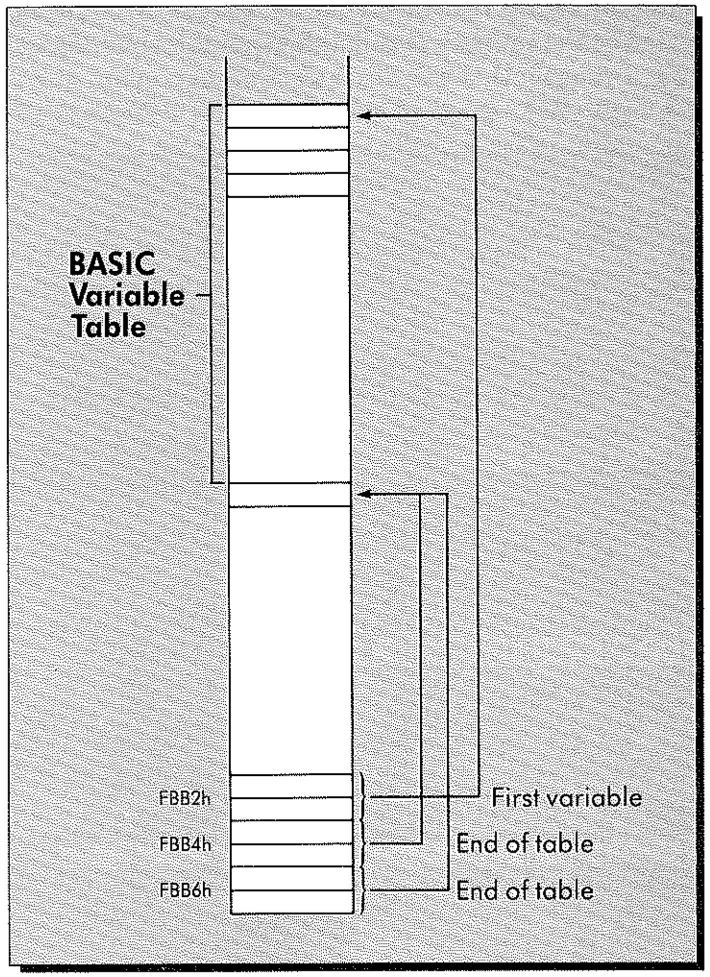


Figure 3-5. BASIC variable table

single-precision real numbers, and 8 for double-precision real numbers. You can see that the code for the type is 3 less than the number of bytes required in the table. For each of these types, the second and third bytes contain the ASCII code for the first character and second character in the name of the variable. If the name consists of only one character, a value of zero is used for the second character.

Now let's examine in detail how each type is stored (see Figure 3-6).

For an integer variable, the fourth and fifth bytes of its entry in the table contain the value of the integer in 16-bit two's complement binary form.

For string variables, the fourth byte contains the length of the string, and the fifth and sixth bytes contain the address of the location where the string is stored in memory.

For single- and double-precision real variables, the fourth through the last bytes contain the value of the number in a floating-point format. Floating point is like scientific format in that there is a sign, an exponent, and a mantissa. We will describe how each of these is stored in the Model 100's floating-point format.

