Inside The TRS-80® Model 100

by
Carl Oppedahl

Weber Systems, Inc.
Cleveland, Ohio

Published by:
Weber Systems, Inc.
8437 Mayfield Road
Chesterland, Ohio 44026

For information on translations and book distributors outside of the United States, please contact WSI at the above address.

Inside the TRS-80® Model 100

Library of Congress Cataloging in Publication
Main entry under title:

Inside the TRS-80® Model 100

Includes index
1. TRS-80 Model 100 (Computer) I. Title.
QA76.8.T184566 1985 001.64 85-11560
ISBN 0-938862-31-6

Contents

Preface

How To Use This Book

This book is intended for several readers: those who wish to do machine language programming; those who wish to do sophisticated BASIC programming; and those who wish simply to understand how the Model 100 works inside.

Understanding the Model 100

Some Model 100 owners are reasonably content using existing software, and so do not really have programming in mind. For these readers, it is my hope that this book will do two things: provide an explanation of how the computer "really works inside"; and perhaps arouse a little curiousity about the interesting world of machine language programming. It is possible to read the entire book from front to back without assembling a single opcode, or typing RUNM even once. Along the way you may still find yourself asking "I wonder how they do that?" If even once you can say, "Ah, now I see how it is done," then this book will have fulfilled its purpose.

Doing Machine Language Programming

For those who wish to do machine language programming, the course of study depends on your previous experience with machine language.

If you have never programmed in machine language before, you should first become reasonably familiar with Model 100 BASIC, including such statements as OPEN, LINE INPUT#1, PRINT#1. You should also learn to use TEXT, the Model 100 word-processing program.

Then read this book starting from the very beginning, skipping only chapter 4. If you feel comfortable with the subjects covered up to and including chapter 5, then proceed with the rest of the book. You will be able to do almost anything you want with the Model 100.

You may find, after reading those chapters, that you need a more general introduction to microprocessors; a list of suggested books appears in the appendices.

If you have programmed in machine language, whether on another microprocessor or on a large computer, you will be able to start right in with chapters 2 and 3, which introduce the 8085 CPU used in the Model 100. Proceed to chapter 5 which describes the hardware and from there read to the end of the book.

If your previous experience is with one or more processors in the 8080 family, you should have no difficulty starting right in with chapter 5. Those whose first computer was a TRS-80 Model I, III, or IV, and who thus know about the Z80 will want to read chapter 4 closely, as it tells how to use Z80 experience and programming aids on the Model 100.

Advanced BASIC Programming

Some readers have mastered everything in the Model 100 owner's manual, including the complete BASIC command vocabulary. Many want to do more, but do not want to learn to do machine programming. The coverage in the owner's manual is sparse, at best, regarding such commands as PEEK, POKE, INP, and OUT, CALL, RUNM, and the like. This book explains the internal features of the Model 100 that may be accessed and controlled through these commands.

If your BASIC programming goal relates to a particular part of the computer, say, the clock/calendar integrated circuit, you may go directly to the chapter on that subject. This chapter describes ways to use the CALL, INP, OUT, PEEK, and POKE commands to perform task, which cannot be accomplished using the common BASIC commands.

For a survey of the many hardware areas which may be controlled with INP and OUT commands, read chapter 5.

Corrections and suggestions for improvement are welcomed, and should be sent to the author.

About The Author

Carl Oppedahl, a lawyer specializing in technological litigation, is a regular contributor to Portable 100/200 and PCM magazines. He holds a bachelor's degree in physics and mathematics from Grinnell College and a law degree from Harvard University. He is an associate with the New York law firm of Kreindler & Kreindler.

Acknowledgements

The author would like to thank Alan Buck, whose unselfish efforts brought the world of computer programming to hundreds of high school students a decade before it was fashionable to do so; Harold Liljedahl, who taught me that when reassembling equipment I should put all the screws in part way before tightening them; Gerald Robbie, who taught me most of what I know about writing; and Lee Kreindler, whose encouragement and support suffice to make almost anything possible.

1

Introduction

Why Machine Language Programming?

The Microsoft BASIC and the nice TEXT and TELCOM programs provided with the Model 100 allow you to undertake an incredible range of programming tasks. Custom software from other suppliers allows one to do almost anything, without having to learn machine language or study opcodes.

Why, then, learn machine language? Machine language is fast—often ten times faster than an equivalent BASIC program, and more compact. If TEXT and TELCOM had been written in BASIC rather than machine language, they would have taken up at least three times the 6.5K they presently occupy in ROM.

The TEXT program, if written in BASIC, would require four minutes for a simple "Find" search in a large file; in machine language it takes less than fifteen seconds.

The TELCOM program simply had to be written in machine language; when characters are arriving at the rate of thirty or more per second and a download to RAM and printer echo are occurring, BASIC simply cannot keep up.

Fast-action games are painfully slow in BASIC; exciting graphics and sound are possible only in machine language.

These are some of the reasones people learn to program in machine language.

What Is Machine Language?

Machine language is just what is sounds like, the language used within the machine. Actually computers are always executing machine language "deep down inside", even when we think of them as executing BASIC, or FORTRAN, or whatever.

The Model 100 computer contains a microprocessor, sometimes called a CPU (for central processor unit), and a variety of memory and I/O (input and output) devices, as shown in figure 1.1.

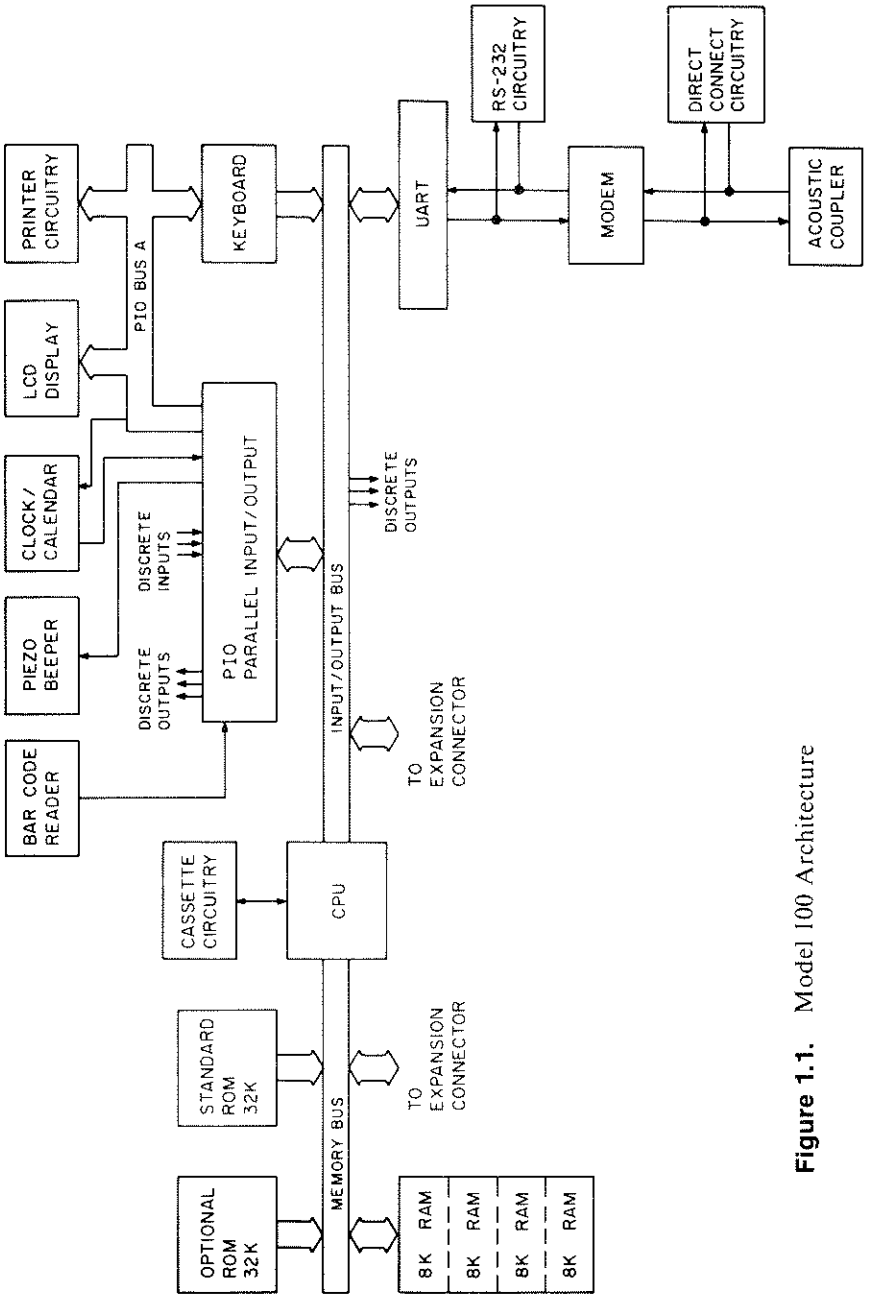

Figure 1.1. Model 100 Architecture

The CPU can communicate directly with the cassette connector; can use the memory bus to store and retrieve information; and can use the input/output (I/O) bus to control and exchange information with a variety of devices, including the keyboard, the liquid-crystal display (LCD) screen, a printer, the telephone line, or any RS-232 device.

That is all fine and good, but how does it really work? For example, when the power is turned on, what exactly happens to result in the time, date and file menu being displayed? When one moves the cursor to one of the file names, and pushes ENTER, what exactly happens?

The answer to such questions lie in an understanding of the way the CPU deals with the circuitry around it, and in a knowledge of the ROM operating system.

The inner workings of the CPU and the operating system provide a background that is presently throughout this book, regardless of the device named in the title of particular chapter.

The CPU

The CPU executes instructions called "opcodes", which are made available to the CPU in memory locations. When power is turned on, the CPU goes to the opcode located at memory location 0000. As it happens, the designers of the Model 100 set up the hardware so that memory location 0000 is in read-only memory (ROM).

Memory location 0000 contains the hexadecimal value C3, or 195 decimal. (You can see this by typing PRINT PEEK (0) while in BASIC.)

The 8085 CPU is designed so that when it encounters the value C3, it prepares to "jump", which means that instead of proceeding with the opcode after C3, it proceeds with an opcode elsewhere in memory. The next two storage locations contain information used by the CPU to figure out where it will jump to. (The jump instruction is analogous to the GOTO command in BASIC.)

The next two locations contain 33 hexadecimal (51 decimal) and 7D hexadecimal (125 decimal). The CPU takes the 33 and 7D groups them as 7D33, and jumps to 7D33.

Hexadecimal Notation

This is as good a time as any to introduce hexadecimal notation. Most addresses, port numbers, and register contents (all defined later) will be given in hexadecimal, usually called "hex", notation. The reason is that everything in the Model 100 (and indeed everything in every microcomputer) is represented by 1's and 0's usually in groups of eight called "bytes".

Each bytes is capable of assuming 256 different values, since there are 256 possible combinations of "1" and "0". Sometimes the value of a particular byte is conveyed with a decimal number from 0 to 255, using a convention assigning decimal values to the individual 1's and 0's, which are called "bits".

For example, the eight-bit value 01110001 has a decimal value of 113, arrived at in this fashion:

(0 x 128) + (1 x 64) + (1 x 32) + (1 x 16) + (0 x 8) + (0 x 4) + (0 x 2) + (1 x 1)

The numbers 128, 64, and so on are powers of two: 128 is two raised to the seventh power, 64 is two raised to the sixth power, and so on. The "0" bit which was multiplied with 128 is thus often referred to as "bit 7", the "1" which was multiplied with 64 is called "bit 6", and so on.

This describes the relation between binary numbers and decimal (base ten) numbers. The connection with hexadecimal is as follows: a group of four bits is matched up with a letter or numerical digit, so that a group of eight bits is matched with a two-letter (or two-digit) combination.

The four-bit values are:

Binary	Hex
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	A
1011	B
1100	C
1101	D
1110	E
1111	F

Using this correspondence, the binary value 01110001 translates to 71 hex.

The Opcode Instruction Set

As mentioned above, the 8085 executes opcodes. Thus far, only one opcode has been mentioned, the C3 opcode which means "jump". By the end of chapter 2, the other opcodes which the 8085 is willing to execute will have been discussed. These include instructions which cause information to be moved from one place to another (the equivalent of LET A=B); which will test for presence of certain conditions and then jump (the equivalent of IF statements); and which add or subtract integers.

Assemblers

Each opcode is an eight-bit byte, and in some cases the one or two bytes following the opcode also form part of an instruction. The two

bytes following the "C3" discussed above give an example of the latter. Thus an instruction may be one, two, or three bytes long.

The bottom 32,768 (decimal) Model 100 memory locations, which are in ROM (running from 0000 to 7FFF), together with the top 2574 (decimal) memory locations, which are in RAM (running from F5F0 to FFFF), are filled with opcodes and related data placed there by Microsoft. Those opcodes, which may be grouped into subroutines and programs, make possible the menu that will be displayed when you turn on the machine as well as the events which occur when you run the applications programs.

The allocation of memory is shown to scale in figure 1.2.

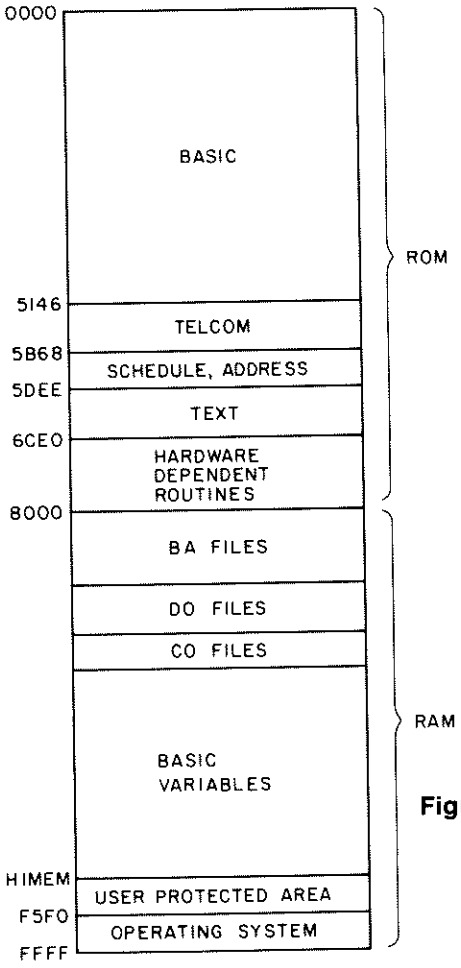

Figure 1.2. Memory Map

When Microsoft developed the opcodes, they were doing machine language programming. You may be sure they did not start by creating a list of hex opcodes and running these. Instead, they followed a development process involving flowcharts, mnemonics, and a variety of programming aids such as assemblers, debuggers and disassemblers.

2

Assembly Programming

Architecture

The 8085 CPU is composed of several parts, as shown in figure 2.1. Signals enter and exit from the CPU at the locations shown in the figure. Sixteen wires are available for selecting which of the 65536 memory addresses is being referred to. Eight of the wires are also used for incoming and outgoing data. The signals shown across the top of the figure have to do with serial input and output, and interrupts, discussed in detail in chapters 12 and 15, respectively. Timing and control signals shown at the bottom of the figure, let the CPU determine whether input or output will occur, and whether the input or output will take place in port space or address space.

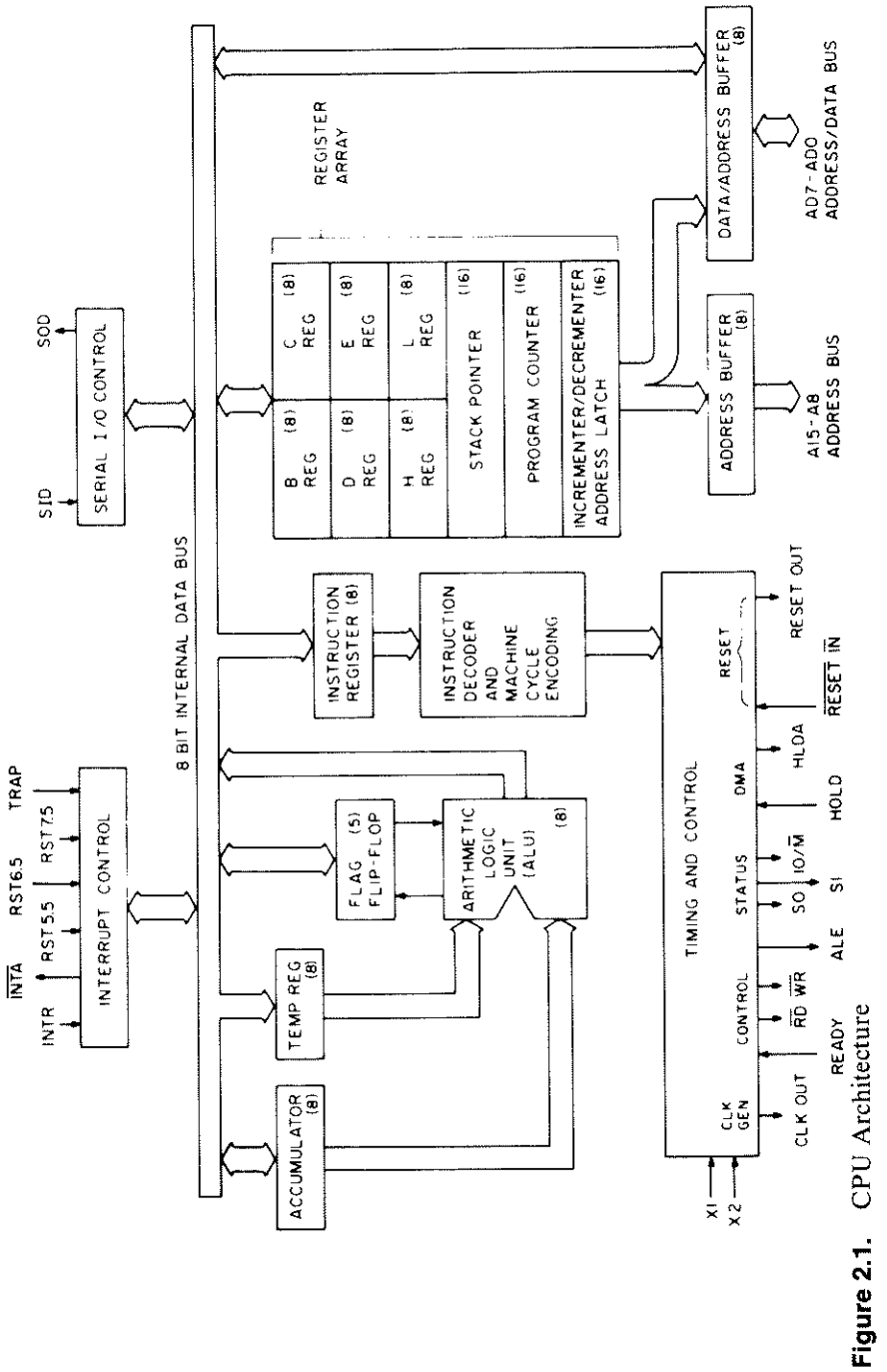

Figure 2.1. CPU Architecture

Within the CPU, various registers contain ones and zeros. These can be manipulated by the programmer through selected opcodes. The registers that are used most often are the A, B, C, D, E, F, H, L, SP, and PC registers.