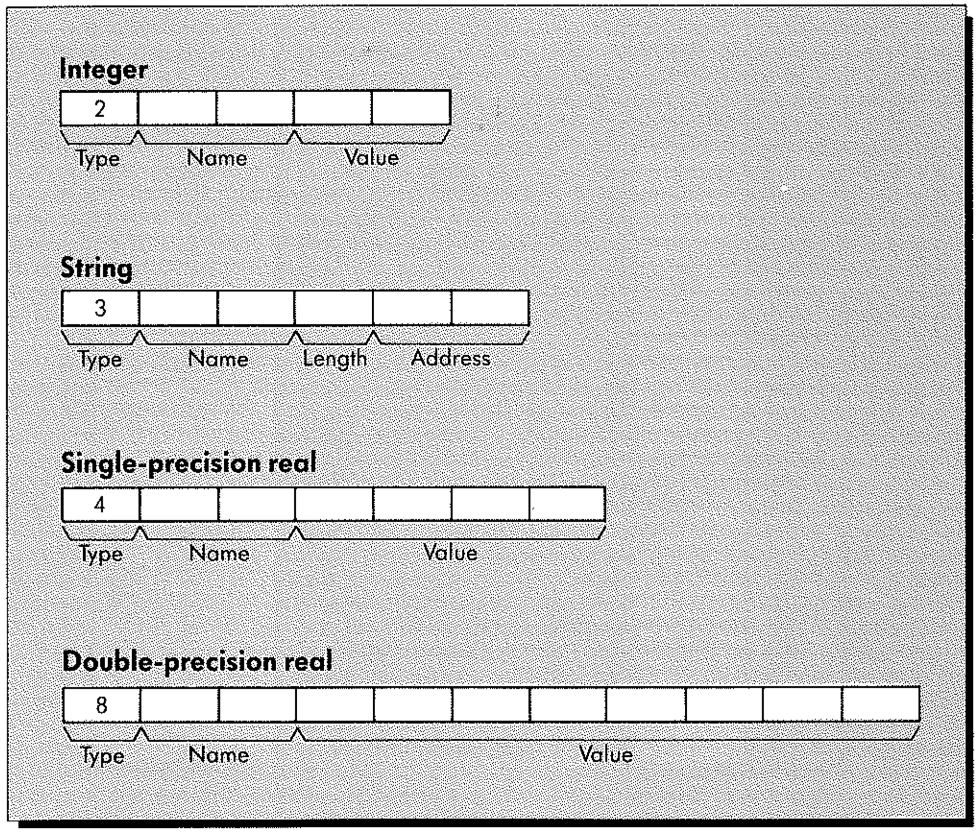


Figure 3-6. Storage allocation for BASIC variable

The sign of the number is stored in bit 7 of the first byte. A bit value of zero indicates a nonnegative number, and a bit value of 1 indicates a negative number.

The exponent of the number is stored in bits 0 through 6 of the first byte, with a bias of 64. This means that the actual exponent is obtained by subtracting 64 from the value stored in these bits.

The mantissa is stored in BCD form. That is, each decimal digit is represented by a nibble (four bits). For single precision there are six BCD digits in the three remaining bytes, and for double precision there are fourteen BCD digits in the remaining seven bytes. In either case, the decimal point is to the left of the most significant BCD digit.

Let's look at a couple of examples: the numbers 1.7 and -1.7. For number 1.7, the sign bit is 0, the actual exponent is 1, and the mantissa has the BCD digits 1 and 7 followed by zeros. This gives the expansion shown in Figure 3-7 for the four bytes allotted to the number.

For − 1.7, the only difference is that the sign bit is 1. The other bits are the same.

Now let's look at the routine to evaluate the right side of the equals sign. This expression evaluator routine is located at DABh = 3499d (see box).

Routine: BASIC Expression Evaluator

Purpose: To evaluate BASIC expressions

Entry Point: DABh = 3499d

Input: Upon entry, the HL register pair points to (contains the address of) a tokenized BASIC expression.

Output: When the program returns, the value of the expression is contained in BASIC's accumulator (locations FC18h = 64,536d to FC1Fh = 64,543d).

BASIC Example: Not directly applicable

Special Comments: None

The expression evaluator routine attacks an expression by assuming that the expression can be written as two expressions connected by a binary operation such as +, −, *, /, ^, or comparison. It assumes that the value of the expression on the left has already been determined. It loops around and around, trying to process a new binary expression each time. The first time through the loop, it calls a routine at F1Ch = 3868d to get the initial value on the left (see box).

Routine: BASIC Function Finder

Purpose: To evaluate unary expressions

Entry Point: F1Ch = 3868d

Input: Upon entry, the HL register pair points to (contains the address of) a term or function call of an expression in a tokenized BASIC command line.

Output: When the routine returns, the value of the term or function is in BASIC's accumulator (locations FC18h = 64,536d through FC1Fh = 64,543d).

BASIC Example: Not directly applicable

Special Comments: None

The expression evaluator uses the system stack to facilitate its operation. Each time through the loop it compares the priority of the current binary operation with the priority of the previous binary operation. If the old operation has higher priority, it is actually performed and then stored in an area of memory that acts as an accumulator; otherwise, all the information to be performed is pushed onto the stack. For some expressions, the priorities are such that many operations remain on the stack, but eventually all operations are performed.

A section of code running from DE6h = 3558d to E28h = 3624d pushes the needed information onto the system stack. This information includes the value and type of the expression on the left, the code for the operation, and a code for its priority. The priorities of these operations are contained in a table located at 2E2h = 738d. The routine also pushes two addresses of its own code onto the stack.

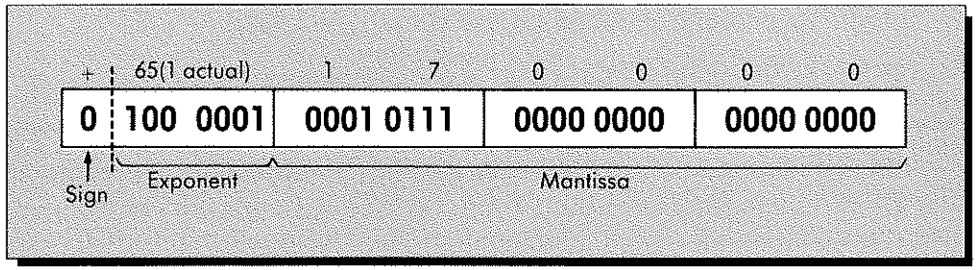


Figure 3-7. Floating point representation for 1.7

The locations FC18h = 64,536d through FC1Fh = 64,543d are used as an accumulator for the expression evaluator. For integers, just FC1Ah = 64,538d and FC1Bh = 64,539d are used; for single precision, locations FC18h = 64,536d through FC1Bh = 64,539d are used; and for double precision, all eight locations are used (see Figure 3-8).