The registers designated by a single letter are eight bits in size, while those designated by two letters are sixteen bits in size. Consistent with this convention, the eight-bit registers may be paired up, creating the AF, BC, DE, and HL registers, often called register pairs.

The Accumulator and the Flag Register

The A register, also called the accumulator, is probably the most frequently used of the 8085 registers. Most of the other registers cannot be used for arithmetic calculations. Only the accumulator has an arithmetic logic unit (ALU) for addition, subtraction, multiplication, and logical AND and OR functions. Only the accumulator can be loaded to and from the input and output ports. Whenever an arithmetic operation is performed in the accumulator, one or more bits of the F or *flag* register will be affected. The Carry flag, for instance, is said to have been *set* if its new value is one or *reset* if its new value is zero.

Set and reset are also used to refer to individual bits elsewhere in the Model 100, including the I/O ports. "Turn on bit 4", "set bit 4," and "make bit 4 equal to one" are synonymous; so are the terms "turn off bit 0," "reset bit 0," and "make bit 0 equal to zero."

Table 2.1. F register

Bit	Symbol	Meaning
0	CY	Set if a *carry* resulted from bit 7 during an addition or a *borrow* resulted from bit 7 during a subtraction. Otherwise reset.
2	P	Set if the parity (number of ones in the accumulator) is even. Otherwise reset.
4	AC	Set if a *carry* resulted from bit 3 during an addition. Otherwise reset.
6	Z	Set if the value in the accumulator is zero. Otherwise reset.
7	S	This is a copy of bit 7 from the accumulator, often used to represent the "sign" of a small integer.

The flags of the F register are laid out as shown in table 2.1. These flags do not change every time the contents of the accumulator change. For example, if the Z flag is set, loading a nonzero value into the accumulator does not, in and of itself, reset it. The flags are set only during certain arithmetic and logic operations.

The condition of a particular flag is most often used as the determining factor in a conditional jump. The contents of the flag register can also be treated as an eight-bit byte and loaded to other registers through the PUSH and POP instructions.

The Other Registers

The B, C, D, E, H, and L registers can be used as general-purpose storage locations. It is easy to move information among them as well as between them and the accumulator.

In addition, they can be used as register pairs. BC and DE are general-purpose register pairs, while HL can be used for a variety of addressing techniques, determining the address in memory to and from which information can be transferred.

When the HL register is used for this purpose, H contains the high-order portion of the address (bits 8 through 15), while L contains the low-order portion (bits 0 to 7).

The PC, or program counter, contains the numerical value of the address containing the opcode that is currently being executed. Usually the PC slowly increases by one address at a time, but its value changes drastically when, for example, a jump instruction is executed.

The SP, or stack pointer, is used whenever subroutine calls and returns occur. It manages an area of RAM called the *stack*. As will be seen in the discussion of the CALL, RET, PUSH and POP instructions, the stack is a LIFO (last-in, first-out) storage device. The design of the 8085 is such that items are placed on the *bottom* of the stack. As the stack grows, it grows into lower addresses of RAM.

Clock Frequency and Execution Speed

From time to time, it is interesting to know just how long it takes for the 8085 to execute an opcode, a subroutine or an entire program. This is helpful in deciding which of several programming techniques will be faster, and is crucial in the design of certain routines which perform time-sensitive input or output functions.

The rate at which the Model 100 executes opcodes is determined by the crystal frequency provided to the 8085 CPU. In the Model 100 it is a 4.9152 Megahertz (cycles per second) crystal designated X2. It happens that the 8085 divides this frequency by two, yielding 2.4576 MHz, and uses that lower frequency to time the opcodes.

At 2.4576 cycles per second, each cycle is about 0.407 microseconds long. Each opcode requires a specified number of cycles to complete. These numbers are listed in this chapter. (For those who have worked with the 8080 or Z80, be careful, as the 8085 cycle times are sometimes different.)

The Opcodes

The most often used opcodes are the *move* or *load* instructions, which do just what you might think they would do.

These instructions have the 8080 mnemonic MOV r,r' (for the word *move*) and Z80 mnemonic LD r,r' (for the word *load*). The 8080

and Z80 mnemonics are discussed in detail in chapter 4. In binary, the opcode is "01ddsss" where ddd is a three-bit representation of the destination register, and sss is a three-bit representation of the source register.

The three-bit values are:

111	A
000	B
001	C
010	D
011	E
100	H
101	L

The opcode 57, or 01010111, causes the contents of the A register (value 111) to be moved to the D register (value 010). The contents of the source register are unchanged. The corresponding mnemonics are MOV D,A and LD D,A.

Seven of the forty-nine possible moves are quite uninteresting, since they accomplish nothing -- such as a move from D to D or from A to A for example. The interesting moves are listed in table 2.2. The MOV instruction requires four clock cycles. This is different from the 8080 which requires five cycles.

Register moves have no effect on the flag register. No single instruction allows loading to or from the F register.

Table 2.2. 8085 register moves

Decimal	Hex	8080 Mnemonic	Z80 Mnemonic
120	78	MOV A,B	LD A,B
121	79	MOV A,C	LD A,C
122	7A	MOV A,D	LD A,D
123	7B	MOV A,E	LD A,E
124	7C	MOV A,H	LD A,H
125	7D	MOV A,L	LD A,L
71	47	MOV B,A	LD B,A
65	41	MOV B,C	LD B,C
66	42	MOV B,D	LD B,D
67	43	MOV B,E	LD B,E
68	44	MOV B,H	LD B,H
69	45	MOV B,L	LD B,L
79	4F	MOV C,A	LD C,A
72	48	MOV C,B	LD C,B
74	4A	MOV C,D	LD C,D
75	4B	MOV C,E	LD C,E
76	4C	MOV C,H	LD C,H
77	4D	MOV C,L	LD C,L
87	57	MOV D,A	LD D,A
80	50	MOV D,B	LD D,B
81	51	MOV D,C	LD D,C
83	53	MOV D,E	LD D,E
84	54	MOV D,H	LD D,H
85	55	MOV D,L	LD D,L
95	5F	MOV E,A	LD E,A
88	58	MOV E,B	LD E,B
89	59	MOV E,C	LD E,C
90	5A	MOV E,D	LD E,D
92	5C	MOV E,H	LD E,H
93	5D	MOV E,L	LD E,L
103	67	MOV H,A	LD H,A
96	60	MOV H,B	LD H,B
97	61	MOV H,C	LD H,C
98	62	MOV H,D	LD H,D
99	63	MOV H,E	LD H,E
101	65	MOV H,L	LD H,L
111	6F	MOV L,A	LD L,A
104	68	MOV L,B	LD L,B
105	69	MOV L,C	LD L,C
106	6A	MOV L,D	LD L,D
107	6B	MOV L,E	LD L,E
108	6C	MOV L,H	LD L,H

Memory Moves

One-byte moves of information can also be performed between a register and any location in memory. The location in memory is determined or *pointed to* by the HL register. As a result, the Z80 mnemonic for this instruction includes the symbol (HL), which means the location in memory whose address is in HL.

LD E,(HL) or MOV E,M, where M stands for memory, takes the contents of the memory location determined by HL and places it in E. The contents of the source code register or source memory location remain unchanged. These data transfers require seven clock cycles.

This is an example of so-called *indirect addressing*, in which the memory location involved is determined by the contents of a register pair.

The memory move opcodes take the form "01dddsss" where ddd or sss are 110 when referring to (HL). Otherwise the memory move opcodes obtain their value as in the Mov r,r' instruction above. The menmonics are listed in table 2.3.

Table 2.3. Moves to and from memory

Decimal	Hex	8080 Mnemonic	Z80 Mnemonic
126	7E	MOV A,M	LD A, (HL)
70	46	MOV B,M	LD B, (HL)
78	4E	MOV C,M	LD C, (HL)
86	56	MOV D,M	LD D, (HL)
94	5E	MOV E,M	LD E, (HL)
102	66	MOV H,M	LD H, (HL)
110	6E	MOV L,M	LD L, (HL)
119	77	MOV M,A	LD (HL),A
112	70	MOV M,B	LD (HL),B
113	71	MOV M,C	LD (HL),C
114	72	MOV M,D	LD (HL),D
115	73	MOV M,E	LD (HL),E
116	74	MOV M,H	LD (HL),H
117	75	MOV M,L	LD (HL),L

In addition to data transfers in which the source is a register or memory location, it is possible to load into a register or memory location from the opcode itself. The eight-bit value to be sent to the destination is simply the second byte of a two-byte opcode. The data

transfer is often called *immediate*, a term which is meant to convey that the byte comes directly (*immediately*) from program memory. The 8080 mnemonic "MVI" stands for *move immediate*. These instructions appear in table 2.4. In each case the value loaded may be any eight-bit value. Here and throughout the chapter the expression "FF" will be used to represent any eight-bit value, and "FFFF" will be used to represent any sixteen-bit value. Recall that when assembled into machine code, a value such as 7F34 becomes 347F in the opcode.

This instruction is analogous to a numerical constant in a BASIC program. Each instruction listed in the table requires seven clock cycles, except the load to (HL), which requires ten cycles.

Table 2.4. Eight-bit immediate load

Decimal	Hex	8080 Mnemonic	Z80 Mnemonic
62	3E FF	MVI A,FF	LD A,FF
6	06 FF	MVI B,FF	LD B,FF
14	0E FF	MVI C,FF	LD C,FF
22	16 FF	MVI D,FF	LD D,FF
30	1E FF	MVI E,FF	LD E,FF
38	26 FF	MVI H,FF	LD H,FF
46	2E FF	MVI L,FF	LD L,FF
54	36 FF	MVI M,FF	LD (HL),FF

Other Eight-bit Moves

All other eight-bit data transfers are limited to the accumulator as either source or destination. These transfers are listed in table 2.5.

The LDA and STA instructions are an example of so-called *direct addressing*, in which the memory location involved in the transfer is indicated by the second and third bytes of a three-byte opcode. The address comes *directly* from the program being executed. Each takes thirteen clock cycles.

The STAX and LDAX instructions allow data transfer to and from the memory location pointed to by the BC or DE register pair. This is register indirect addressing. Each instruction requires seven clock cycles.

The IN and OUT instructions cause an eight-bit word to be transferred between the accumulator and an input or output port.

These instructions are discussed at length in chapter 5. The IN and OUT instructions also constitute direct addressing, because the port address involved in the data transfer is determined directly by the program. Each instruction requires ten cycles.

Table 2.5. Other eight-bit data transfers

Decimal	Hex	8080 Mnemonic	Z80 Mnemonic
58	3A FF FF	LDA FFFF	LD A,(FFFF)
10	0A	LDAX B	LD A,(BC)
26	1A	LDAX D	LD A,(DE)
50	32 FF FF	STA FFFF	LD (FFFF),A
2	02	STAX B	LD (BC),A
18	12	STAX D	LD (DE),A
219	DB FF	IN FF	IN A,(FF)
211	D3 FF	OUT FF	OUT (FF),A

Sixteen-bit Data Transfers

The 8085 instruction set also allows sixteen-bit data transfers, as listed in table 2.6. The LHLD and SHLD instructions, for load HL direct and store HL direct, transfer two bytes of information to or from two adjacent locations in memory. The second and third bytes of the three-byte instruction determine the memory address to or from which L is loaded. The H register is loaded to or from the next higher memory address. The addressing is direct because the memory locations involved are specified directly by the program. Each instruction requires sixteen clock cycles.

The LXI (for load extended immediate) allow the loading of a sixteen-bit value from the program to a register pair. Ten clock cycles are required.

Table 2.6. Sixteen-bit data transfers

Decimal	Hex	8080 Mnemonic	Z80 Mnemonic
42	2A FF FF	LHLD FFFF	LD HL,(FFFF)
34	22 FF FF	SHLD FFFF	LD (FFFF),HL
1	01 FF FF	LXI B,FFFF	LD BC,FFFF
17	11 FF FF	LXI D,FFFF	LD DE,FFFF
33	21 FF FF	LXI H,FFFF	LD HL,FFFF
49	31 FF FF	LXI SP,FFFF	LD SP,FFFF

One lone instruction is available for an exchange of registers: the XCHG or EX DE, HL, with decimal value 235 and hex value EB. It exchanges the contents of the HL and DE registers. This instruction, by far the briefest of the sixteen-bit transfers, requires only four clock cycles.

Stack Operations

When you write a machine-language program, the most frequent reminders of the existence of the stack are the PUSH and POP instructions. The stack is also affected during subroutine CALLs and RETurns. Be sure there is always a POP for every PUSH, and vice versa.

When a PUSH is executed, the contents of the specified register pair are placed on the stack.

The contents of the high-order register of the pair (A, B, D, or H) are moved to the memory location one lower than the location pointed to by SP. The contents of the lower-order register are moved to the memory location two positions below the location pointed to by SP. The contents of SP are decremented by two. Twelve clock cycles are required to execute a PUSH.

The POP instruction undoes the action of the PUSH instruction. Ten clock cycles are required. The stack-related opcodes are listed in table 2.7.

The PUSH and POP instructions provide the only means of loading the entire flag register. For example, PUSH AF followed by POP BC moves the flag register contents into the C register and in the process, destroys whatever was previously stored in the B and C registers.

The 8080 mnemonic uses the rather cryptic abbreviation PSW (processor status word) to refer to the AF register pair.

The SPHL instruction moves the contents of HL to the SP register. Six clock cycles are required.

The XTHL instruction exchanges the contents of the HL register with the location pointed to by SP. More specifically, the contents of L are exchanged with the contents of the memory location pointed to by SP, and the contents of H are exchanged with the contents of the memory location one address higher than that pointed to by SP. Sixteen clock cycles are required.

Table 2.7. Stack-related opcodes

Decimal	Hex	8080 Mnemonic	Z80 Mnemonic
193	C1	POP B	POP BC
209	D1	POP D	POP DE
225	E1	POP H	POP HL
241	F1	POP PSW	POP AF
197	C5	PUSH B	PUSH BC
213	D5	PUSH D	PUSH DE
229	E5	PUSH H	PUSH HL
245	F5	PUSH PSW	PUSH AF
249	F9	SPHL	LD SP,HL
227	E3	XTHL	EX (SP),HL

Branch Instructions

A variety of instructions are available to transfer control; each causes the PC to do something other than simply increment.

The JMP (jump) instruction loads a new value into the PC. The conditional jump instructions test the condition of one or more flags and load a new value into the PC only if the condition is satisfied. In each case, the new PC value is contained in the second and third bytes of a three-byte opcode. This is shown by the value FFFF in table 2.8. Each jump requires ten cycles, unless a conditional jump fails to satisfy its condition, in which case, seven cycles are required.

The flag requirements for the various conditions are given in table 2.8.

Table 2.8. Jump instructions

Decimal	Hex	8080 Mnemonic	Z80 Mnemonic
195	C3 FF FF	JMP FFFF	JP FFFF
218	DA FF FF	JC FFFF	JP C,FFFF
210	D2 FF FF	JNC FFFF	JP NC,FFFF
250	FA FF FF	JM FFFF	JP M,FFFF
242	F2 FF FF	JP FFFF	JP P,FFFF
234	EA FF FF	JPE FFFF	JP PE,FFFF
226	E2 FF FF	JPO FFFF	JP PO,FFFF
202	CA FF FF	JZ FFFF	JP Z,FFFF
194	C2 FF FF	JNZ FFFF	JP NZ,FFFF

Table 2.9. Conditional execution

Condition	Flag Contents to Satisfy the Condition
C-carry	CY=1
NC-no carry	CY=0
P-positive	S=0
M-minus	S=1
PE-parity even	P=0
PO-parity odd	P=1
Z-zero	Z=1
NZ-nonzero	Z=0

Subroutine Calls

The call and return instructions share with the jump instructions the ability to change the PC and thus to redirect the flow of program execution. Just as in a BASIC subroutine call, the 8085 CALL instruction transfers control to a specified address. The place where the call occurred is noted, so that when the subroutine finishes (returns), control can be returned there.

When a CALL opcode is encountered, the PC is incremented so that it points to the next executable instruction following the opcode that was the CALL. The PC value is placed on the stack just as if it were PUSHed there. The second and third bytes of the CALL instruction are treated as an address and placed in the PC. Execution continues based on the PC contents.

Later, when a RET instruction is encountered, the stack is "popped" and the sixteen-bit value on the stack is placed in the PC. Execution continues based on the PC contents. The call and return require eighteen and ten cycles, respectively.

The 8085 subroutine instructions use the stack as defined by the SP, just as the POP and PUSH instructions do. As a result, programming errors can occur causing the return instruction to load a meaningless value into the PC. The 8085 may start executing code that is not even a program. At best the program will not execute properly; at worst everything in RAM may be lost, and you will find yourself back at January 1, 1900.

Two precautions will keep you out of trouble. Be sure the number of PUSHes and POPs encountered in the execution of the subroutine are always equal, regardless of any internal branching, and never tamper with the SP register.

Conditional Calls and Returns

The 8085 recognizes conditional subroutine calls and returns. The conditions that can be tested for are the same as the conditions listed in table 2.9. If the condition fails, execution proceeds to the next instruction, much the same as in the case of a conditional jump.

A conditional call requires eighteen clock cycles if the condition is satisfied, and nine cycles otherwise. A conditional return requires twelve cycles if the condition is satisfied, and six cycles otherwise.

The subroutine opcodes are listed in table 2.10.

Table 2.10. Subroutine opcodes

Decimal	Hex	8080 Mnemonic	Z80 Mnemonic
205	CD FF FF	CALL FFFF	CALL FFFF
220	DC FF FF	CC FFFF	CALL C,FFFF
212	D4 FF FF	CNC FFFF	CALL NC,FFFF
252	FC FF FF	CM FFFF	CALL M,FFFF
244	F4 FF FF	CP FFFF	CALL P,FFFF
236	EC FF FF	CPE FFFF	CALL PE,FFFF
228	E4 FF FF	CPO FFFF	CALL PO,FFFF
204	CC FF FF	CZ FFFF	CALL Z,FFFF
196	C4 FF FF	CNZ FFFF	CALL NZ,FFFF
201	C9	RET	RET
216	D8	RC	RET C
208	D0	RNC	RET NC
248	F8	RM	RET M
240	F0	RP	RET P
232	E8	RPE	RET PE
224	E0	RPO	RET PO
200	C8	RZ	RET Z
192	C0	RNZ	RET NZ

continued on following page

Restart Instructions

The 8085 also responds to eight "call" opcodes each of which is a single byte long, one-third the length of the usual subroutine call. These instructions are listed in table 2.11, and require twelve clock cycles each to execute.

Table 2.11. Restart instructions

Decimal	Hex	8080 Mnemonic	Z80 Mnemonic	Location of Called Subroutine
199	C7	RST 0	RST 00	0000
207	CF	RST 1	RST 08	0008
215	D7	RST 2	RST 10	0010
223	DF	RST 3	RST 18	0018
231	E7	RST 4	RST 20	0020
239	EF	RST 5	RST 28	0028
247	F7	RST 6	RST 30	0030
255	FF	RST 7	RST 38	0038

Restart instructions allow the eight most frequently used subroutines in ROM to be assigned to these one-byte opcodes. This saves program space every time the routine is called. In a machine other than a Model 100, the restart instructions allow an interrupting device to "jam" a single byte onto the data bus in such a way that the single byte

can determine which of several routines will be used to handle the interrupt. (The latter process is described further in chapter 15.)

These one-byte subroutine calls, called RST (for restart) instructions, are dedicated to eight predetermined addresses near 0000H. Because that port of address space is in ROM and thus cannot be changed by the user, the user cannot change what happens when one of the RST instructions is executed.

Nevertheless, the ROM code associated with the RST instructions may be put to use; the ROM functions are listed in table 2.12.

Table 2.12. Model 100 restart routines

Mnemonic	Jumps to	Function
RST 00	7D33	Same as RESET
RST 08		Find any BASIC special character
RST 10	0858	Find next BASIC character
RST 18		Compares DE and HL
RST 20	4B44	Sends character to LCD or PRT
RST 28	1069	Determines variable type
RST 30	33DC	Returns sign of first floating point accumulator
RST 38	7FD6	Indexed jump (see text)

The RST 20 instruction is handy. It sends a character to the screen or printer depending on the condition of an output flag in RAM (*see* chapters 10 and 13).

The RST 18 routine compares the DE and HL registers. Upon return from the routine, the Z flag is set if the registers are identical, and reset otherwise. To see how this is done, disassemble the code at 0018 to 001D.

The RST 00 instruction accomplishes the same thing as a jump to 0000, although in fewer bytes. RST 00 causes an initialization of the kind that occurs when the RESET button is pressed or the power to the Model 100 is turned on.

The RST 38 instruction is best thought of as a two-byte instruction. The byte following the RST is used as an offset. It points to a two-byte address in a table located in RAM at addresses FADA through FB39, and that address is jumped to. This may be compared with the ON ... GOTO command in BASIC.

The RST 38 and RAM table are designed to allow for expansion. Many BASIC commands contain an RST 38, and in each case, the address in the RAM table returns (in cases where an existing Model 100 feature may change) or generates an FC error (in cases where a feature is to be added). Particular RST 38 applications are described in later chapters.

If you want to test your understanding of the stack to the fullest, disassemble and study the code at 7FD6 through 7FF3, which accomplishes the vectored jump of the RST 38 subroutine.

The RST 08 instruction can also be thought of as a two-byte instruction. The byte following the RST 08 opcode is treated as an ASCII character or BASIC token. It is compared with the next character in a BASIC program line. One odd occurrence is the use of the exchange instruction at 0009. The exchange is performed between HL and the memory location pointed to by SP, but usually that memory location is in ROM. The instruction is essentially used as a one-way data transfer, since you cannot successfully load data into ROM.

The RST 10 routine is used in parsing a BASIC program line. It locates the next significant character, ignoring spaces, tabs, and the like.

The RST 28 and 30 routines are used in handling variables. RST 28 determines the variable type, and RST 30 returns the sign of the value in the first floating-point accumulator.

Another Jump Instruction

The 8085 CPU recognizes a jump instruction that might be thought of as a sixteen-bit register load, since it loads the contents of the HL register into the PC. It is analogous to the BASIC ON ... GOTO command. The decimal value for the opcode is 233 (hex E9). The 8080 mnemonic is PCHL, and the Z80 mnemonic is JP (HL).

Arithmetic Functions

The 8085, like all microprocessors, has rather limited arithmetic capabilities. It can add or subtract, if the numbers are not too large, and it can multiply and divide, if the multiplier or divisor is a power of two. Anything else must be performed as a combination of the above instructions.

The arithmetic and logic functions of the 8085 set and reset the various flags in the F register. The first and most straightforward of the arithmetic functions is addition. Several kinds are provided for. In each case, one of the two numbers to be added is placed in the accumulator. After the instruction has been executed, the result is found in the accumulator.

The other of the two numbers to be added can be found in another register (register direct addressing), somewhere in memory (register indirect addressing), or in the latter part of a two-byte instruction (immediate). The available opcodes are listed in table 2.13.

Table 2.13. Eight-bit Add instructions

Decimal	Hex	8080 Mnemonic	Z80 Mnemonic
135	87	ADD A	ADD A,A
128	80	ADD B	ADD A,B
129	81	ADD C	ADD A,C
130	82	ADD D	ADD A,D
131	83	ADD E	ADD A,E
132	84	ADD H	ADD A,H
133	85	ADD L	ADD A,L
134	86	ADD M	ADD A,(HL)
198	C6 FF	ADI FF	ADD A,FF

The ADD (and its relatives, the subtract, add-with-carry, and subtract-with-borrow) direct instructions require four cycles, while the indirect and immediate instructions require seven cycles. In each case, all of the flags in the F register are updated, based on the final condition of the accumulator.

The Carry (C or CY) and Auxiliary Carry (AC) flags deserve special explanation.

When two numbers are added, the sum is often too large to fit into the accumulator. When this happens, the carry (CY) flag is set, and a properly written program checks to see if a carry occurred and responds accordingly.

Similarly, when dealing with BCD (binary-coded decimal, explained in chapter 11) numbers, you may want to know whether the result is too large to fit in four bits. If a carry occurred from bit 3 to bit 4, the auxiliary carry flag (also sometimes known as the half-carry), is set. It is reset otherwise.

The AC flag cannot be used as the condition of a jump or call. It is used by the DAA instruction.

Addition with Carry

When numbers that do not fit into a single accumulator are being manipulated, you must break them into parts and treat them separately. When two such numbers are being summed, you add from right to left, just as you were taught in grade school.

When the rightmost part is added, a carry may occur. You need a way to add that carry to the part of the number that has not yet been summed (the part to the left of the part that has been summed). The 8085 instruction set specifically provides for this with the "add-with-carry" opcodes, which are listed in table 2.14. The add-with-carry instructions are identical to the ADD instructions in all respects except for the handling of the carry flag.

Table 2.14. Eight-bit Add-with-carry

Decimal	Hex	8080 Mnemonic	Z80 Mnemonic
143	8F	ADC A	ADC A,A
136	88	ADC B	ADC A,B
137	89	ADC C	ADC A,C
138	8A	ADC D	ADC A,D
139	8B	ADC E	ADC A,E
140	8C	ADC H	ADC A,H
141	8D	ADC L	ADC A,L
142	8E	ADC M	ADC A,(HL)
206	CE FF	ACI FF	ADC A,FF

Subtraction

The subtract and subtract-with-borrow instructions work the same way as the add and add-with-carry instructions. The two carry flags have analogous meanings. In the case of subtraction, the CY flag indicates whether a *borrow* was attempted from bit 8. This happens when a larger number is subtracted from a smaller one. The AC flag indicates whether a borrow was made from bit 4. The subtract and subtract-with-borrow instructions appear in table 2.15. In the case of the subtract-with-borrow, the borrow flag is subtracted from the accumulator as part of the subtraction process.

Table 2.15. Subtraction instructions

Decimal	Hex	8080 Mnemonic	Z80 Mnemonic
151	97	SUB A	SUB A,A
144	90	SUB B	SUB A,B
145	91	SUB C	SUB A,C
146	92	SUB D	SUB A,D
147	93	SUB E	SUB A,E
148	94	SUB H	SUB A,H
149	95	SUB L	SUB A,L
150	96	SUB M	SUB (HL)
214	D6 FF	SUI FF	SUB FF
159	9F	SBB A	SBC A,A
152	98	SBB B	SBC A,B
153	99	SBB C	SBC A,C
154	9A	SBB D	SBC A,D
155	9B	SBB E	SBC A,E
156	9C	SBB H	SBC A,H
157	9D	SBB L	SBC A,L
158	9E	SBB M	SBC A,(HL)
222	DE FF	SBI FF	SBC FF

In table 2.15, the 8080 mnemonics SUI and SBI stand for *subtract immediate* and *subtract-with-borrow immediate*, while the Z80 mnemonic SBC stands for *subtract-with-carry*.

Eight-bit increments and decrements

The 8085 can be instructed to increment (increase by one) or decrement (decrease by one) an eight-bit register or the contents of a memory address. All condition flags except CY are affected. In the case of the registers, the instruction requires four cycles, while the memory increment or decrement requires ten cycles. These instructions are listed in table 2.16.

Table 2.16. Eight-bit increments and decrements

Decimal	Hex	8080 Mnemonic	Z80 Mnemonic
60	3C	INR A	INC A
4	04	INR B	INC B
12	OC	INR C	INC C
20	14	INR D	INC D
28	1C	INR E	INC E
36	24	INR H	INC H
44	2C	INR L	INC L
52	34	INR M	INC (HL)
61	3D	DCR A	DEC A
5	05	DCR B	DEC B
13	OD	DCR C	DEC C
21	15	DCR D	DEC D
29	1D	DCR E	DEC E
37	25	DCR H	DEC H
45	2D	DCR L	DEC L
53	35	DCR M	DEC (HL)

Binary-Coded Decimal Operations

One arithmetic instruction is used solely for binary-coded decimal operations. After addition has taken place, it can be used to clean up the accumulator so that neither the upper half nor the lower half of the accumulator contains the BCD equivalent of a number greater than nine.

This is what happens when the DAA (decimal adjust accumulator) instruction is executed. If the low-order four bits constitute ten or larger or if the AC flag was set previously, the value six is added to the accumulator. This may cause a carry into bit 4.

If the high-order four bits have a value of ten or more or if the CY flag was set previously, the value six is added to the high-order four bits of the accumulator. This may cause a carry into bit 8 to occur.

This opcode has a value of 39 decimal, or 27 hex, and requires four clock cycles. All condition flags are affected. The opcode is used in six different places in the ROM BASIC arithmetic routines, between 2C34 and 2E40.

Sixteen-bit Arithmetic

The sixteen-bit capability of the 8085 is quite limited. Addition can be performed, and the HL register is used as the accumulator. Prior to the summation, one of the addends is located in the HL register, and the other is located in another register pair. When the addition is finished, the sum is completed in the HL register.

Of the various condition flags, only the CY flag is affected by sixteen-bit arithmetic. It is set if the sum yields a carry at bit 15. The instruction requires ten clock cycles. The instructions are listed in table 2.17. The mnemonic DAD stands for *double add.*

Table 2.17. Sixteen-bit addition

Decimal	Hex	8080 Mnemonic	Z80 Mnemonic
9	09	DAD B	ADD HL,BC
25	19	DAD D	ADD HL,DE
41	29	DAD HL	ADD HL,HL
57	39	DAD SP	ADD HL,SP

Sixteen-bit increment and decrement instructions are also available. These are listed in table 2.18. Each requires six clock cycles; no condition flags are affected. The mnemonics INX and DCX stand for *increment extended* and *decrement extended* respectively.

Table 2.18. Sixteen-bit increments and decrements

Decimal	Hex	8080 Mnemonic	Z80 Mnemonic
3	03	INX B	INC BC
19	13	INX D	INC DE
35	23	INX H	INC HL
51	33	INX SP	INC SP
11	0B	DCX B	DEC BC
27	1B	DCX D	DEC DE
43	2B	DCX H	DEC HL
59	3B	DCX SP	DEC SP

Logical Operators

The 8085 is designed to perform logical operations on eight-bit values. This is useful not only in calculations but also as a way of setting a particular bit equal to 1 or 0.

The AND, OR, and XOR instructions are listed in table 2.19. In each case, one number is placed in the accumulator, the operation is performed, and the result is found in the accumulator. The operations take place exactly as described on page 111 of the Model 100 Owner's Manual. The other number can be found in another register. It can be a value found at a memory address, or it can be a constant stored as the second byte of a two-byte instruction (immediate).

The immediate and memory operations require seven clock cycles, and the register operations require four cycles.

Table 2.19. AND, OR, and XOR operations

Decimal	Hex	8080 Mnemonic	Z80 Mnemonic
167	A7	ANA A	AND A
160	A0	ANA B	AND B
161	A1	ANA C	AND C
162	A2	ANA D	AND D
163	A3	ANA E	AND E
164	A4	ANA H	AND H

continued on following page

Decimal	Hex	8080 Mnemonic	Z80 Mnemonic
165	A5	ANA L	AND L
166	A6	ANA M	AND (HL)
230	E6 FF	ANI FF	AND FF
183	B7	ORA A	OR A
176	B0	ORA B	OR B
177	B1	ORA C	OR C
178	B2	ORA D	OR D
179	B3	ORA E	OR E
180	B4	ORA H	OR H
181	B5	ORA L	OR L
182	B6	ORA M	OR (HL)
246	F6 FF	ORI FF	OR FF
175	AF	XRA A	XOR A
168	A8	XRA B	XOR B
169	A9	XRA C	XOR C
170	AA	XRA D	XOR D
171	AB	XRA E	XOR E
172	AC	XRA H	XOR H
173	AD	XRA L	XOR L
174	AE	XRA M	XOR (HL)
238	EE FF	XRI FF	XOR FF

With the AND, OR, and XOR instructions, the carry (CY) flag is cleared. The OR and XOR operations clear the AC flag as well.

The manner in which the AND instruction handles the AC flag differs between the 8080 and the 8085. This is one of the few substantive differences between the two CPUs that might make an 8080 program run incorrectly on an 8085. In the case of the 8085, the AND operation simply turns the AC flag on. The 8080, however, sets it equal to bit 3 of the result of the AND operation in the accumulator.

TURNING A BIT ON

For example, to turn on bit 2 of the accumulator, use the OR instruction with a value of two to the power of two, namely 4.

TURNING A BIT OFF

To turn off bit 3 of the accumulator, use the AND instruction with a value that has all bits on except bit 3, namely 11110111, or FC hex.

TESTING A REGISTER PAIR FOR ZERO

Often you want to know whether the value in a register pair, such as BC, has been decremented to zero. The easiest way to do this is to load B into the accumulator, and OR it with the C register.

COMPARISON OPERATIONS

Sometimes you want to know whether two values are equal, but the algebraic difference of the two is not required. The compare instruction determines the relation between two values. It sets the condition flags at the values they would have if the two numbers had been subtracted. The value in the accumulator is unchanged as a result of the operation.

In particular, the Z flag is set at 1 if the two compared values are equal. The carry (CY) flag is set at 1 if the accumulator contents are less than the other value. The latter result is obtained because if the operation had been a subtraction and if a larger value were subtracted from the accumulator, a borrow would have occurred. The compare instructions are listed in table 2.20.

Table 2.20. Compare instructions

Decimal	Hex	8080 Mnemonic	Z80 Mnemonic
191	BF	CMP A	CP A
184	B8	CMP B	CP B
185	B9	CMP C	CP C
186	BA	CMP D	CP D
187	BB	CMP E	CP E
188	BC	CMP H	CP H
189	BD	CMP L	CP L
190	BE	CMP M	CP (HL)
254	FE FF	CPI FF	CP FF

Rotate Instructions

The rotate instructions are easier to illustrate than to describe (*see* figure 2.2.) A rotate instruction shifts the entire contents of the accumulator one position to the left or right, leaving an open bit at the empty end of the accumulator. In the RLCA and RRCA instructions, what goes out one end of the accumulator comes back in the other end and goes to the carry. In the RLA and RRA instructions, what goes out one end of the accumulator ends up in the carry, while the contents of the carry come in the empty end of the accumulator. An RLA followed by an RRA leaves everthing as before, with no loss of information. In contrast, either of the RLCA and RRCA instructions destroy the former contents of the carry flag. The opcodes and mnemonics are listed in table 2.21.

Table 2.21. Rotate instructions

Decimal	Hex	8080 Mnemonic	Z80 Mnemonic
23	17	RAL	RLA
31	1F	RAR	RRA
7	07	RLC	RLCA
15	0F	RRC	RRCA

Each instruction requires four clock cycles, and in each case only the carry flag is affected.

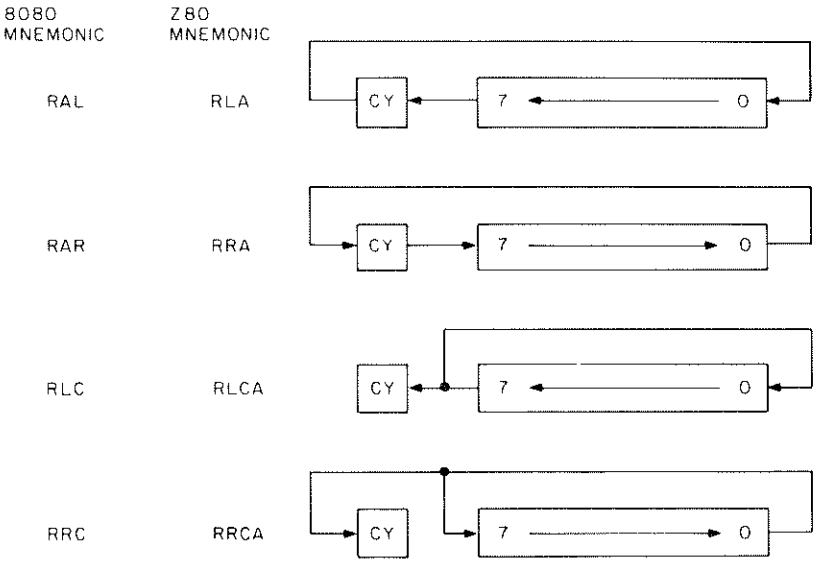

Figure 2.2. Rotate instructions

USES FOR THE ROTATE INSTRUCTIONS

Rotation to the left or right is comparable to multiplying or dividing by two, respectively.

If a jump that is conditional on either bit 7 or 0 of the accumulator is desired, the most economical way to program it is to rotate the bit of interest into the carry, and then jump conditional on the carry flag.

When serial information is being sent to the CPU, whether as an input on the SID pin (*see* chapter 12), or through an I/O port (*see* chapter 11), the rotate instructions provide a handy way to reconstruct the byte:

- rotate the recently received bit into the carry
- load the partial byte (from the previous bits received) into the accumulator
- rotate the carry bit into the accumulator
- repeat the process until eight bytes have been loaded.

A similar process may be used to send out serial data from the CPU, as discussed in chapters 11 and 12.

Other Logical Instructions

The *accumulator complement* instruction turns off each bit of the accumulator that is on, and turns on each bit that is off. This is also known as a *ones complement*. No condition flags are affected. The operation requires four clock cycles.

The opcode and mnemonics are shown in table 2.22.

Table 2.22. Other logical operations

Decimal	Hex	8080 Mnemonic	Z80 Mnemonic	Meaning
55	37	STC	SCF	set carry flag
63	3F	CMC	CCF	complement carry flag
47	2F	CMA	CPL	complement accumulator

Because of the way negative numbers are treated in addition and subtraction, simply complementing the accumulator is not the way to change the sign on an integer stored there. Instead a *two's complement* must be performed. A two's complement consists of the performance of a one's complement followed by the addition of 1.

The Carry Flag

The carry flag may be turned on by means of the *set carry flag* instruction or may be complemented by the *complement carry flag* instruction. The only condition flag affected is the carry flag. Each operation requires four clock cycles.