Let's look at a fairly simple example. This example will illustrate how the stack and the accumulator are used and how special starting and ending conditions are handled.

Suppose we wish to evaluate the expression:

2 + 3 * 5 + 4

You should look at Figure 3-9 during the following explanation. The leftmost expression is 2. Its value is put in the accumulator. The first binary operation is +. The first binary operation is always pushed. First the value in the accumulator, and then the operation, are pushed onto the stack. The next value, 3, is then evaluated in the accumulator. The next operation is *. It has a higher priority than +, so the value 3 from the accumulator and the * operation are also pushed onto the stack. Next, the value 5 is processed in the accumulator. The next operation is +, which has a lower priority than the previous operation (*), so evaluation of the * is begun. The computation 3 * 5 is performed and the result left in the accumulator. At this point the stack contains the 2 and the +, and 15 is in the accumulator. The next operation is +, which has the same priority as the + on the stack. Thus the operation on the stack is performed, yielding 17 in the accumulator. Since no more operations are pending, the value 17 and then the + operation are pushed onto the stack. Finally, the 4 is processed into the accumulator. Since the + is the last operation, it is processed, yielding a result of 21 in the accumulator.

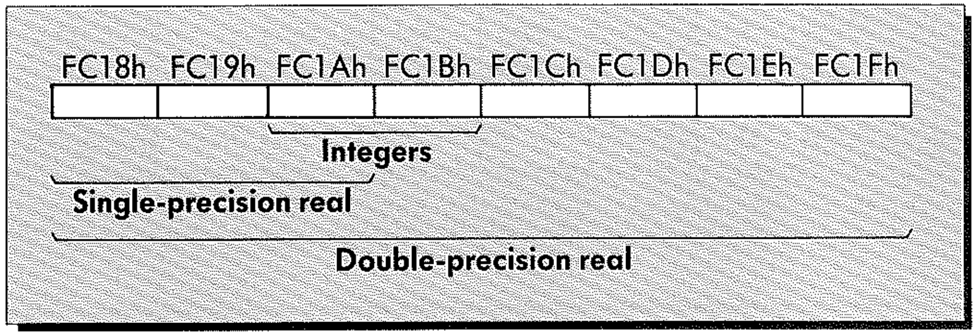
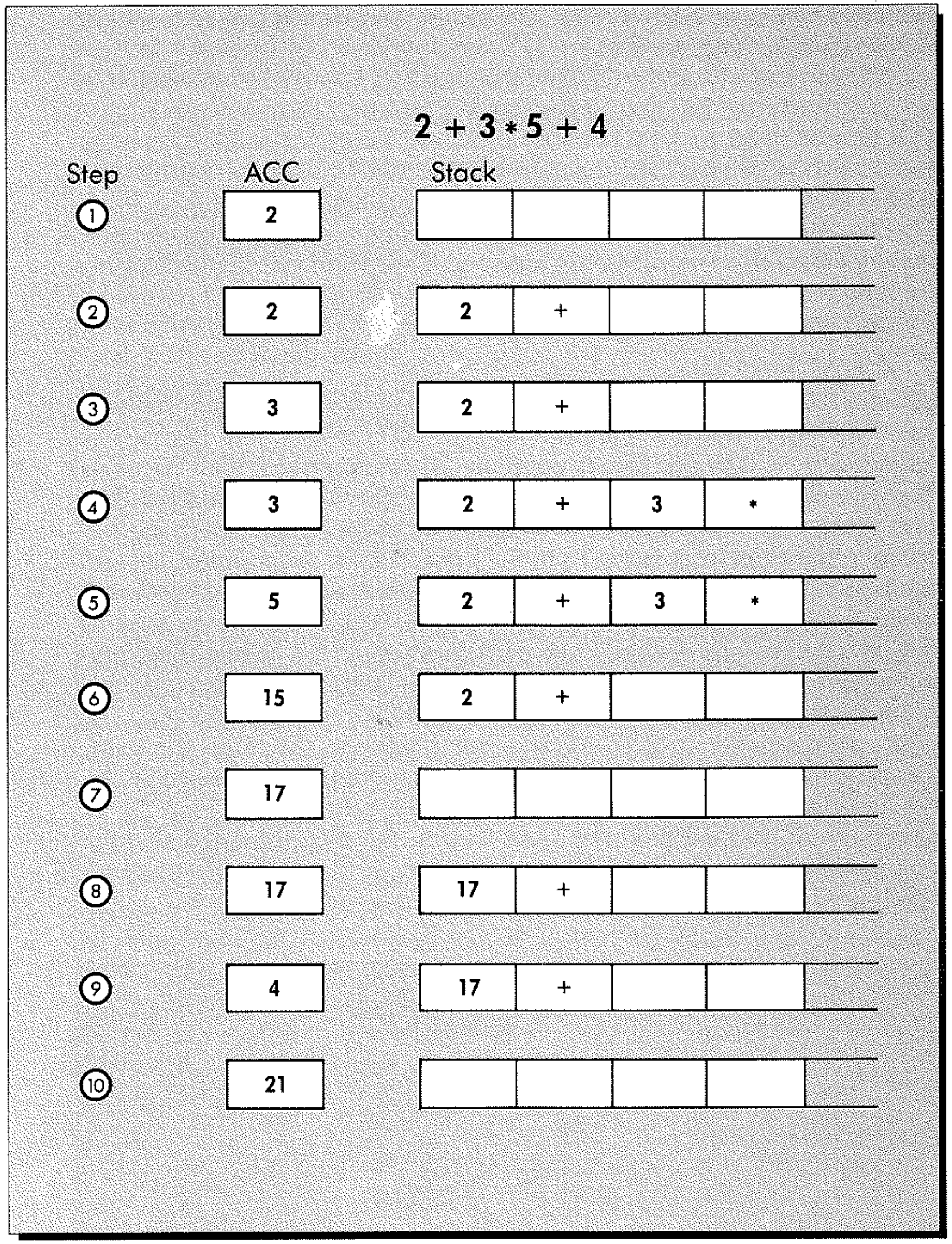