Machine Control Instructions.

The HALT instruction stops the processor. This is used in ROM as part of the power-down sequence at 1431-1458, and also appears enigmatically in the TEXT program at 6AC3.

The only way to overcome a halt is by a RESET or by turning the power off and on again.

NOP stands for *no operation* and is just that -- an instruction which does nothing except use up four clock cycles. No flags are affected. Sometimes a NOP is placed in RAM to allow new instructions to be inserted later without the need to reassemble the program.

Table 2.23. Machine control opcodes

Decimal	Hex	8080 Mnemonic	Z80 Mnemonic
243	F3	DI	DI
251	FB	EI	EI
118	76	HLT	HALT
0	00	NOP	NOP
32	20	RIM	(none)
48	30	SIM	(none)

The EI and DI instructions are used to enable and disable interrupts, respectively. This is discussed in detail in chapter 15. Interrupts are disabled when a time-sensitive process is to be performed or when a particular sequence of loading registers or output ports must not be interfered with.

The RIM (read interrupt mask) and SIM (set interrupt mask) instructions are used to perform cassette input and output (*see* chapter 12) and to mask selected interrupts (*see* chapter 15).

3

Advanced BASIC

There are a number of BASIC commands and functions which, though only briefly explained in the owner's manual, must be fully understood before one can begin machine language programming with the Model 100. These are covered in detail in this chapter.

Recall from the discussion in chapter 1 that the Model 100 architecture allows the 8085 CPU to communicate with the integrated circuits around it by means of I/O ports (of which 18 are implemented out of a possible 256) and memory locations, all of which are implemented.

In assembly language the I/O ports are accessed by means of IN and OUT opcodes, while memory locations are accessed by means of load, store, and move opcodes. In BASIC the I/O ports are accessed by means of the INP function and OUT command, while the memory

locations are accessed by means of the PEEK function and POKE command. This is summarized in table 3.1.

Table 3.1. Machine language and BASIC data transfer

Access to	Direction	BASIC Method	Typical Machine Language 8080	Z80
Input port	To CPU	INP(n)	IN n	IN A,(n)
Output port	From CPU	OUT n,d	OUT n	OUT (n),A
ROM,RAM	to CPU	PEEK(n)	MOV r,M LDA n LDAX	LD A,(n)
RAM	from CPU	POKE n,d	MOV M,r STA n STAX n	LD (n),A

In addition to the differences in terminology between assembly language and BASIC operations, there is a syntactical difference. In assembly language an opcode is an opcode. In BASIC, however, some reserved words (like PEEK and INP) denote functions, forming part or all of an expression, and thus never follow a line number or colon; while others (like POKE and OUT) are functions and always follow a line number or a colon.

PEEK

The PEEK function is executed with one argument, the memory address, which is an integer in the range 0 to 65535. The value returned by the function is the contents of that memory location, an integer in the range 0 to 255. PEEKs are always harmless -- executing a PEEK will not alter the memory contents.

The PEEK function returns an eight-bit integer, but often one wishes to use PEEKs to obtain a sixteen-bit integer (such as the HIMEM value, discussed below) from two adjacent memory addresses. Because the 8085 stores such integers with the high-order eight bits in the higher memory address, the proper way to combine the PEEK values is to multiply the higher PEEK by two to the eighth power, or 256.

For example, to return the sixteen-bit value at 64430, evaluate,

```
PEEK(64430)+256*PEEK(64431)
```

Many locations worth peeking to (or at) are listed in figure 18.4.

POKE

The POKE command has two operands. The first, an integer in the range 0 to 65535, specifies the memory address to which data is written, and the second, an integer in the range 0 to 255, is the data value to be written.

Although the BASIC interpreter allows POKEs to locations below 32768, such pokes have no effect, because those locations are in ROM. (If the standard ROM were switched out, as described in chapter 5, and if RAM were put in its place, then a POKE to that part of memory would accomplish something.)

Using POKEs blindly can be dangerous. Many pointers, flags, and the like, stored in the area 62960-65535 are essential to the operating system, and if changed indiscriminantly, can cause destruction of user files. This destruction may not occur until several days later, when a particular file or routine is accessed.

Some values in the 62960-65535 area, however, such as the SOUND flag, may be freely changed by means of a POKE command. These *safe* addresses are discussed in later chapters in the context of particular Model 100 functions.

Even if POKEs are confined to RAM below 62960, problems can still arise. A BA file, for instance, contains two-byte addresses for succeeding BASIC line numbers. If a line number address is changed, the BASIC interpreter's reaction when that program is run (which might be several days later) is unpredictable.

DO files are somewhat safer. As long as one stays away from the beginning and end-of-file bytes, one may change values freely. The worst that can happen is that the file, when later viewed through TEXT or OPENed in BASIC, will be found to have the wrong contents.

CO files contain load addresses which, if tampered with, may cause a later LOADM or RUNM to yield unpredictable addresses.

Often one wishes to store a sixteen-bit value in RAM. This is accomplished in a manner similar to a sixteen-bit PEEK. The low-order eight bits are stored to the named location, and the high-order eight bits are stored to the next higher memory location.

For example, to store the value 62700 to memory address 62964, one must separate the low-order and high-order parts of the integer 62700. In principle this could be accomplished as follows:

```
high-order part is 62700 AND (255*256)
low-order part is 62700 AND 255
```

Unfortunately, BASIC will not perform AND, OR and XOR operations on integers larger than 32767. One way to avoid this problem is to test the desired value, say 62700, and if it is bigger than 32767, simply subtract 65536 from it, yielding in this case -2836. Because BASIC does operate on integers between -32767 and 0, they may be used.

Thus the high-order part is (-2836 AND (255*256)), and the low-order part is (-2836 AND 255). The proper POKEs are:

```
POKE 62694, (62700-65536)/256
POKE 62965, (62700-65536) AND 255
```

Long division can also be used to separate the high and low-order parts. For an integer N, the high-order part H is H=INT(N/256), and the low-order part L is the remainder, namely L=N-(H*256).

Input Ports

The various active input ports are surveyed in chapter 5, and discussed in detail in later chapters relating to particular devices. The INP function has one argument, the port number, which is an integer in the range 0 to 255. The value returned by the function is the data from that input port.

When an input is attempted from a port that has not been implemented in hardware, the value returned turns out to be the number of the port. This happens because of the shared address and data lines in the 8085.

Use of the INP function, like the PEEK function, is always harmless — it is impossible to hurt anything.

OUT

The OUT command, like the POKE command, has two arguments. It uses the second argument (an integer between 0 and 255) as the outgoing data. The first argument, also an integer in the range 0 to 255, is the port number to which the data is to be sent. The various output port numbers implemented in the Model 100 are surveyed in chapter 5 and discussed in detail in later chapters.

OUT commands can be dangerous, though not, perhaps, as dangerous as POKEs. Sending the wrong value to an output port can select the option ROM when you don't want it (any port in the range 224-239) or can cut off power in a disorderly way (port 178 or 186). Output to other port numbers is generally harmless, although this may, for example, upset the UART settings.

ALLOCATING RAM — THE MAXRAM AND HIMEM VALUES

As shown in figure 1.2, RAM is partitioned into areas with distinct uses. The lowest RAM address (32768 decimal or 8000 hex in a 32K RAM model) is stored in RAM at 64192. Recall from our earlier discussion that this value may be determined by evaluating:

```
PEEK(64192)+256*PEEK(64193)
```

BA files are placed starting at the lowest RAM address. The top address for the last BA file is stored at 64430. This value, and all the other "top addresses" discussed below, are updated every time a BA program changes in size, is created through the SAVE command, or killed. DO files are placed above BA files in RAM. The top of the DO files is pointed to by 64432. That pointer, and other pointers described below, are updated when a DO file grows, shrinks, or is killed.

CO files are next with the top of the CO files pointed to by 64434. That pointer is updated when a CO file is created or destroyed.

The BASIC variables, string and numeric, are stored above the CO files. These are set whenever a BASIC program is being run.

Each block of data is "bumped upward" when more data is stored below. For instance, if a BASIC program writes data to a DO file,

several items move up in memory: the top boundary of the DO files, the entirety of the CO files, and the BASIC variable area.

As more and more data is stored, sooner or later, one is greeted with the "OM" error, which means out of memory. This is because the types of data described so far consume all of the space between 8000 hex (in a 32K machine) and HIMEM, the user-imposed limit on data storage.

HIMEM

HIMEM is a special BASIC variable which may form part of an expression (e.g. PRINT HIMEM-4) but cannot appear to the left of an equal sign. HIMEM is stored at 62964, and so may be changed by POKEing to 62964 and 62965. (*See* the preceding discussion of POKEing of sixteen-bit values).

The more conventional way to change HIMEM is using BASIC's CLEAR command.

CLEAR

The CLEAR command, when executed in BASIC, always clears string and numerical variables and closes all files. Any dimensioned variables lose their dimensioned status.

By convention the CLEAR command should appear at the beginning of a program. For example, run this program:

```
10 F=25:DIM G(100)
20 CLEAR
30 PRINT F:G(99)=45:PRINT G(99)
```

The value for F will be displayed as 0, not 25, as F was CLEARed in line 20. The attempt to set G(99) equal 45 will be met with a "BS" (bad subscript) error, as the CLEAR command's execution in line 30 caused G to lose its dimensioned status.

Because of the way BASIC handles variables, it needs to know ahead of time how much space will be needed for string variables. Usually BASIC reserves 256 bytes, but this value may be changed by providing an argument for the CLEAR command. (Values smaller than 256 can be specified. This can be helpful if additional numeric variable space is required.)

The unused, reserved string variable space can be determined by evaluating the function FRE(""). String constants do not occupy the string variable space as shown in the following program:

```
10 CLEAR:PRINT FRE (""):I$="GHJKGKJHG":
PRINT FRE (""):I$="GHJKGKJHG"+"":PRINT FRE("")
```

The first value printed will be 256, the number of bytes set aside for string variables. When the variable I$ is assigned a value, one might think doing so would consume some of the variable space. BASIC, however, simply stores a pointer to the place in the BA program (much lower in RAM) where the string constant assigned to I$ can be found. The free space is still 256.

When I$="GHJKGKJHG"+"" is executed, BASIC is forced to evaluate a string expression, and I$ must point to the result. The only place to store the result is in the string variable space. As a result the free space is diminished.

By executing the FRE function with a numeric argument, e.g. FRE(0) or FRE(4.4), the amount of unused BASIC numeric space can be determined.

The CLEAR command may also have a second argument, namely a user-selected value for HIMEM. As mentioned above, HIMEM sets an upper limit on how high in memory user data storage (BA, DO, and CO files, and BASIC variables) may expand. As a result, anything above the address stored in HIMEM will generally be undisturbed.

The largest permissible value stored in HIMEM is F5F0, the bottom address of the operating system RAM area. This value, F5F0, is available to BASIC in the special variable MAXRAM. MAXRAM, like HIMEM, cannot appear on the left side of an equals sign. If and when a disk operating system is installed on the Model 100, the value returned by MAXRAM will be smaller, probably E000 hex (57344 decimal).

The second argument of the CLEAR function is checked before BASIC changes HIMEM. If an integer larger than MAXRAM is

given, an "FC" (function call) error will result, while an attempt to set it lower than the top of BASIC variables will yield an "OM" (out of memory) error.

To remove any protection given earlier to high memory, simply type CLEAR 256,MAXRAM. This does not immediately destroy data in the protected area; it simply renders it vulnerable in the event that BA, DO, or CO files, or BASIC variables expand upward to fill that area.

SAVEM,CSAVEM

Four BASIC commands, CLOADM, CSAVEM, LOADM, and SAVEM have been provided to facilitate manipulation of machine-language programs.

Upon generating machine language instructions, the assembler places these in RAM, usually at the location where it is intended that the machine language program will run. In the Model 100, this is usually a location in RAM between HIMEM and MAXRAM. The block of memory containing the program has a starting and an ending address, and the program itself usually has a entry (or transfer) address.

(Although assemblers vary, usually the start address is set by an ORG or ORIGIN psuedo-opcode, and the entry address is set by an END psuedo-opcode.)

The SAVEM command may be used to store the program as a RAM file, freeing the protected RAM area for other uses. SAVEM's syntax is:

```
SAVEM strexp, stadd, endadd, tradd
```

The argument strexp is a string expression containing a filename preceded optionally by a device. The device may be CAS: or RAM:. If none is specified, RAM is assumed. The filename will have the extension ".CO" appended. The arguments stadd and endadd are integers specifying the starting point and ending point in RAM of the block to be stored in the CO file.

The CO file that results from the command contains not only the data from high RAM, but also information about where the data will be reloaded if the file is accessed using menu selection (discussed later) or LOADM. As discussed in chapter 18, the CO file contains the address to start reloading to (always the same as the start address when the SAVEM was performed); the size of the program (obtained by subtracting endadd from stadd); and the transfer address. This applies to both tape and RAM files.

Note that after a SAVEM to RAM, two copies of the machine language program now exist — one in high memory and one somewhere between the DO files and the BASIC variable space. If memory space is short, one may free up the high RAM by typing CLEAR 1,MAXRAM.

The CSAVEM command functions like the SAVEM command except that a device of CAS: is automatically prefixed to the filename.

LOADM AND CLOADM

Assuming a tape or RAM file with extension CO has been created, the machine language program may be reloaded to protected RAM by using the BASIC command LOADM. The syntax of the command is:

```
LOADM strexp
```

where strexp is a string expression containing a filename and optionally a device type of CAS: or RAM:. (If no device is specified, RAM: is assumed.)

First, the CO file, which may originate from tape or RAM, is opened. The proposed start address for reloading is compared with HIMEM. The file is loaded only if its start address is greater than HIMEM. The purpose of this check is to avoid damage to user files in the area between 8000 hex and HIMEM.

Before executing LOADM, then, it may be necessary to use the CLEAR command to reset HIMEM to a low enough value to accommodate the CO file.

If LOADM is executed rather than the program mode, the start, end, and entry addresses will be displayed on the screen.

It should be clear from this discussion that the BASIC commands SAVEM and LOADM only allow the program to be loaded to its original location.

The BASIC command CLOADM works exactly like LOADM except that a device type of CAS: is prefixed to the filename.

VARPTR

One unpublished aspect of the VARPTR function is that if a file number is used as the argument, the value returned is the address of the file control block. Further discussion would be beyond the scope of this book.

CALL

BASIC's CALL performs essentially like the machine language CALL. If and when the called routine returns, control is returned to the calling BASIC program.

The syntax of the CALL function is as follows,

```
CALL add,exp1,exp2
```

where add is required, but exp1 and exp2 are optional.

The argument add is an integer between 0 and 65535 and is the address called. Thus the called routine may be located in ROM or RAM. One would expect that the only RAM addresses called would be those in the area HIMEM and MAXRAM, since there is usually no other executable code in RAM. (There is machine code in CO files but it cannot be CALLed from BASIC as the jump addresses within the CO file do not correspond to the place in RAM where the CO file resides.)

The argument exp1 is an integer between 0 and 255, and is placed in the accumulator just prior to the call. The argument exp2 is an integer between -32768 and 65536, and is converted to a sixteen-bit integer and placed in the HL register-pair. (*See* the previous discussion of BASIC's treatment of sixteen-bit integers.)

Often one wishes to use HL as a pointer to a series of memory locations containing ASCII characters, for printing or other processing. This can be accomplished by placing the characters into a string variable and using the VARPTR function to determine exp2. For example if the string is F$, its location is the sixteen-bit integer at VARPTR(F$)+1. In other words, if B=VARPTR(F$), then exp2 should be PEEK(B+1)+256*peek(B+2).

Calls can be dangerous. The called routine is surely not as fully error-protected as BASIC itself, and data stored in RAM may be disturbed. Make frequent backups of user files until you are certain the called routine is safe as these may be destroyed.

The CALL command provides one of two easy ways of executing user-written machine language programs. Assuming the machine language program is located somewhere between HIMEM and MAXRAM, one may run a short BASIC program to CALL the machine language program. Often a program contains both a BASIC portion and a machine language portion. For instance the BASIC portion could be used to prompt to open a file, and the machine language portion could be used to search RAM or ROM for a certain value or read a cassette tape in non-standard format. The BASIC program first sets HIMEM; then it uses LOADM to place the machine program into high memory. Then, with one or more CALLs to one or more addresses in the machine language program, the "dirty work" is done.

MENU SELECTION OF CO FILES

The Model 100 operating system is designed to respond to menu selection not only of BA files (by running them) and DO files (by entering TEXT), but also of CO files. When the cursor is moved to a CO file name and ENTER is pushed, the CO file is loaded to high memory, assuming that the current vaue of HIMEM leaves enough room for the file. (If HIMEM is too high in value, the operating system will beep and return to the menu.) Any machine language program previously located in the protected high RAM area will be destroyed.

Assuming an entry or transfer address was specified when the CO file was created, CPU execution will proceed at that address.

For this reason menu selection of a CO file can be dangerous. Unless you are quite certain that the CO program behaves itself and does not tamper with the wrong parts of RAM memory, be prepared for loss of user files.

BATTLE OF THE CO FILES

Z80 programmers are accustomed to so-called relative jumps, which make it possible to write programs which will run no matter where in RAM they have been loaded. Unfortunately, the 8085 has no relative jump instructions in its instruction set. Each jump must contain the actual sixteen-bit address to which control will be transferred. Unless part of the machine language program actually resides at that location, the program will not function properly.

No problem arises when a BASIC program uses a single CO routine and no other, since the BASIC program may LOADM the CO file into high memory and use it, paying no attention to the previous contents.

POKING INTO PROTECTED MEMORY

BASIC itself can be used both to load and run a machine language program. Obviously this is practical only if the program to be run is small, because no one enjoys typing in long BASIC DATA statements.

But in situations where it is intended that a routine, perhaps a bar-code-reader device handler, will remain in high RAM indefinitely, there will be a conflict. The cleanest, but perhaps most troublesome, solution is to reassemble the other machine routine so that it may reside in RAM below the bar-code-reader driver, and protect both routines.

Any two companies offering a line of machine language software for the Model 100 will likely choose loading addresses designed to conflict with each other's loading addresses, so that whichever company makes the first sale to a customer will be likely to make all future sales. The customer may need to disassemble and reassemble such routines to change their load locations.

Program development is tedious since even the smallest program change necessitates recalculating and retyping numerous decimal integers.

Nonetheless, without access to an assembler, BASIC POKEs are really the only choice. Also, sometimes machine language is intended for just one little piece of a program whle BASIC is used to receive the user input or to format the output. Such a BASIC program must first use the CLEAR command to protect the portion of high memory where the machine language code will be placed. Then a FOR loop is used to load the values into memory. If desired, a CALL command may then be used to execute the machine language.

The following example demonstrates how to use BASIC POKEs to accomplish machine language programming. Note how similar the ADDRESS and SCHEDUL programs are — they scan the ADRS and NOTE files, respectively, and display or print records containing a particular word or phase. The entry addresses for the two programs may be found in the system file directory (described in chapter 18). These are 5B68 and 5B6F.