Figure 3-8. BASIC's accumulator



2 + 3 * 5 + 4

Step	ACC	Stack			
①	2				
②	2	2	+		
③	3	2	+		
④	3	2	+	3	*
⑤	5	2	+	3	*
⑥	15	2	+		
⑦	17				
⑧	17	17	+		
⑨	4	17	+		
⑩	21				

Figure 3-9. Evaluating an expression

The binary operations are performed starting at location E6Ch = 3692d. The code for the operation (+, −, *, and so on) is stored at location FB66h = 64,358d. The types are checked for the left and right sides. If they are not in agreement, various numerical conversion routines are called to make sure that the types do match. The tables in low ROM described earlier are used to dispatch to the appropriate operation.

Now let's look at the routine at F1Ch = 3868d, which evaluates single expressions. It uses the RST 2 call to skip spaces and look for numerical values. If it finds a numerical value, it jumps to 3840h = 14,400d, where the number is converted to the appropriate internal format. It next checks for a variable by calling the routine at 40F2h = 16,626d, which checks whether the next character is in the range from A to Z. If it is, the routine jumps to location FDAh = 4058d, where it calls the routine at 4790h = 18,320d to search for the variable among the existing variables. It gets the value of this variable and returns. The routine at F1Ch = 3868d continues, checking for such conditions as quotes, minus, NOT, and other unary operations and functions. It also looks for parentheses. It jumps to the appropriate code to handle whatever condition it finds.

Following this loop as it goes from F1Ch = 3868d through F46h = 3910d, to F51h = 3921d through F55h = 3925d, to F60h = 3936d through F7Dh = 3965d, to FA3h = 4003d through FCBh = 4043d, will allow you to see how the BASIC intepreter tries to detect all the possible "unary" operations.

The TELCOM Program

The TELCOM program allows you to use the Model 100 as a smart terminal for another computer either through an RS-232C communications line or through a telephone connection. You can even use it to upload and download text files.

The code for the TELCOM program runs from 5146h = 20,806d to about 5796h = 22,422d (see box). The main command input loop runs from 5152h = 20,818d to 5177h = 20,855d. The commands are input through the routine at location 4644h = 17,988d, which is called at location 516Ah = 20,842d. Dispatching is done via a routine at 6CA7h = 27,815d, which is called at 5175h = 20,853d. A table showing the names and addresses of the TELCOM commands starts at 5185h = 20,869d (see Table 3-1).

Routine: TELCOM

Purpose: To handle telecommunications

Entry Point: 5146h = 20,806d

Input: None

Output: Enters the TELCOM program

BASIC Example:

```
   CALL 20806
```

Special Comments: The routine does not return to BASIC.

One of the TELCOM commands is TERM, which puts you into terminal mode (see box). Here is the BASIC command that will take you directly from BASIC to TERM:

```
CALL 21589
```

This is useful if you have just saved a BASIC file to another computer over the RS-232C communications line, using the SAVE "COM:... command in BASIC, and want to reestablish two-way communications with the other computer.

Command	Address
STAT	5100h = 20,736d
TERM	5455h = 21,589d
CALL	522Fh = 21,039d
FIND	524Dh = 21,069d
MENU	5797h = 22,423d

Table 3-1. TELCOM commands

Routine: TERM — a TELCOM mode

Purpose: To establish terminal mode for telecommunications

Entry Point: 5455h = 21,589d

Input: None

Output: Enters terminal mode of the TELCOM program.