If you disassemble the code at 5B68-5B72, you will see that each entry point leads to 5B74, the "plain vanilla" program underlying the two. Prior to reaching 5B74, each entry point sets DE to point to an ASCII string of the name of the file to be scanned and sets the accumulator with a flag value to indicate whether the command prompt will be "Adrs:" or "Schd:".

Your first reaction might be to simply use the BASIC CALL command to call 5B74. Unfortunately the CALL command cannot be used to set the DE register, only the HL. Furthermore, to keep the stack in order, a jump to 5B74 must be executed rather than a call.

Thus the way to harness the ADDRESS/SCHEDL program for your own use is to set HL to point to the filename, and then execute the following machine code:

```
D1              POP DE
EB              EX DE, HL
C3    74 5B     JMP 5B74
```

The purpose of the first POP is to eliminate the return address from the CALL, because it will never be used. EX represents a simple method of loading the value in HL into the DE register. A jump to the ROM program follows.

After writing the assembly language program, you must assemble it. This can be done by hand using the opcode tables in the appendices. The decimal values appear in the CHR$ functions.

The rest of the program retrieves the user input, converts it to uppercase, and provides the pointer to the location of the opcodes for the CALL. The program in finished form follows:

```
20 P$=CHR$(209)+CHR$(235)+CHR$(195)+CHR$
(116)+CHR$(91):PRINT"Input file:
";:LINEINPUTF$:F$=LEFT$(F$,6)+".DO"+CHR$
(0):FORI=1TOLEN(F$):C=ASC(MID$(F$,I,1))
:IFC>96 AND C<123THENC=C-32
120 POKE64984+I,C:NEXT:A=VARPTR(P$)
:CALLPEEK (A+1) +256*PEEK(A+2),0,64985
```

4

Borrowing From Z80 Experience

Differences Between The Z80 and 80C85 Instruction Sets

Recall from the discussion in chapter 2 that the most machine level programming is accomplished by writing a program in human-readable *assembly mnemonics* which are processed by an assembler to yield hexadecimal, and ultimately binary, values which are stored in RAM or ROM and then executed by the CPU.

Of the 256 possible values which may be returned when the CPU fetches an eight-bit byte from memory, 244 have precisely the same result when executed by 8080, 8085 and Z80 processors. Two hundred of them comprise one-byte opcodes, eighteen serve as the first byte of two-byte opcodes, and twenty-six serve as the first byte of three-byte opcodes. These values are listed in appendix C, tables C.1, C.2, and C.3.

Twelve eight-bit values (08, 10, 18, 20, 28, 30, 38, CB, D9, DD, ED, and FD), however, are treated differently by the three processors. The behavior of the 8080 is undefined for all twelve; they may be thought of as *gaps* in the 8080 instruction set. (There are even more gaps, so defined, in the 8008 instruction set.)

The designers of the 8085 filled in two of the twelve gaps: 20H and 30H are one-byte instructions which load the interrupt mask register (present in the 8085 but not in the 8080 or Z80) to and from the accumulator (A) register. The behavior of the 8085 is undefined as to the remaining ten values; this is shown by the data entries in tables C.1, C.2, and C.3. Nonetheless, any program written for the 8080 will run on the 8085, since the permissible instructions for the 8085 merely expand upon, but do not change, the 8080 instruction set.

The designers of the Z80 filled in all twelve gaps in the 8080 instruction set, but filled them in differently than did the designers of the 8085. Thus, while any program written for the 8080 will run on the Z80, not every program written for the 8085 will run on the Z80. More to the point, for Model 100 owners as a general rule programs written for the Z80 will not run on the 8085.

Because of the close relationship between the Z80 and 8085 instruction sets, programming techniques and algorithms from Z80 machines are nonetheless helpful in writing and modifying programs for the Model 100. Indeed, many readers of this book first learned assembly language programming with the Z80 microprocessor, since it was used in the Radio Shack TRS-80 Model I, III, and IV computers. Thus it is instructive to devote some attention to the Z80 instruction set.

Let's compare, for example, the behavior of the 8085 and Z80 CPUs upon fetching the value 7EH in the course of executing a program. By referring to table C.1, you can see that a programmer accustomed to the 8085 would think of this as a MOV A,M instruction, while a Z80 programmer will consider it to be a LD A,(HL) instruction. The result in each case is the same — the contents of one of the 65536 memory locations available to the CPU (and pointed to by the 16-bit register HL) are moved (loaded) to the A register. The only difference is a semantic one — the assembly mnemonic read by an assembler is MOV A,M according to the conventions for mnemonics

set up by Intel when it introduced the 8008 and 8080 processors, or LDA, (HL) according to the conventions set up by Zilog when it introduced the Z80.

Likewise an analogy may be drawn between most of the Z80 and 8085 mnemonics. Table C.2 lists the 8085 mnemonics alphabetically with the corresponding Z80 mnemonic. Similarly, table C.3 lists the Z80 mnemonics alphabetically with the corresponding 8085 mnemonic.

As should be apparent from this discussion, those Z80 operations whose opcodes appear in the gaps of the 8080 instruction set do not appear in the 8085 instruction set. They would result in undefined behavior if executed by the 8085.

When converting a Z80 program for 8085 execution, it is not just a matter of finding a combination of 8085 instructions to replace each non-8085 Z80 instruction. This is because the Z80 contains several registers (IX, IY, AF', BC', DE', HL', refresh register R, and interrupt page register I) not present in the 8080 or 8085. The Z80 instructions manipulating these registers have no close substitute in the 8080 or 8085 instruction sets.

Many other non-8085 instructions, though, have relatively easy substitutes in the 8085. For example, in the 8085:

- There are no bit set or test instructions, though of course the equivalent can always be accomplished through the AND, OR, and rotate instructions.
- There are no relative jumps JR nor DJNZ. Many Z80 programs are location-independent through use of relative jumps, while any jump instruction for the 8085 must provide the absolute address.
- The various incremented and decremented loads and compares are missing. These include: LDI, LDIR, LDD, CPI, CPIR, CPD, and CPDR. These routines may all be performed piecemeal by combinations of simplier 8085 instructions.
- The NEG instruction, which performs a 2's complement on the A register and sets various flags, is not available. The former may be performed by taking the 1's complement CPL (CMA) and adding one, but it is important to realize the CPL instruction only sets the H and N flags.

- Two Z80 16-bit arithmetic instructions are missing: ADD HL,ss and SBC HL,ss. The ADD HL,ss (DAD) instruction is a close but not perfect substitute because it sets only the H,N and C flags.
- Rotate instructions (RLC, RL, RRC, RR) may be applied only to the A register. Thus if it is desired to rotate the contents of some other register, it will be necessary to load the register to the accumulator, rotate it, and load the contents back to the other register.
- The shift instructions of the Z80: SLA, SRA, and SRL, are not available. The accumulator rotate instructions must be used instead, masking the left and rightmost bits as necessary with AND or OR instructions.
- The Z80 double-word rotate instructions (RLD and RRD) are not available. Single-word rotate instructions must be used instead.
- The following Z80 I/O instructions are not available in the 8085: IN r,(C), OUT (C),r, INI, INIR, IND, INDR, OUTI, OTIR, OUTD, and OTDR. In the 8085, input and output ports may be loaded to and from the A register only, and the port number must appear directly as part of the two-byte opcode.
- The interrupt modes of the 8085 are determined through the SIM instruction, rather than the Z80 opcodes IM 0, IM 1, and IM 2. Return from an interrupt is accomplished with a simple RET rather than RETI or RETN.

The following general principles, then, should guide you in converting a Z80 program for Model 100 use:

1. I/O locations in the Model 100 are all different than for any other machines. (*See* table 5.4.) In particular, the TRS-80 Models I, III, and IV accomplish some I/O through memory-mapped devices accessed by LD instructions rather than IN and OUT instructions.
2. Any Z80 instruction starting with 08, 10, 18, 20, 28, 30, Z8, CB, D9, DD, ED, or FD must be replaced somehow.
3. Subroutine calls to ROM will be different in the Model 100 than in any other machine.
4. References to IX, IY, AF', BC', DE', HL', refresh register R, and interrupt page register I must be replaced somehow.
5. Relative jumps (JR) must be converted to absolute jumps (JP).

If you have a Z80 assembler operating on another machine (such as a Model I, III, and IV) you can use it as an aid in Model 100 assembly programming, at least as a convenient way to assemble mnemonics into hex code. (Strictly speaking, this is called using the assembler as a *cross-assembler*.) The following points should be considered:

1. Always print and examine a listing of the assembly process to be sure you have not used *forbidden* opcodes starting with Z8, 10, 18, 20, 28, 30, 38, CB, D9, DD, ED, and FD. Your listing should show no four-byte opcodes, as there are no four-byte opcodes in the 8085 instruction set.
2. If you are using RIM or SIM instructions, you will have to insert them manually, holding their place in the mnemonic source with NOP instructions.
3. If the assembler accepts 8080 mnemonics, you may wish to learn and assemble them instead of Z80 mnemonics, as this provides a built-in safeguard against using *forbidden* opcodes.

With a bit of ingenuity you may be able to work out a way to load the hex values from the assembly machine into the Model 100.

5

Understanding the Hardware of the Model 100

The Model 100 uses an 80C85 microprocessor. The letter "C" indicates that it is a CMOS version of the 8085 device, which means that it draws very little power. Most of the integrated circuits within the Model 100 are CMOS devices. This was done to conserve battery power. In this and subsequent chapters, 8085 is used to reference this microprocessor rather than 80C85, since the two devices are so similar.

Communication between the 8085 CPU and the rest of the world is accomplished almost exclusively through the input/output ports. The 8085 has four ways of communicating with the circuitry:

- the 65536 memory addresses which are used in the Model 100 for RAM and ROM access
- the 256 I/O ports, of which twelve are currently used in the Model 100
- the serial input and output pins, used in the Model 100 for cassette I/O
- the interrupt pins which are used for various purposes.

Memory Locations

The CPU has the ability to load to and from a large number of memory addresses, selected by turning on and off combinations of the sixteen address lines. The number of distinct addresses is two to the sixteenth power, or 65536.

In the Model 100, the bottom half of this so-called address space contains read-only memory or ROM. Depending on certain port outputs, memory accesses within this part of the address space connect with the standard ROM chip M12 or a chip in an optional ROM socket M11.

In other words, PEEKs to addresses below 32768 yield one set of values if the standard ROM is selected, and another set of values if the option ROM is selected. When the computer is turned on, it sets itself to the standard ROM. (The option ROM socket is discussed in chapter 17.)

Read-only memory, as its name suggests, cannot be written to. If you try to change its contents, by means of a POKE in BASIC or a store instruction in machine language, you will find its contents unchanged. Fortunately you cannot cause any harm to the ROM by doing this.

The top half of address space (numerically speaking) is set aside for RAM chips, shown in figure 5.1. An 8K machine (catalog no. 26-3801) has RAM soldered in place from E000 to FFFF with three sockets in the area from 8000 to DFFF. A 24K machine (catalog no. 26-3802) has RAM soldered in place from A000 to FFFF and a single socket for 8000 to 9FFF.

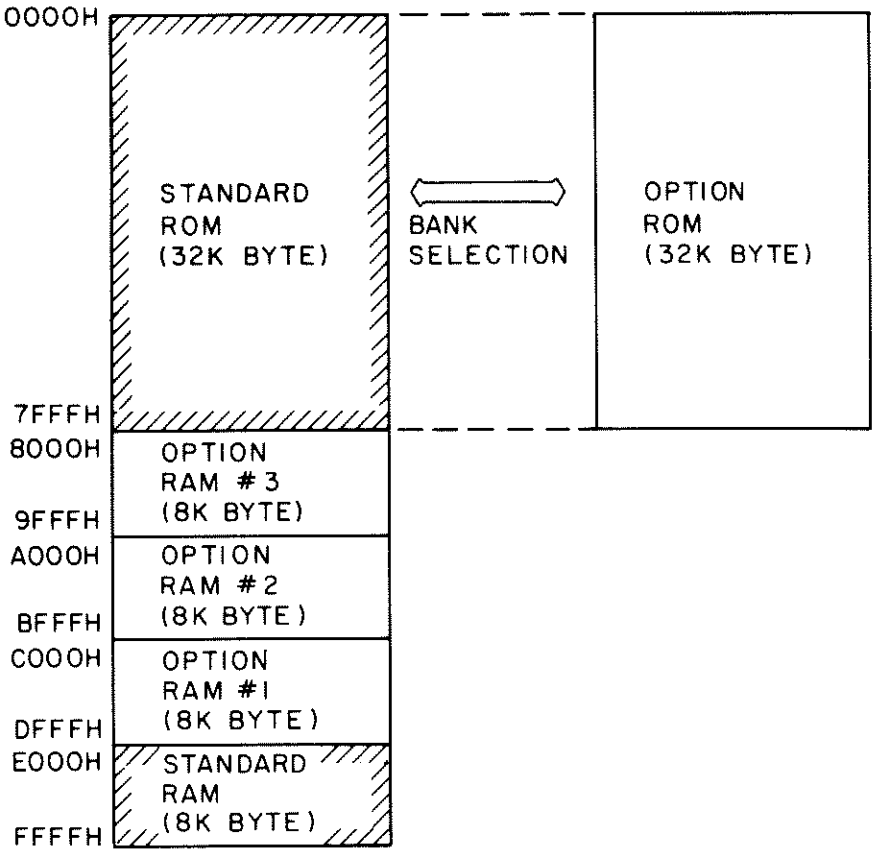

Figure 5.1. Memory map

Optional RAM modules can be installed. They can be plugged into any vacant sockets and the CPU will be able to access them, but the ROM operating system will only "discover" and use the RAM that extends in an unbroken series down from FFFF.

The serial input and serial output pins of the CPU, SID and SOD (pins 5 and 4 respectively) are used for cassette input and output. This is discussed in detail in chapter 12. The interrupt pins of the CPU (pins 6, 7, 8, 9, and 10) provide a variety of inputs to the CPU. They are discussed in detail in chapter 15. The cassette and interrupt pin assignments are shown in table 5.1.

Table 5.1. Input/Output signals originating or terminating at the CPU chip

Pin	CPU Name	Signal	Function
4	SOD	SOD	Serial output to cassette
5	SID	SID	Serial input from cassette
7	RST 7.5	TP	256-Hertz pulse
8	RST 6.5	DR	UART data received
9	RST 5.5	BCR	Bar-code reader input
10	INTR	INTR	Expansion bus pin 17

When the 8085 executes an IN or OUT instruction, several things happen. The CPU asserts the IO/M* line, indicating that it wants to talk to an I/O device rather than a memory chip. The eight-bit port address is made available on the address lines. Actually, the CPU offers the port address twice, once on the top half of the sixteen address lines, and again on the bottom half. Next, the CPU waits for an I/O device to offer or accept an eight-bit byte of data.

The eight address lines used to select I/O ports are connected to an *address decoder*, one or more integrated circuits whose task is to interpret the various possible combinations of off and on among the eight lines. Other devices are then activated accordingly. Most of the port address decoding duties in the Model 100 are performed by M16, which takes bits 7, 6, 5, and 4 of the port address as input and determines which port integrated circuit, if any, is activated. This is shown in figure 5.2.

The connection of the IO/M* signal to M16 insures that M16 responds only to I/O port addresses and not to memory (RAM or ROM) addresses. The connection of address bit 7 causes M16 to respond only when bit 7 is on. In other words it responds only to port addresses above 127 (80H to FFH).

The connection of bits 4, 5, and 6 causes one of the eight outputs to be selected (pulled low) in response to the bit pattern. The chip-select signal Y0 is activated for any port address in the range of 80 to 8F. Y1 is activated for any address in the range of 90 to 9F, and so on up to Y7, which corresponds to F0 to FF.

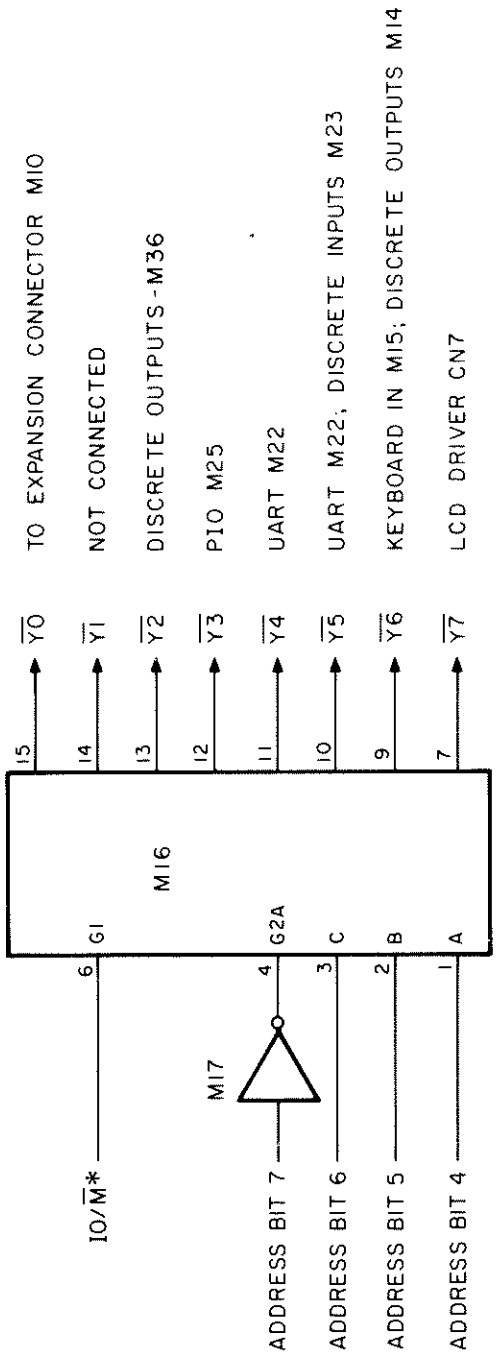

Figure 5.2. Address decoding bits 7, 6, 5, and 4

The integrated circuits to which the Y signals are attached are listed in table 5.2, which shows the input devices, and table 5.3, which shows the output devices. (The "contents" column in table 5.3 is explained later.)

Table 5.2. Input port hardware

Y	Port	Integrated Circuit
Y3	BB	M25- PIO (port C)
Y4	C8	M22- UART (receiver buffer)
Y5	D8	M23- buffer
Y6	E8	M15- buffer
Y7	FE,FF	CN7- LCD connector

Table 5.3. Output hardware and access to contents

Y	Port	Integrated Circuit	Contents
Y2	A8	M36- flipflop	FAAE
Y3	B8	M25- PIO	Not available
Y3	B9	M25- PIO (port A)	Input port B9
Y3	BA	M25- PIO (port B)	Input port BA
Y3	BC,BD	M25- PIO (divider)	Not available
Y4	C8	M22- UART transmitter	Not available
Y5	D8	M22- UART control	Not available
Y6	E8	M14- Flipflop	FF45
Y7	FE,FF	CN7- LCD connector	

What about port address lines 0, 1, 2, and 3? M16 pays no attention to them. It has no way of knowing whether the CPU has requested a port input from 80, 81, 82, or any other value up to 8F. It activates Y0 for any of these addresses. This makes for a certain arbitrariness when you are writing a program. If the task is to get a byte of data from the UART, the result is the same whether you input from C0 or CF, or any value in between. This is because the UART itself ignores address bits 0 to 3. Throughout the ROM, the value C8 is used.

A few of the devices selected by M16 do pay attention to bits 0 to 3. For example, the PIO chip, selected by Y3, looks at bits 0 to 2 and responds differently, depending on which bit is on. B8, B9, BA, BB, BC, and BD are distinct port addresses in the Model 100. Taking into account the various ways port address lines are connected in the Model 100, you can develop a map of "port space" somewhat like the address space, as shown in figure 5.1. This is depicted as a diagram in figure 5.2 and table 5.4.

Table 5.4. Port numbers

Port Number	Input Function	Output Function
00-6F	Not used	Not used
70-8F	See text	See text
90-9F	Hobby use	Hobby use
A8*	Not used	Phone/modem
B8 (&B0)	Not used	PIO divider control C3-start 43-stop
B9 (&B1)	Port contents	Parallel outputs: LCD, LPT, KB control, clock/calendar
BA (&B2)	Port contents	Output pins (see table 5.6)
BB (&B3)	Input pins (see table 5.7)	Not used
BC (&B4)	Not used	PIO divider lower byte
BD (&B5)	Not used	PIO divider upper byte and mode
B6,B7,BE,BF	Not used	Not used
C8*	UART in- coming data	UART outgoing data
D8*	input pins	UART control
E8*	Keyboard input	Output pins (see table 5.8)
FE*	LCD	LCD
FF*	LCD	LCD

* Model 100 port addresses are not fully decoded. For example, all ports A0 through AF respond identically to A8.

The Ports

No circuitry has been supplied to handle ports 00 to 6F, and nothing in the ROM suggests expansion in that area.

Ports 70 through 8F, though only partially implemented through the Y0 signal, appear to be intended for loading parallel data to and from some mass storage device plugged into the expansion bus connector M10. Perhaps these are part of the optional I/O control unit or RAM file unit referred to on page 4-12 of the service manual.

Ports 90 through 9F, which correspond to port-select signal Y1, are not connected to anything. This is described in the service manual as an optional telephone answering unit, and is discussed further in chapter 17.

Port A8 controls discrete functions. Bit 0 disconnects the telephone instrument and bit 1 enables modem carrier transmission. Usually you want to change only one of the bits. You can find the present contents of the port in RAM at FAAE, change bits using AND and OR operations, and write out to the port and to RAM.

Output port B8 programs the 81C55 PIO. The 81C55 PIO (Programmable Input/Output) chip is a forty-pin integrated circuit that does much of the I/O work of the Model 100. As it comes from the factory, it contains 256 bytes of RAM that never get used in the Model 100. It also contains three ports (A, B, and C) that are capable of being programmed as input or output ports, but the wiring of the Model 100 is such that ports A and B are always used for output (and their interrupt capability is never used), and C is always used for input. *See* table 5.5, which shows how the PIO discrete input and outputs are wired. Table 5.5 also shows other PIO connections.

Of the eight bits that can be output to port B8, six never change, as they would make the PIO do things the Model 100 wiring does not let it do. If you do inadvertently send the wrong values for these bits, no harm is done to the hardware.

Only two of the bits ever vary, bits 6 and 7. They control a so-called timer, which as used in the Model 100, would be better termed a divider. The divider-control bits 6 and 7 should be 11 (binary) to start the divider, and 10 to stop it. (This is explained in detail in chapters 7 and 9.)

Table 5.5. Signals originating or terminating at PIO

Pin	PIO Name	Signal	Function
1	PC3	BCR	Bar-code reader input
2	PC4	CTS	Clear-to-send
5	PC5	DSR	Data-Set ready
6	TO	RRC	UART receiver clock
6	TO	TRC	UART transmitter clock
8	CE	Y3	Ports 176-191 select
32	PB3	RS232C	RS232/modem select
33	PB4	PCS	Power control signal
35	PB6	DTRR	Data Terminal Ready
36	PB7	RTS	Request-to-send/off hook
37	PC0	DATAOUT	Clock/Calendar serial out
38	PC1	BUSYNOT	Line printer selected
39	PC2	BUSY	Line printer busy

Port B9 is the general-purpose parallel output, accomplished through PIO port A. It is used for the printer (*see* chapter 10), LCD (chapter 13), and keyboard (chapter 6). In addition, it is used to send serial data to the clock/calendar chip (chapter 11). Current contents of the port are obtained by reading from the port.

Port BA, like A8, controls discrete functions. Unlike A8, it is accomplished through the PIO chip (port B), so that current contents of the port are obtained by reading from the port. Bit 0 scans the keyboard modifier keys such as the shift and control keys. Bits 0 and 1 address the LCD. Bits 2 and 5 control the beeper (*see* chapter 9). Bit 3 switches from RS232 to modem mode (*see* chapters 7 and 8). Bit 4 removes power to the computer (*see* chapter 16). Bits 6 and 7 assert DTR and RTS when in RS-232 mode (*see* chapter 7). Bit 7 hangs up the phone when in modem mode (*see* chapter 8). This is shown in table 5.6.

Port BB is PIO input port C, used for sensing discrete signals. Bit 0 is clock/calendar data. Bits 1 and 2 are printer status. Bit 3 is bar-code reader input (chapter 14). Bit 4 is CTS or ANS/ORIG (chapters 7 and 8), and bit 5 is DSR or DIR/ACP (chapters 7 and 8). These input signals are shown in table 5.7.

Table 5.6. Output signals (output port BA)

Bit	Function
0	LCD control; keyboard scan e.g. SHIFT, NUM, CAPS
1	LCD control
2	Disconnect beeper from PIO divider
3	Switches from RS232 to modem
4	Power-control signal
5	Direct beeper control
6	DTR (0 yields + at RS-232 pin 20)
7	In RS-232 mode: RTS (0 yields + at RS-232 pin 4) in modem mode: phone line on-hook

Table 5.7. Input signals (input port BB)

Bit	Function
0	Clock/calendar data to CPU
1	LPT not busy (PRINTER pin 25)
2	LPT busy (PRINTER pin 21)
3	BCR input (1=ground at pin 2)
4	In RS232 mode (CTS; + at RS-232 pin 5 yields logic 0) in modem mode (1=ANS, 0=ORIG)
5	In RS232 mode (DSR; + at RS-232 pin 6 yields logic 0) in modem mode (1=ACP, 0=DIR)
6,7	Not used (always 1)

Ports BC and BD load the low and high bytes, respectively, of the divider used by the PIO to produce the baud rate (chapter 7) and beep frequency (chapter 9).

The UART sends and receives data through port C8 (chapter 7). Port D8, like port BB, provides discrete inputs to the CPU. Bit 0 is the carrier-detect signal (chapter 8). Bits 1, 2, and 3 indicate UART overrun, framing, and parity errors (chapter 7). Bit 4 is the UART transmitter buffer register empty signal (chapter 7). Bit 5 is the phone jack RP signal (chapter 17). Bit 6 is not fully implemented in hardware (*see* chapter 17), and bit 7 is the low-power signal (chapter 16).

The UART parameters (parity, word length, and so on) are programmed through output port D8 (chapter 7).

Output port E8 controls a number of discrete functions. Bit 0 selects the option ROM. Bit 1 strobes the printer. Bit 2 strobes the clock/calendar chip and bit 3 controls the cassette motor. You can find the present contents of the port in RAM at FAAE. These signals are shown in table 5.8.

Table 5.8. Output signals (output port E8; contents at FF45)

Bit	Function
0	STROM (1=select option ROM M11)
1	STROBE (1=ground at PRINTER pin 1)
2	STB (1=clock/calendar strobe)
3	REMOTE (1=CASSETTE pins 1 and 3 shorted)
4-7	Not used

Input port E8 provides a parallel input— the results of a keyboard scan. Input and output ports FE and FF are used for the liquid crystal display.

Connectors in the Model 100

The connectors in the Model 100 are sometimes referred to by number, and are listed in order in table 5.9. In addition, the LCD printed circuit board contains connectors numbered CN1, CN2, and CN3, The acoustically-coupled modem contains two connectors each numbered CN1, and two connectors numbered CN2.

Table 5.9. Model 100 connectors

Connector	Function
CN1	Keyboard connector (chapter 6)
CN2	Bar-code reader connector (chapter 14)
CN3	Cassette connector (chapter 12)
CN4	Phone connector (chapter 8)
CN5	Printer connector (chapter 10)
CN6	RS-232C connector (chapter 7)
CN7	Liquid-Crystal Display connector (chapter 13) (corresponds to CN1 on LCD board)
CN8	Low Battery LED connector (chapter 16 (corresponds to CN3 on LCD board)
CN9	DC 6V connector (chapter 16)
M10	Expansion bus connector (chapter 17)
M11	Option ROM socket (chapter 17)

The Model 100 contains three relays, listed in table 5.10.

Table 5.10. Relays in the Model 100

Relay	Function
RY1	Cassette motor control (chapter 12)
RY2	Telephone off-hook (chapter 8)
RY3	Phone instrument relay (chapter 8)

The Model 100 contains three quartz crystals used for various time-sensitive functions; these are listed in table 5.11.

Table 5.11. Model 100 crystals

Crystal	Frequency	Used by
X1	32.768 KHz	Clock/Calendar (chapter 11)
X2	4.9152 MHz	CPU
		PIO (chapters 7, 9)
		UART (chapter 7)
		Beeper (chapter 9)
X3	1.000 MHz	Modem (chapter 8)

In addition to the switches contained in the keyboard, there are five switches controlling major functions of the Model 100. They are listed in table 5.12.

Table 5.12. Model 100 switches

Switch	Function
SW-1	Answer/originate switch (chapter 8)
SW-2	Direct/acoustic switch (chapter 8)
SW-3	Memory power switch (chapter 16)
SW-4	Reset pushbutton (chapter 16)
SW-5	On/off switch (chapter 16)

6

The Keyboard

The Model 100 keyboard has fifty-six conventional typewriter style keys and sixteen small function keys. They are identical electrically. The computer's response to the pressing of a particular key is determined by the program being run. Since most programs use one of two ROM routines for reading the keyboard, it makes sense to think of the keys in terms of the values returned by these routines.

Hardware Theory of Operation

The Model 100's keyboard consists of key switches soldered to a printed circuit board. The keys are numbered on the board and the correspondence between key numbers and the legend printed on the key top is shown in table 6.1. To reorder a key top, you will need the reference number given in the table.

Table 6.1. Printed circuit board key designations.

PCB Key	Serv. Ref. no.	Man. Description
1	P-100	f1
2	P-100	f2
3	P-100	f3
4	P-100	f4
5	P-100	f5
6	P-100	f6
7	P-100	f7
8	P-100	f8
9	P-100	PASTE
10	P-100	LABEL
11	P-100	PRINT
12	P-100	BREAK
13	P-100	Leftarrow
14	P-100	Rightarrow
15	P-100	Uparrow
16	P-100	Downarrow
17	P-200	ESC
18	P-101	1
19	P-102	2
20	P-103	3
21	P-104	4
22	P-105	5
23	P-106	6
24	P-107	7
25	P-108	8
26	P-109	9
27	P-110	0
28	P-201	-
29	P-202	=
30	P-203	BKSP
31	P-214	TAB
32	P-127	q
33	P-133	w
34	P-115	e
35	P-128	r
36	P-130	t
37	P-135	y
38	P-131	u
39	P-119	i
40	P-125	o
41	P-126	p

continued on following page

PCB Key	Serv. Ref. no.	Man. Description
42	P-204	[
43	P-217	Enter
44	P-215	CTRL
45	P-111	a
46	P-129	s
47	P-114	d
48	P-116	f
49	P-117	g
50	P-118	h
51	P-120	j
52	P-121	k
53	P-122	l
54	P-205	;
55	P-206	'
56	P-207	CAPS
57	P-216	SHIFT
58	P-136	z
59	P-134	x
60	P-113	c
61	P-132	v
62	P-112	b
63	P-124	n
64	P-123	m
65	P-208	,
66	P-209	.
67	P-210	/
68	P-216	SHIFT
69	P-211	GRPH
70	P-218	Space
71	P-212	CODE
72	P-213	NUM

The seventy-two Model 100 keys reside in port space unlike the Model I or III keys which reside in memory address space. This means that in the Models I and III the CPU determines which keys have been pressed by loading in data from certain of the 65536 possible memory addresses. In the Model 100, however, the CPU determines key closures by loading in data from certain of the 256 possible I/O ports. As in the Model I and III, keyboard scanning requires the constant

attention of the CPU. This means that if a period of time passes during which the CPU has not scanned the keyboard, any key pressed during that time is ignored.

Keyboard scanning can occur due to direct action by the program being executed or because certain interrupts are enabled.

Keyboard Scanning

The electrical layout of the seventy-two keys is an eight-by-nine array, as shown in figure 6.1. Sixty-four keys lie in the main array and are scanned together. They appear in table 6.2. The eight remaining keys, most of which modify other keys, appear in table 6.3.

To determine if a particular key has been pressed, the CPU sends a "0" to a selected bit of output port B9 (or B1; decimal 177 or 185) and "1's" to the other bits. It also makes sure bit 0 of output port BA is on. This selects a column in table 6.2. Then, the byte at input port E8 (any port in the range E0 through EF will do, decimal 224 through 239) is examined. If one of the bits is "0", the key corresponding to that row on table 6.2 has been pressed.

A slightly different procedure applies to the keys in table 6.3. To scan these keys, the CPU sends a "0" to bit 0 of output port BA and all "1's to output port B9, while examining input port E8. If any of the bits are "0", the key corresponding to that row on table 6.3 has been pressed.

Table 6.2. "Main array" key locations in port space. Input takes place through input port E8. If a bit is off, that key has been pressed.

INP E8 bit	Output port B9 bit							
	0	1	2	3	4	5	6	7
7	L	K	I	/	8	Down	ENTER	F8
6	M	J	U	.	7	Up	PRINT	F7
5	N	H	Y	,	6	Right	LABEL	F6
4	B	G	T	'	5	Left	PASTE	F5
3	V	F	R	;	4	=	ESC	F4
2	C	D	E	[	3	-	TAB	F3
1	X	S	W	P	2	0	DEL	F2
0	Z	A	Q	O	1	9	Space	F1

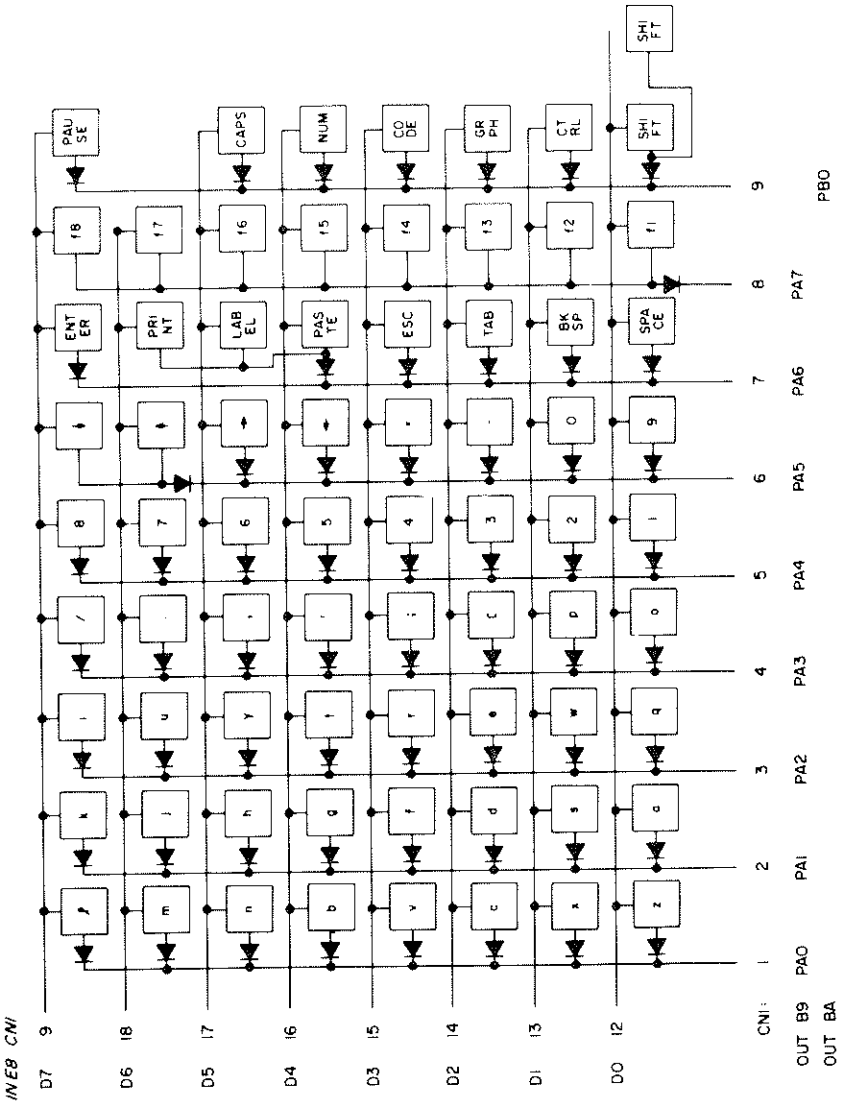

Figure 6.1. Keyboard array

Table 6.3. Modifier key locations in port space. Input condition: assumes bit 0 is off at output port BA. Input takes place through input port E8. If a bit is off, that key has been pressed.

Input port BA Bit	Key
7	BREAK
6	Always one (no key at this location)
5	CAPS
4	NUM
3	CODE
2	GRPH
1	CTRL
0	SHIFT (either or both keys)

To see how rows and columns are scanned, enter and run the program shown in figure 6.2.

```
100 FOR I=0 TO 7: OUT 177,255 XOR (2 ^ I):
    A=INP(224): IF A=255 THEN 200
150 PRINT "Table 6.2";I, 255 XOR A
200 NEXT I
201 OUT 178,0 : A=INP(224): IF A=255 THEN 300
202 PRINT "Table 6.3,"255 XOR A
300 GOTO 100
```

Figure 6.2. BASIC program demonstrating key scanning.

When you run this program and press a key, the column and row from table 6.2 and table 6.3 are printed. The rows are designated by the numerical value of the bit. For example, 128 means bit 7.

Sometimes all you want to know is whether a key has been pressed. This can be accomplished by sending a "0" to output port B9 and a "0" to bit 0 of output port BA. If input port E8 has the value FF (255), a key has not been pressed. You can see this in figure 6.1. Sending "0's" to the output port sets all nine columns low. If a key had been pressed, then one of the eight data lines at the input port would be low, or logic "0". The numerical value of the input port would be something other than all "1's" (FF, or 255 decimal).

In most circumstances the CAPS, NUM, CODE, GRPH, CTRL, and SHIFT keys are intended to generate a character only in conjunction with a key in the main array. It is often adequate to send all "0's" to output port B9, leaving bit 0 of output port BA at its present "1" state.