BASIC Example:

```
   CALL 21589
```

Special Comments: The routine does not return to BASIC.

The TERM mode has several commands. A "dispatcher" routine located at 54FCh = 21,756d (see Table 3-1) and a table starting at 550Dh = 21,773d are used to branch to the routines for each of these commands.

In Chapter 7 we will study the serial communications devices and the ROM routines that run them.

The MENU Program

The MENU program allows you to see what files you have in your Model 100 and select a particular one for editing or execution. It displays the main menu, which shows a directory of the files. The names of the files are displayed in six rows with four files to a row, giving a total of twenty-four possible files. You can select a particular file either by typing its name or by placing the cursor over the name in the directory and pressing ENTER.

The code for the MENU program extends from 5797h = 22,423d to about 5B67h = 23,399d. It consists of an initialization section, a main command loop, and a number of routines to handle the various cursor and dispatching commands. The command loop gets keystrokes from the user and interprets them as cursor and selection commands.

Routine: MENU

Purpose: To display the menu directory and select program or file

Entry Point: 5797h = 22,423d

Input: None

Output: The directory is displayed on the screen.

BASIC Example:

```
CALL 22423
```

Special Comments: This CALL will cause you to exit BASIC and return to the main menu.

The File Directory

The directory is displayed on the LCD screen as part of the initialization stage of the MENU program. There are a total of twenty-seven possible entries in the RAM directory. However, there are only twenty-four spots for entries in the LCD menu display. The other three entries are false entries; two secretly store the names of those who helped develop the Model 100.

The directory is stored in RAM starting at F962h = 63,842d (see Figure 3-10). Each entry in the RAM directory takes up eleven bytes. The first byte contains the file type and protection code (see Figure 3-11). The individual bits are assigned as follows: bit 7 is 1 if the entry is currently being used and 0 if it is invalid, bit 6 is 1 if it is an ASCII (DO) file, bit 5 is 1 if it is a machine-language file (CO), bit 4 is 1 if it is a ROM file, and bit 3 is 1 if it is an invisible file. Thus, for example, the ROM programs BASIC, TEXT, TELCOM, ADDRSS, and SCHEDL are stored as files of type B0h = 176d (valid, machine language, and ROM bits are all on). The second and third bytes contain the address of the body of the file in the usual low/high format. The next eight bytes contain the name of the file. For files created by the Model 100's various utilities, the first six bytes contain the main part of the name and the last two bytes contain the file extension. If the main part of the name is less than six characters, the remaining bytes between the main part of the name and the file extension are filled with spaces.

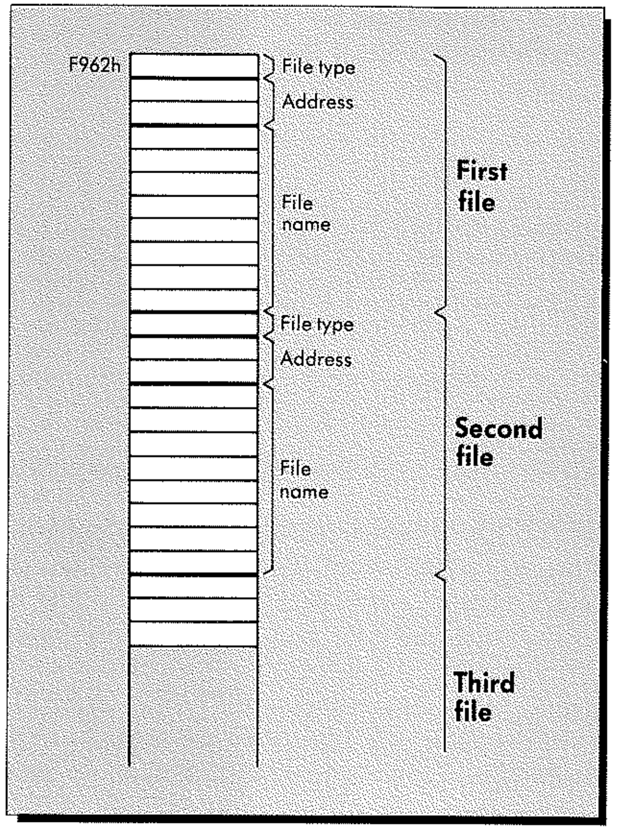


Figure 3-10. The directory

7	6	5	4	3	2	1	0
Valid entry?	ASCII file?	Machine language?	ROM file?	Invisible?	0	0	0