Multiple-Use Ports

Output ports B9 and BA serve many functions besides keyboard scanning. All of the bits of output port B9, for example, are used in keyboard scanning and for line printer output and LCD control. One of the bits is used for serial output to the clock/calendar chip. Bit 0 of output port BA, which is used for keyboard scanning, is also used for LCD control. The other bits of output port BA serve other functions.

Because of the multiple uses of ports, two precautions are in order if you are to do your own keyboard scanning. Avoid interfering with other functions, when using output port BA. Do not disturb bits 1 to 7. This is done by reading in the contents of the output port through input port BA, changing bit 0 through AND and OR statements, and sending the new value out to port BA.

The other precaution is to avoid letting other functions interfere with keyboard scanning. Here is a keyboard input routine that checks if the CTRL-BREAK combination has been pressed:

```
7D44  3E  EC  MVI A,EC   ;LD A,EC
7D46  D3  BA  OUT BA     ;OUT (BA),A
7D48  3E  FF  MVI A,FF   ;LD A,FF
7D4A  D3  B9  OUT B9     ;OUT (B9),A
7D4C  DB  E8  IN E8      ;IN A,(E8)
7D4E  E6  82  ANI 82     ;AND 82
```

This is the routine used upon power-up or reset to determine whether CTRL-BREAK was pressed at the moment the power was turned on or at the instant the RESET button was pressed. If so, all RAM files are wiped out and the Model 100 reinitializes all of memory.

The value EC at 7D44 is chosen carefully to avoid turning the power off unintentionally. It activates the modifier column of the

keyboard. The value FF at 7D48 keeps all the keys in the main array out of the scan. The input value received at 7D46 is ANDed with 82 to select bits 7 and 1 of table 6.2. The accumulator is zero only if both the CTRL and BREAK keys have been pressed.

Suppose that an interrupt were to occur between the time of execution of lines 7D46 and 7D4C. The interrupt routine might easily change the contents of output port B9 or BA, allowing keys other than CTRL and BREAK to satisfy the AND 82 test with disastrous results. It is for this reason that interrupts were disabled at line 7D33.

For those who use the published ROM calls for keyboard input, the routines contain the necessary protections to safeguard against a problem arising from the shared use of ports B9 and BA. If you do your own keyboard scanning, however, you should disable or mask interrupts for crucial inputs as discussed in chapter 15.

Keyboard ROM Calls

Radio Shack has published two sets of ROM calls— calls to collect characters from the keyboard and calls to set up the function keys.

Pressing keys provides characters to the Model 100 only if the CPU is paying attention. For keyboard input in the Model 100, it is not necessary for the main program to scan the keyboard repeatedly or even to scan it at all. This is because the CPU is sent a TP (timing pulse) signal from the clock/calendar every four milliseconds. The routine executed by the CPU upon arrival of the TP interrupt includes a keyboard scan. As you will see in chapter 15, the TP interrupt is serviced by the CPU only if interrupts are enabled and if the TP interrupt is unmasked.

When the interrupt routine finds that a key, or more accurately a key-combination has been pressed, the corresponding ASCII code is calculated. There are 128 standardized ASCII values, ranging from zero to 127. The ROM routines return the ASCII value in the A register, which can contain 256 possible values. The Model 100 defines key-combination values for all 256 values. There are 269 key-combinations defined for the Model 100. The additional thirteen possibilities are also returned in the A register but with the carry flag on. These "pseudo-ASCII" values are listed in table 6.4.

Table 6.4. Pseudo-ASCII values for certain keys.
(Radio Shack publication 700-2245 gives these values incorrectly. This table should be used instead.)

value in accumulator	key pressed
0	f1
1	f2
2	f3
3	f4
4	f5
5	f6
6	f7
7	f8
8	LABEL
9	PRINT
A	SHIFT-PRINT
B	PASTE

When keyboard scanning occurs as a result of the TP interrupt, the character returned, if any, is stored in a buffer. This is sometimes called a queue. Keyboard scanning is not the only way characters are loaded into the buffer. Pressing PASTE, or any of the eight function keys, also causes characters to be loaded into the buffer.

If a key is pressed long enough to activate the repeat action, about one second, the ASCII value for that key will show up in the buffer many times.

When characters are loaded into the buffer they do not, in and of themselves, end up on the LCD screen. The main program must arrange for characters to be displayed on the screen if that is desired. Routines to accomplish this are described in chapter 13.

KEYBOARD INPUT ROM ROUTINES

Perhaps the simplest of the keyboard input routines is one which bears a striking resemblance to the BASIC function INKEY$. This routine, named KYREAD and called at 7242, determines whether, at the time of the call, any key combination is being pressed.

If no meaningful key combination has been pressed, the routine returns with the Z (zero) flag on. If the Z flag is off, a character has been received, and its value is found in the accumulator. If the C (carry) flag

is on, the value in the accumulator is not an ASCII value but is instead a pseudo-ASCII value and should be interpreted according to table 6.4. This routine is somewhat like the Model I/III routine KBCHAR at 002B.

You may wonder why the routine is defined as responding to "meaningful" key-combinations. Just because a key was pressed at the time of the scan does not mean the result will be located in the accumulator. For example, if GRPH-G is pressed, the routine still comes back with the Z flag on. This is because GRPH-G is not meaningful. Table 6.5 shows this. If you want to be able to detect a GRPH-G, you will have to scan the keyboard yourself, as described earlier in the chapter.

Sometimes you may want to know if a key has been pressed since the CPU last received a character from the keyboard. This can be accomplished by CHSNS, called at 13DB. Upon return from the routine, if the Z flag is set, no keys have been pressed.

Assembly language programmers often wish to assign special significance to two ASCII values, 03 hex and 13 hex, because they have special meanings in BASIC. The value 03 hex, associated with CTRL-C and SHIFT-BREAK, is often used to terminate the program in progress. The value 13 hex, associated with CTRL-S and PAUSE, is often used to temporarily suspend the function is progress, such as scrolling of the screen. A routine is available to determine whether 03 or 13 have been typed. The routine BRKCHK, called at 7283, returns with the C flag set if either character has been received and reset if neither has been received.

The routine KEYX, called at 7270, combines the functions of BRKCHK and CHSNS. If the Z flag is set, no character has been received. If the Z flag is reset, at least one character has been received. If the C flag is set, a CTRL-S or CTRL-C has been received.

If you learn that a character resides in the keyboard buffer, you must obtain the character. This is done by calling CHGET at 12CB. As in the case of KYREAD, the condition of the carry flag indicates whether the value returned in the accumulator is to be understood as ASCII or pseudo-ASCII.

CHGET has one other use. If no character is in the buffer, CHGET causes the computer to wait until a character is typed before returning control to the calling program. In this respect, it is somewhat like the Model I/II KBWAIT routine at 0049. Put another way, if you do not want to wait for a character, but simply want to process any pending buffer entries, you should check the buffer first (using CHSNS or KEYX) and not call CHGET unless something is there.

A related but unpublished routine is located at 5D64. This routine performs the CHGET function and converts the results to uppercase.

Calling CHGET to retrieve a character from the buffer does not cause the character to be displayed on the screen. The interrupt routine that collects the character and places it in the buffer does not put it on the screen. If you want it to appear on the screen, you have to put it there yourself. This is a complicated task because the entry can be a delete or backspace or another character that requires special attention.

Often you may want to receive a line of characters from the keyboard, terminated by an ENTER (carriage return, ASCII value 0D hex). This could be accomplished with the routines described above but doing so would involve calling routines repeatedly and checking each character to see if it is an ENTER.

The routine INLIN, called at 4644, performs this task. The line that was typed appears in a RAM buffer starting at F685. This routine is somewhat like the Model I/III routine KBLINE at 0040.

As the line is typed, the user is able to correct characters with the left arrow, backspace, or CTRL-H. It is also possible to erase the whole line with CTRL-U and start over. All characters are displayed on the screen. The INLIN routine is used in the BASIC LINE INPUT command (at 0C5F) and is used to collect the user's input in response to the BASIC Ok prompt at 051D, the TELCOM prompt at 516A, the Term Width: prompt at 55D4, and the ADRS and SCHEDL command lines at 5BD2.

Label Lines and Functioning Keys

When a function key (f1 through f8) is pressed, a string of characters associated with that key is loaded to the keyboard buffer. It is possible for the user to change the strings assigned to these function

keys by calling STFNK at 5A7C. Before calling the routine a so-called "f-string" sequence must be set up, and the HL register must point to (contain the address of) the sequence. The sequence must be set up in a precise way; it is composed of eight f-strings, where each f-string is made up of sixteen or fewer characters. If it is made of fewer than sixteen characters, the last character must have bit 7 turned on.

Only the lowest seven bits of the f-string will be displayed on the screen and (when the function key is pushed) loaded to the keyboard buffer. As a result, the CODE and GRAPH characters may not appear in an f-string. (The only exceptions are GRPH-SHIFT-hyphen and GRPH-SHIFT-[, with ASCII values 124 and 126 decimal, respectively.)

However, all ASCII values up to 127 decimal may appear in an f-string including such exotic characters as CTRL-G, which appears as the "beep" when sent to the screen.

Let's look at a typical f-string sequence, the TELCOM label line, shown in table 6.5.

Table 6.5. TELCOM label line setting

```
5155 21 51 LXI  HL,51A4;LD HL,51A4
5158 CD7C5A CALL   5A7C
.
.
.
51A4 46 69 6E  64      ;"Find"
51A8 A0                ;space with bit 7 on
51A9 43 61 6C  6C      ;"Call"
51AD A0
51AE 53 74 61  74      ;"Stat"
51B2 A0
51B3 54 65 72  6D      ;"Term"
51B7 8D                ;ENTER with bit 7 on
51B8 80 80 80          ;f5, f6, f7 empty
51BB 4D 65 6E  75      ;"Menu"
51BF 8D
```

This table illustrates a few points about function key labels. If after pushing the function key, the function is to be processed immediately, a command line terminator like ENTER is required. This is the case with function keys f4 and f8.

If, on the other hand, the user can enter characters before pressing ENTER, one may turn on bit 7 of the last label character or append a space with bit 7 on. This the case with function keys f1, f2, and f3.

Finally, if the value 80 is specified for keys that are to be blank, such as keys f5, f6, and f7, the ROM routine CLRFNK at 5F79 loads 80 to all eight keys, resulting in a clearing of all the function keys and labels.

Once the labels are set, other routines cause them to be displayed on the screen — DSPFNK at 42A8, or erased from the screen ERAFNK at 428A.

The actions of STFNK and DSPFNK are combined in STDSPF, at 42A5; this routine, like STFNK, requires that HL be set to point to the f-string sequence.

The RAM location at F63D is used as a label line enable flag. If it is nonzero, the label line is considered to be enabled. (The flag is set at 443B.) The routine FNKSB at 5A9E will cause the function key labels to be displayed (by calling DSPFNK) only if F63D is nonzero.

Function key f-string sequences in ROM include BASIC at 5B46; ADRS and SCHEDL at 5D0A, and 5D1E; TELCOM at 51A4, 5443 and 5D2B; TEXT at 5E15; and a sequence to clear all labels at 5B3E.

ROM KEYBOARD SCANNING

Several tables are stored in ROM which are used to convert the key closures to ASCII values. Consider first the keys in the first six columns of table 6.1 (except the four arrow keys). These forty-four keys, alone and when modified by the SHIFT, GRPH, and CODE keys, account for the majority of the possible ASCII values returned by INKEY$ or the ROM calls. The decoding is done with the aid of a table located in ROM at 7BF1 to 7CF8.

For any of these keys, the ASCII value is located at the memory location pointed to by the sum of the following decimal numbers:

- 31729 (start of table)
- bit number 0-7 found to be turned off at input port 232
- eight times the bit number 0-7 turned off at output port 185

- if SHIFT key pressed, add 44
- if GRPH or CODE key pressed, add 88 or 132 respectively.

This ROM table was used to generate table 6.5, which shows the effect of SHFT, GRPH, and CODE keys on the above-mentioned forty-four keys.

The ROM table contains a number of zero entries, e.g. CODE—SHIFT-z. This does not mean that the INKEY$ value for that combinations of key-presses is CHR$(0). Instead INKEY$ returns a null string (""). One way to get a CHR$(0) is with the input CTRL-SHIFT-2.

Portions of this table are used in several ROM routines, at 718E, 7195, 71A0, 71B2, and 720D.

Decoding of Function Keys

Decoding of keys in the last two columns of table 6.2 is accomplished in similar fashion with ROM tables at 7D0B (for SHIFTed keys) and 7D10 (for unSHIFTed keys). To obtain the values returned by ROM call KYREAD or CHGET you must subtract 64 from the value in the ROM table.

Decoding the Directional Arrows

The directional arrows take on differing values depending on whether the SHIFT or CTRL keys are pressed. Table 6.7 shows the values and locations of lookup tables.

Table 6.6. ASCII values for SHFT, GRPH, CODE keys

Key	Unshifted	Shift	GRPH	GRPH-Shift	CODE	CODE-Shift
'	39	34	140	0	160	164
,	44	60	153	248	188	221
-	45	95	92	124	197	167
.	46	62	151	247	207	0
/	47	63	138	0	174	0
0	48	41	125	0	175	166
1	49	33	136	225	192	208
2	50	64	156	226	0	0
3	51	35	157	227	193	209
4	52	36	158	228	0	0
5	53	37	159	229	0	0
6	54	94	180	230	0	0
7	55	38	176	0	196	212
8	56	42	163	0	194	210
9	57	40	123	0	195	211
;	59	58	146	245	173	0
=	61	43	141	0	190	168
[	91	93	96	126	181	0
a	97	65	133	235	182	177
b	98	66	149	0	0	0
c	99	67	132	255	162	171
d	100	68	0	237	187	215
e	101	69	143	233	198	214
f	102	70	130	238	0	191
g	103	71	0	253	0	0
h	104	72	134	251	0	0
i	105	73	142	243	199	213
j	106	74	0	244	203	219
k	107	75	155	250	201	217
l	108	76	154	249	202	218
m	109	77	129	246	0	165
n	110	78	150	0	205	0
o	111	79	152	242	183	178
p	112	80	128	241	172	0
q	113	81	147	231	200	216
r	114	82	137	234	0	170
s	115	83	139	236	169	185
t	116	84	135	252	0	186
u	117	85	145	240	184	179
v	118	86	0	0	189	222
w	119	87	148	232	0	0
x	120	88	131	239	161	223
y	121	89	144	254	204	220
z	122	90	0	224	206	0

Table 6.7. ASCII values for Arrow keys

	Left	Right	Up	Down	ROM Table Location
Unshifted	29	28	30	31	7D1B
SHIFT	1	6	20	2	7D07
CTRL	17	18	23	26	7D2F*

Nums Decoding

Certain keys take on alternate meanings when the NUMS key is down. The routine to convert such keys is located at 7233-7241, and it uses a table at 7CF9-7D06. Assuming the key pressed has already been decoded to ASCII, the lower-case value is compared to the value at 7CF9, 7CFB, and so. If there is a match, the value immediately following (7CFA, 7CFC, etc.) is substituted.

Arriving At A Particular INKEY$ Value

The question often arises what combination of keystrokes will produce a certain INKEY$ value. The ASCII table given in the Radio Shack manuals is incomplete and incorrect. The correct and complete repertoire of keystrokes is given in table 6.8. Note, for example, that in some cases (e.g. 1, 2, 6, 8, 9, 13, 17, 18, 20, 23, and 26) more than one combination of keystrokes will produce a given INKEY$ value.

* Subtract 64 from table values.

Table 6.8. INKEY$ keystroke combinations

Decimal	Hex	LCD	INKEY$	
1	01		CTRL-a	SHIFT-Leftarrow
2	02		CTRL-b	SHIFT-Downarrow
3	03		CTRL-c	
4	04		CTRL-d	
5	05		CTRL-e	
6	06		CTRL-f	SHIFT-Rightarrow
7	07		CTRL-g	
8	08		CTRL-h	BKSP
9	09		CTRL-i	TAB
10	0A		CTRL-j	
11	0B		CTRL-k	
12	0C		CTRL-l	
13	0D		CTRL-m	ENTER
14	0E		CTRL-n	
15	0F		CTRL-o	
16	10		CTRL-p	
17	11		CTRL-q	CTRL-Leftarrow
18	12		CTRL-r	CTRL-Rightarrow
19	13		CTRL-s	
20	14		CTRL-t	SHIFT-Uparrow
21	15		CTRL-u	
22	16		CTRL-v	
23	17		CTRL-w	CTRL-Uparrow
24	18		CTRL-x	
25	19		CTRL-y	
26	1A		CTRL-z	CTRL-Downarrow
27	1B		ESC	
28	1C		Rightarrow	
29	1D		Leftarrow	
30	1E		Uparrow	
31	1F		Downarrow	
32	20		Space	
33	21	!	SHIFT-1	
34	22	"	SHIFT-'	
35	23	#	SHIFT-3	
36	24	$	SHIFT-4	
37	25	%	SHIFT-5	

Decimal	Hex	LCD	INKEY$	
38	26	&	SHIFT-7	
39	27	'	'	
40	28	(	SHIFT-9	
41	29	)	SHIFT-0	
42	2A	*	SHIFT-8	
43	2B	+	SHIFT-=	
44	2C	,	,	
45	2D	-	-	
46	2E	.	.	
47	2F	/	/	
48	30	0	0	NUM-m
49	31	1	1	NUM-j
50	32	2	2	NUM-k
51	33	3	3	NUM-l
52	34	4	4	NUM-u
53	35	5	5	NUM-i
54	36	6	6	NUM-o
55	37	7	7	
56	38	8	8	
57	39	9	9	
58	3A	:	SHIFT-;	
59	3B	;	;	
60	3C	<	SHIFT-,	
61	3D	=	=	
62	3E	>	SHIFT-.	
63	3F	?	SHIFT-/	
64	40	@	SHIFT-2	
65	41	A	SHIFT-a	
66	42	B	SHIFT-b	
67	43	C	SHIFT-c	
68	44	D	SHIFT-d	
69	45	E	SHIFT-e	
70	46	F	SHIFT-f	
71	47	G	SHIFT-g	
72	48	H	SHIFT-h	
73	49	I	SHIFT-i	
74	4A	J	SHIFT-j	
75	4B	K	SHIFT-k	

Decimal	Hex	LCD	INKEY$
76	4C	L	SHIFT-l
77	4D	M	SHIFT-m
78	4E	N	SHIFT-n
79	4F	O	SHIFT-o
80	50	P	SHIFT-p
81	51	Q	SHIFT-q
82	52	R	SHIFT-r
83	53	S	SHIFT-s
84	54	T	SHIFT-t
85	55	U	SHIFT-u
86	56	V	SHIFT-v
87	57	W	SHIFT-w
88	58	X	SHIFT-x
89	59	Y	SHIFT-y
90	5A	Z	SHIFT-z
91	5B	[	[
92	5C	\	GRPH--
93	5D	]	SHIFT-[
94	5E	^	SHIFT-6
95	5F	_	SHIFT--
96	60	`	GRPH-[
97	61	a	a
98	62	b	b
99	63	c	c
100	64	d	d
101	65	e	e
102	66	f	f
103	67	g	g
104	68	h	h
105	69	i	i
106	6A	j	j
107	6B	k	k
108	6C	l	l
109	6D	m	m
110	6E	n	n
111	6F	o	o
112	70	p	p
113	71	q	q