Figure 3-11. The bits of the file type byte

The following BASIC program displays the directory, giving file type and address for each file. It even shows the hidden files.

```
1ØØ  ' DISPLAY DIRECTORY
11Ø  '
12Ø    FOR I = 63842 TO 64138 STEP 11
13Ø     PRINT USING "###"; PEEK(I);
14Ø     ADR = PEEK(I+1)+256*PEEK(I+2)
15Ø     PRINT USING "######";ADR;"   "
16Ø     FOR K=3 TO 1Ø
17Ø      PRINT CHR$(PEEK(I+K));
18Ø     NEXT K
19Ø     PRINT
2ØØ    NEXT I
```

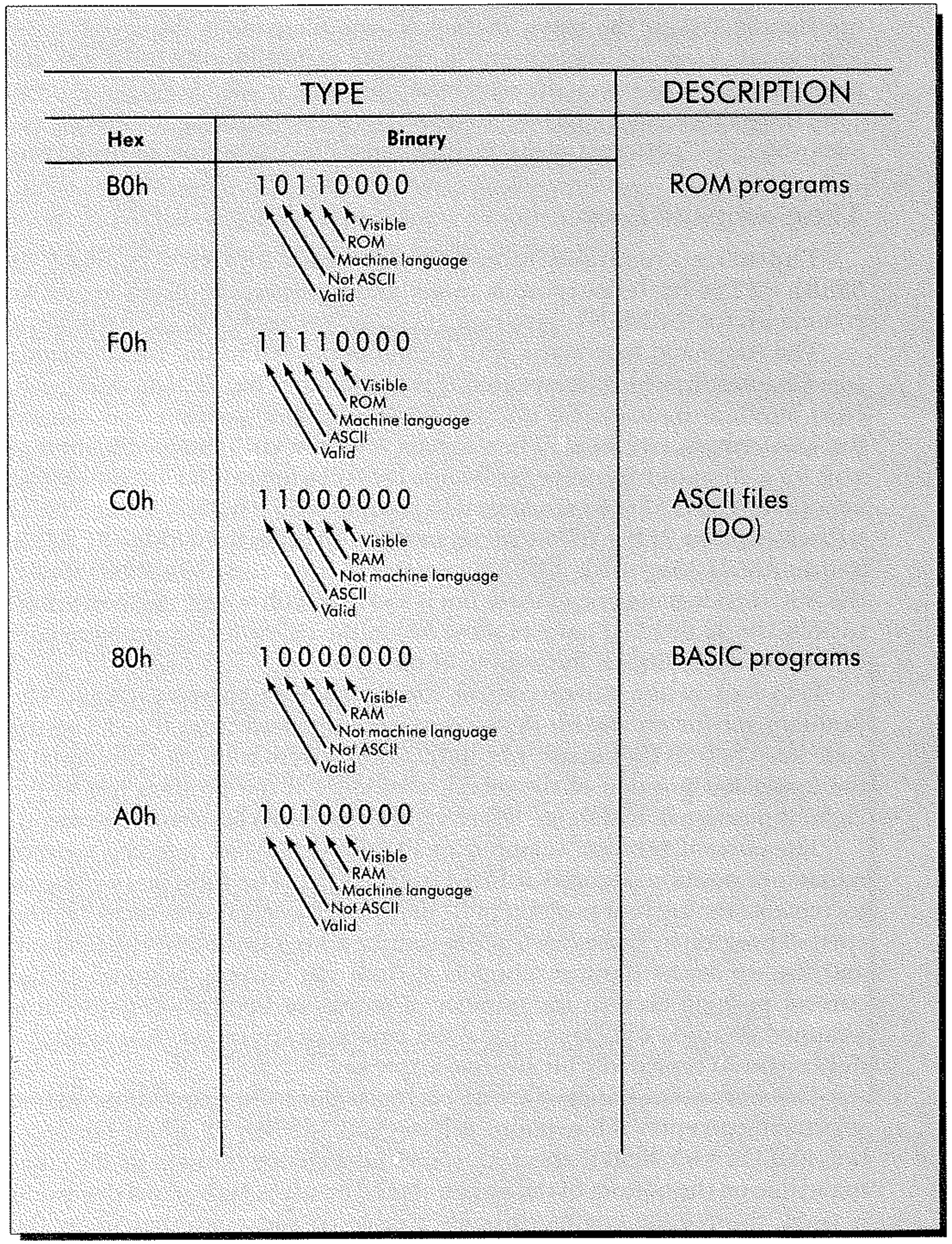
The program is straightforward. It consists of a loop that goes through all the directory entries, displaying the numbers and text as described above.

Now let's look at the MENU program in ROM. It is located at 5797h = 22,423d. It begins by setting up the screen for display of the menu. It turns off the reverse characters, cursor, and function key display, and it unlocks the display. It displays the time and date and the Microsoft copyright notice.

The main loop for displaying directory entries begins at 57F8h = 22,520d. Just before the loop, the DE register pair is loaded with the address of a short table located at 5B1Eh = 23,326d (See Figure 3-12). This table contains the following list of file types: B0h = 176d, F0h = 240d, C0h = 192d, 80h = 128d, and A0h = 160d. The table terminates with a zero. This routine will display directory entries with only these file types, and it will display them in this order. The first type (B0h = 175d) corresponds to the ROM program files such as BASIC, TEXT, and TELCOM. If you look at your display, you will see that they are always listed first. The third type (C0h = 192d) corresponds to regular ASCII files (file extension DO), and the fourth type (80h = 128d) corresponds to regular BASIC program files (file extension BA). You will notice that all ASCII (DO) files are listed before all BASIC (BA) files.

The main loop of the directory display gets a file type from the table, places it in the C register, and calls a routine at 5970h = 22,896d to display all directory entries of that file type. The loop continues until all the files of these five displayable types are displayed on the LCD screen.

The routine at 5970h = 22,896d loads the DE register pair with the beginning address of the directory (F962h = 63,842d) and the B register with the value 27, which is the total number of directory entries. It then goes into a loop that cycles through the entries, picking up the file type and



TYPE		DESCRIPTION
Hex	Binary	
B0h	10110000 Visible ROM Machine language Not ASCII Valid	ROM programs
F0h	11110000 Visible ROM Machine language ASCII Valid	
C0h	11000000 Visible RAM Not machine language ASCII Valid	ASCII files (DO)
80h	10000000 Visible RAM Not machine language Not ASCII Valid	BASIC programs
A0h	10100000 Visible RAM Machine language Not ASCII Valid	

Figure 3-12. Displayed file types

comparing it against the specified file type in the C register. If the types do not match, it skips the entry. If they do, the filename is displayed at the appropriate place on the screen. A routine at 59C9h = 22,985d sets the proper cursor position on the screen for the entry.

The main directory display routine concludes by filling in missing entries with a "-.-" pattern and displaying "Select:" at the bottom of the screen.

The Command Loop

The main command loop extends from 585Ah = 22,618d to 588Bh = 22,667d. Its purpose is to get and interpret the user's keystroke commands for the MENU program.

The command loop starts with a conditional call to the BEEP routine (see Chapter 8), which is executed if the command buffer has overflowed. Next it calls a routine at 5D70h = 23,920d to update the time and date on the LCD screen (see box). Then it calls a routine at 5D64h = 23,908d to wait for a character from the keyboard. It checks for various special ASCII codes. An ASCII 13 (enter) causes it to jump to 58F7h = 22,775d, an ASCII 8 (backspace) or 7Fh = 127d (delete) causes it to jump to 588Eh = 22,670d, and an ASCII 15h = 21d (CTRL) U causes it to jump to 5837h = 22,583d. If the ASCII code is not one of these but is less than 20h = 32d, the loop jumps to 589Ch = 22,684d. If none of these occur, the character is printed as part of the command line on the bottom of the display.

Let's look at the cursor control commands in a bit more detail. The positions for the entries on the screen are numbered from 0 to 23 in the code that controls the cursor. The numbering system is simple: position 0 is the upper left position of the menu, position 1 is the position just to the right of position 0, and so on, left to right from top to bottom of the menu.

The current position is stored in FDEEh = 24,046d, and the current maximum position is stored at FDEFh = 65,007d. The right arrow routine increments the current position by 1, the left arrow routine decrements the current position by 1, the down arrow routine adds 4 to the current position, and the up arrow routine subtracts 4 from the current position. If the current position exceeds the number of entries or becomes negative, it is wrapped around. You can see how the cursor wraps or cycles around the display as you repeatedly hit any one arrow key.

The routine to handle (ENTER) actually does the dispatching to the selected menu entry. This routine is located at 58F7h = 22,775d. The location FDEDh = 65,005d is checked to see if the cursor position or the command line on the bottom of the screen should be used. If the cursor position is used, the routine counts through a table of addresses that maps the position numbers of the entries on the screen with the corresponding positions in the directory in RAM. This table begins at FDA1h = 64,929d. The (ENTER) routine finds the directory entry in RAM and checks the file type. For file type A0h = 160d, it jumps to 254Bh = 9547d; for file type B0h (ROM command files), it jumps to 596Fh = 22,895d; for file type F0h = 240d, it jumps to F624h = 63,012d (a RAM location); and for file type C0h = 192d (ASCII files), it jumps to 5F65h = 24,421d. If the file type is none of the above, it is treated as a BASIC file. After shoving the beginning address into location F67Ch = 63,100d and doing a couple of other things, the routine jumps to the execution loop of the BASIC interpreter.

The ADDRSS and SCHEDL Programs

The ADDRSS and SCHEDL programs are utilities that allow you to use your machine better. The ADDRSS program helps you to find addresses and telephone numbers that you have stored in your Model 100's memory, and the SCHEDL program helps you look up notes of things.

The code for the ADDRSS program extends from 5B68h = 23,400d to about 5B6Eh = 23,406d. The code for the SCHEDL program extends from 5B6Fh = 23,407d to about 5BA8h = 23,464d. Both programs jump to 5B74h = 23,412d.

Routine: ADDRSS

Purpose: To locate addresses and telephone numbers in the personal address directory file ADRS.DO

Entry Point: 5B68h = 23,400d

Input: None

Output: Enters the ADDRS program

BASIC Example:

```
CALL 23400
```

Special Comments: This CALL does not return to BASIC.

Routine: SCHEDL

Purpose: To locate schedule items in the personal notes file NOTE.DO

Entry Point: 5B6Fh = 23,407d

Input: None

Output: Enters the SCHEDL program

BASIC Example:

```
CALL 23407
```

Special Comments: This CALL does not return to BASIC.

The ADDRSS and SCHEDL programs behave very much like, and share much code with, the TEXT program described below. For this reason, we will not investigate these programs any further.

The TEXT Program

The TEXT program is the editor or word processor for the Model 100. The code for this program extends from 5DEEh = 24,046d to about 6BF0h = 27,632d. It consists of an initialization section, a main character input loop, and a set of routines to implement the various cursor control and editing commands.

Routine: TEXT

Purpose: To edit text files

Entry Point: 5DEEh = 24,046d

Input: None

Output: Enters the TEXT program

BASIC Example:

```
CALL 24046
```

Special Comments: This CALL does not return to BASIC.

The first part of the code for TEXT sets up the screen and gets the file to be edited. It calls a routine at 5A7Ch = 23,164d to set the function key display. It uses the table at 5E22h = 24,098d, which is empty. There are no function keys available at this point of the TEXT program. It then displays a message asking you to enter the file. It calls a routine at 2206h = 8710d to get the filename and locate the file and then jumps to 5F65h = 24,421d to edit the file. If a new file needs to be created, the routine called MAKTXT at 220Fh = 8719d is called. Location 5F65h = 24,421d is the same entry point dispatched to by the MENU program for ASCII files.

Routine: MAKTXT

Purpose: To create a text file

Entry Point: 220Fh = 8719d

Input: Upon entry, the filename must be stored in memory, starting at location FC93h = 64,659d. The .DO part of the file name need not be included.

Output: Enters the TEXT program

BASIC Example:

```
CALL 8719
```

Special Comments: None

The code at 5F65h = 24,421d sets up the screen for normal editing and loads the function keys with the options "Find", "Load", "Save", "Copy", and so on.

The main edit loop extends from 5FDDh = 24,541d to 6015h = 24,597d. The loop looks like a subroutine with a RET instruction at the end. However, at the top of the loop, the address of the top of the loop is pushed onto the stack. Thus the RET returns to the top each time.

At 5FEDh = 24,557d, the edit loop calls a routine at 63E5h = 25,573d to get the next key (see box). For control characters, it uses a table at 6015h = 24,597d to dispatch to routines to perform the various editing functions. For regular characters, it jumps to 608Ah = 24,714d, where it enters the character into the text.

Routine: Get Key

Purpose: To wait for a key for TEXT program

Entry Point: 63E5h = 25,573d

Input: From the keyboard

Output: Upon return, the ASCII code of the key is in the A register.

BASIC Example: Not applicable

Special Comments: Only called from TEXT

The Initialization Routines

The initialization routines help set up or configure the various I/O devices in the Model 100 to start the computer after it gets stuck and will not respond to you through the keyboard.

The code for warm and cold I/O initialization runs from about 6CD6h = 27,862d to 6D3Eh = 27,966d (see boxes). The last part of the ROM, starting at 7D33h = 32,051d, contains code to start up the computer.

Routine: INITIO — Cold Start

Purpose: To cold start the I/O of the Model 100

Entry Point: 6CD6h = 27,862d

Input: None

Output: The Model 100 I/O is reset — cold start.

BASIC Example:

```
CALL 27862
```

Special Comments: Watch out, this is a cold restart! It clears the area from FF40h = 65,344d to FFFDh = 65,533d. This area contains the keyboard buffer, among other things. It continues into the warm start reset after clearing that area.

Routine: INITIO — Warm Start

Purpose: To warm start the I/O of the Model 100

Entry Point: 6CE0h = 27,872d

Input: None

Output: Initializes the I/O devices of the Model 100.

BASIC Example:

```
CALL 27872
```

Special Comments: None

The Primitive Device Routines

The code for the primitive-level routines for handling devices runs from about 6D3Fh = 27,967d to 7D32h = 32,050d and includes tables. In subsequent chapters we will study many of these routines in detail.

Summary

In this chapter we have surveyed the Model 100's ROM from beginning to end. We have concentrated on the areas that manage the BASIC interpreter and the MENU program, for these are the keys to understanding many other secrets of the Model 100's operation.

A particular area of interest is the code for the LET command, in which the BASIC interpreter finds variables and evaluates expressions. We have shown how you can gain direct control of this code to make an interactive function evaluator.