Decimal	Hex	LCD	INKEY$
114	72	r	r
115	73	s	s
116	74	t	t
117	75	u	u
118	76	v	v
119	77	w	w
120	78	x	x
121	79	y	y
122	7A	z	z
123	7B	{	GRPH-9
124	7C	|	GRPH-SHIFT--
125	7D	}	GRPH-Ø
126	7E	~	GRPH-SHIFT-[
127	7F		SHIFT-BKSP
128	80		GRPH-p
129	81		GRPH-m
130	82		GRPH-f
131	83		GRPH-x
132	84		GRPH-c
133	85		GRPH-a
134	86		GRPH-h
135	87		GRPH-t
136	88		GRPH-1
137	89		GRPH-r
138	8A		GRPH-/
139	8B	Σ	GRPH-s
140	8C		GRPH-'
141	8D		GRPH-=
142	8E		GRPH-i
143	8F		GRPH-e
144	90		GRPH-y
145	91		GRPH-u
146	92		GRPH-;
147	93		GRPH-q
148	94		GRPH-w
149	95		GRPH-b
150	96		GRPH-n
151	97		GRPH-.
152	98		GRPH-o

Decimal	Hex	LCD	INKEY$
153	99		GRPH-,
154	9A		GRPH-l
155	9B		GRPH-k
156	9C		GRPH-2
157	9D		GRPH-3
158	9E		GRPH-4
159	9F		GRPH-5
160	A0		CODE-'
161	A1		CODE-x
162	A2		CODE-c
163	A3	£	GRPH-8
164	A4		CODE-SHIFT-'
165	A5		CODE-SHIFT-m
166	A6		CODE-SHIFT-Ø
167	A7		CODE-SHIFT--
168	A8		CODE-SHIFT-=
169	A9		CODE-s
170	AA		CODE-SHIFT-r
171	AB		CODE-SHIFT-c
172	AC		CODE-p
173	AD		CODE-;
174	AE		CODE-/
175	AF	¶	CODE-Ø
176	B0		GRPH-7
177	B1		CODE-SHIFT-a
178	B2		CODE-SHIFT-o
179	B3		CODE-SHIFT-u
180	B4		GRPH-6
181	B5		CODE-[
182	B6		CODE-a
183	B7		CODE-o
184	B8		CODE-u
185	B9		CODE-SHIFT-s
186	BA		CODE-SHIFT-t
187	BB		CODE-d
188	BC		CODE-,
189	BD		CODE-v
190	BE		CODE-=
191	BF		CODE-SHIFT-f

Decimal	Hex	LCD	INKEY$
192	C0	â	CODE-1
193	C1	ê	CODE-3
194	C2	î	CODE-8
195	C3	ô	CODE-9
196	C4	û	CODE-7
197	C5	^	CODE--
198	C6	ë	CODE-e
199	C7	ï	CODE-i
200	C8	á	CODE-q
201	C9	í	CODE-k
202	CA	ó	CODE-l
203	CB	ú	CODE-j
204	CC	ý	CODE-y
205	CD	ñ	CODE-n
206	CE	ã	CODE-z
207	CF	õ	CODE-.
208	D0	Â	CODE-SHIFT-1
209	D1	Ê	CODE-SHIFT-3
210	D2	Î	CODE-SHIFT-8
211	D3	Ô	CODE-SHIFT-9
212	D4	Û	CODE-SHIFT-7
213	D5	Ï	CODE-SHIFT-i
214	D6	Ë	CODE-SHIFT-e
215	D7	É	CODE-SHIFT-d
216	D8	Á	CODE-SHIFT-q
217	D9	Í	CODE-SHIFT-k
218	DA	Ó	CODE-SHIFT-l
219	DB	Ú	CODE-SHIFT-j
220	DC	Ý	CODE-SHIFT-y
221	DD	Ù	CODE-SHIFT-,
222	DE	È	CODE-SHIFT-v
223	DF	À	CODE-SHIFT-x
224	E0		GRPH-SHIFT-z
225	E1	▘	GRPH-SHIFT-1
226	E2	▝	GRPH-SHIFT-2
227	E3	▀	GRPH-SHIFT-3
228	E4	▖	GRPH-SHIFT-4
229	E5	▚	GRPH-SHIFT-5

Decimal	Hex	LCD	INKEY$
230	E6	▟	GRPH-SHIFT-6
231	E7	▔	GRPH-SHIFT-q
232	E8	▄	GRPH-SHIFT-w
233	E9	▌	GRPH-SHIFT-e
234	EA	▐	GRPH-SHIFT-r
235	EB	▛	GRPH-SHIFT-a
236	EC	▜	GRPH-SHIFT-s
237	ED	▙	GRPH-SHIFT-d
238	EE	▟	GRPH-SHIFT-f
239	EF	▀	GRPH-SHIFT-x
240	F0	┌	GRPH-SHIFT-u
241	F1	─	GRPH-SHIFT-p
242	F2	┐	GRPH-SHIFT-o
243	F3	┬	GRPH-SHIFT-i
244	F4	├	GRPH-SHIFT-j
245	F5	│	GRPH-SHIFT-;
246	F6	└	GRPH-SHIFT-m
247	F7	┘	GRPH-SHIFT-.
248	F8	┴	GRPH-SHIFT-,
249	F9	┤	GRPH-SHIFT-l
250	FA	┼	GRPH-SHIFT-k
251	FB	◤	GRPH-SHIFT-h
252	FC	◢	GRPH-SHIFT-t
253	FD	◥	GRPH-SHIFT-g
254	FE	◣	GRPH-SHIFT-y
255	FF	▒	GRPH-SHIFT-c

7

UART Operation And The RS-232 Interface

This chapter examines the UART, a specialized integrated circuit used for serial communications. The RS-232 port is also discussed in detail. The UART is used also for modem communications, a topic covered in chapter 8.

Parallel and Serial Data

Most communication within the Model 100 and with outside devices is accomplished using parallel data busses. *Parallel bus* refers to a group of data paths (usually wires), which at a given instant convey several bits of data (usually eight), each representing a one or a zero.

A serial transmission line, on the other hand, involves a single data path, which at a given instant conveys a single bit of data.

All other factors being equal, a parallel bus conveys data faster than a serial bus and requires less hardware per bit.

Nonetheless, serial transmission circuitry is an essential part of any computer, for two major reasons. First, any data to be transmitted by audio signals, e.g. by telephone or cassette, can only be sent serially since the medium permits only a limited number of states to represent data values. With 300-baud modem communication, for example, one audio tone represents a "1" and another tone represents a "0".

Secondly, a serial interface was required in the Model 100 to make it compatible with the many computers and peripheral devices that have serial interfaces intended to meet the RS-232C standard.

In the Model 100, a universal asynchronous receiver/transmitter, or UART, is used to convert the Model 100's parallel data to serial data. A multiplexer is used to connect the UART either to the RS-232C port or to the telephone modem circuitry.

CONVERTING FROM PARALLEL DATA TO SERIAL DATA

The process of parallel-to-serial conversion is shown in Figure 7.1. Figure 7.1a depicts an eight-bit binary word being sent down a serial transmission line. (The term *word* is used in this chapter as synonymous with the term *eight-bit byte*.) The fourth bit is just leaving the TR (transmitter register) at the time depicted here. (The fact that a single 0 or 1 state is in transition here is the defining characteristic of a so-called serial bus.)

In figure 7.1b, we see that another word has been loaded into the TR from the TBR (transmitter buffer register). Not shown in this figure is the means by which the transmitter circuitry informs the CPU that the TBR is now ready to accept another word.

In figure 7.1c we see, in transition, the loading of yet another word into the TBR. At this instant eight possible 0's or 1's are on the verge of being communicated to the TBR. (This is the defining characteristic of a so-called *parallel* bus.)

What Is A UART?

UART is an acronym for *universal asynchronous receiver/transmitter*. The parallel-to-serial process just described is termed the *transmitter* function. The reception of serial data from a distant device and conversion to parallel data is termed the *receiver* function. The

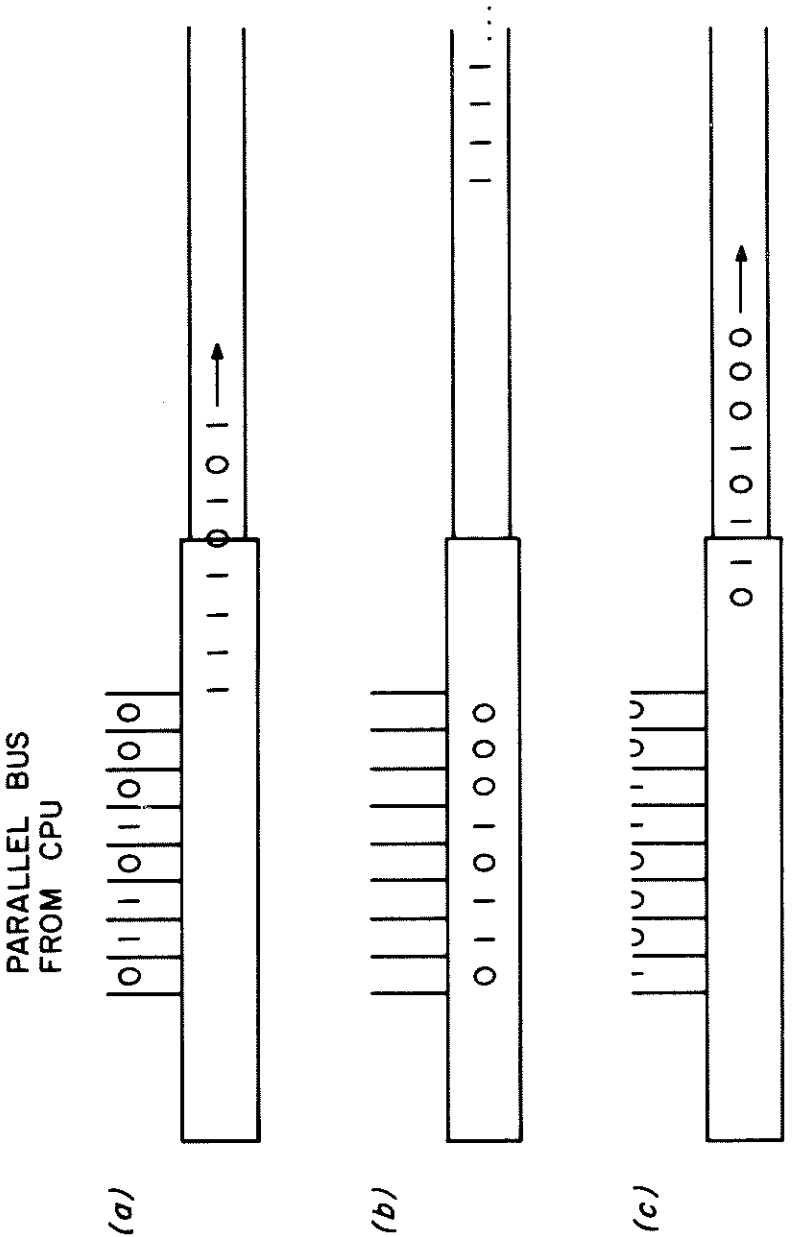

Figure 7.1. Parallel to serial conversion

adjective *asynchronous* means that the receiver can handle incoming words at irregular intervals. There is no requirement that subsequent bursts of 0's and 1's be separated by an unchanging interval, nor that any warning (other than a so-called *start bit*) be given that a word is forthcoming. It is termed *universal* because it is capable of being programmed for any of a variety of word lengths, data rates, and so on.

As with virtually all Model 100 integrated circuits, information enters and leaves the UART on conductors which carry 5 volts to represent a 1 and 0 volts to represent a 0. Thus the eight 1's and 0's depicted schematically in figure 7.1c are in reality eight voltage levels on eight wires. (The wires are pins 26-33 of the integrated circuit.) Similarly, the 1's and 0's shown leaving to the transmitter to the right represent periods of time during which a wire (pin 25) carried either a 5-volt signal or a 0-volt signal.

TIMING

Suppose a log was made of the voltage at the output of the transmitter as time passes. What would be observed? The result for transmission of a capital "A" at 300 baud is shown in table 7.1.

Table 7.1. Transmitter output voltage as a function of time (transmission of uppercase A at 300 baud)

Before Transmission	**5 volts**	
Transmission Begins		
(set t=0)	0 volts	Beginning of start bit
t= 3.3 msec	5 volts	Beginning of least significant bit (bit 0)
t= 6.6 msec	0 volts	Beginning of next bit(bit 1)
t= 23.3 msec	5 volts	(bit 6)
t= 26.6 msec	0 volts	Most significant bit (bit 7)
t=30 msec (and thereafter)	5 volts	Stop bit or bits

Prior to the transmission of a character, the UART is putting out a five volt signal. The UART signals that it will soon be transmitting a word by dropping the voltage to 0 for one *bit time*. In the example shown the bit time is about 3.33 milliseconds, or 300 bits per second.

(The number of bits per second is called the *baud rate* after J.M.E. Baudot, a French inventor who died in 1903 and pioneered in the field of serial communications.)

The next eight bit times vary between five volts or zero volts corresponding to the bits of the word being transmitted. For example, the numerical value of a capital A according to the ASCII standard code is 65, or 01000001 in binary. The bits are labeled bit 7, bit 6, and so on, down to bit 0 at the right end of the binary number. By convention, the least significant bit, bit 0, is transmitted first. Thus, for one bit time, the output is again at five volts. Then, for about 16.6 milliseconds, the transmitter puts out zero volts. This is because bits 1 to 5 of the ASCII "A" are zero. (The 16.6 millisecond period is composed of five *windows*, each 3.3 milliseconds long. These might have been five volts had some character other than "A" been transmitted.)

The "1" of bit 6 is then represented by five volts for 3.3 milliseconds followed by the 0 of bit 7, which appears as zero volts for 3.3 milliseconds.

A parity bit would appear at this point in the sequence if one was being sent. With even parity, the parity bit is selected so as to make an even number of 1's in the transmitted word; with odd parity, it is chosen to make an odd number of 1's.

Finally, one or more stop bits, composed of a logic 1 (a 5 volt output) are sent. The stop bits are indistinguishable from the period of logic 1 (5 volts) that lies between the time the stop bit of this word has finished and the time the next word begins.

The Inner Workings of the UART

The architecture of the UART is shown in figure 7.2. It is a forty-pin integrated circuit, type number IM6402, and is designated M22 in the Model 100. The transmit and receive rates are determined by the TRC and RRC (transmitter register clock and receiver register clock) signals entering the UART at the left.

Parallel data enter at the top, passing through the TBR into the TR, through a multiplexer and then out the TRO (transmitter register output) line.

Received data enter by way of the RRI (receiver register input) line, through a multiplexer to the RR (receiver register), thence to the RBR (receiver buffer register).

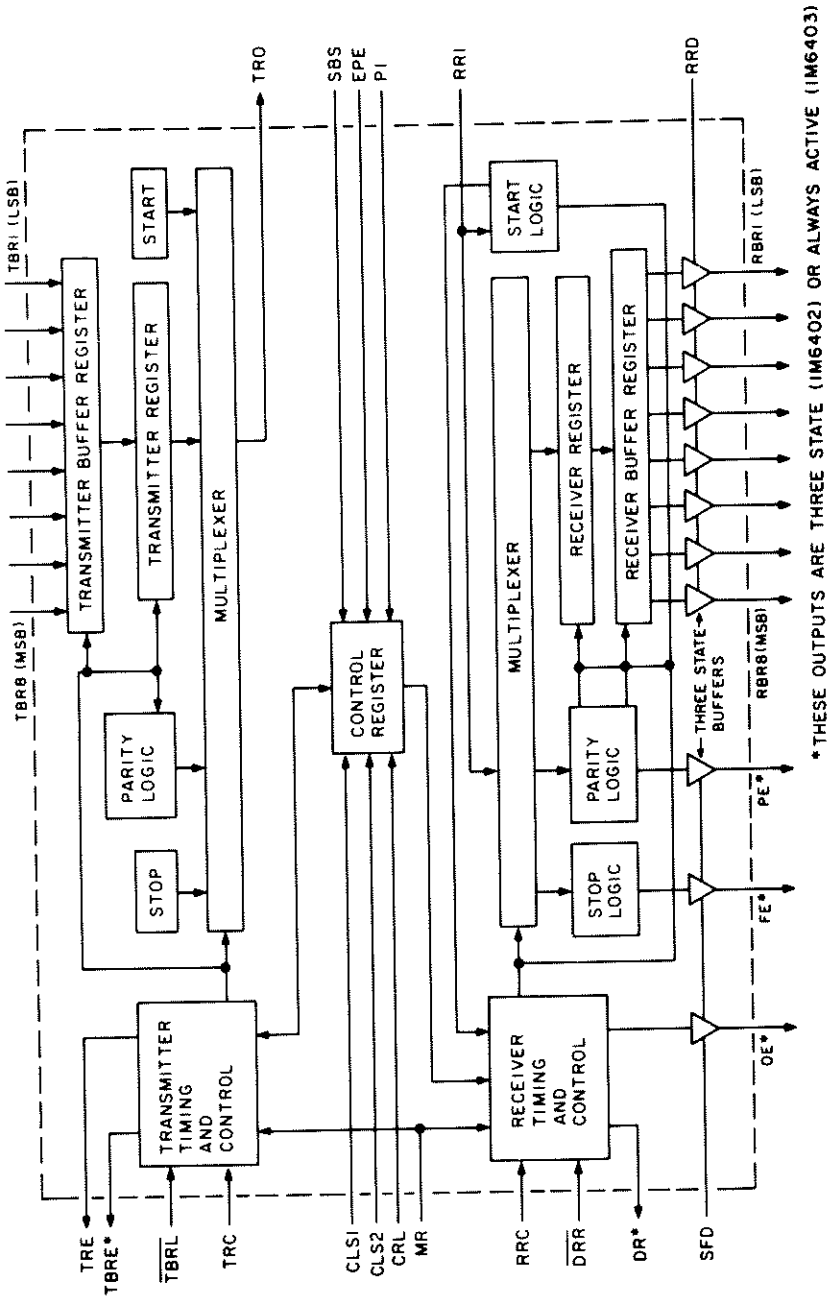

Figure 7.2. UART architecture

The UART requires certain one-wire signals for word length (CLS1 and CLS2), stop bit selection (SBS), and parity (PI, EPE). These signals are loaded (through CPU ports) as discussed in the following section.

CPU Communication with the UART

The CPU tells the UART how to configure the transmitted words (as well as what sort of words to expect to receive), provides a baud rate frequency for the UART, gives the UART characters to transmit, responds to the UART's signal that a character has been received, and learns whether any errors occurred in data reception. Most of these functions take place through the CPU input and output ports; one function occurs by way of a CPU interrupt. Each of these processes will be discussed in turn.

SERIAL WORD CONFIGURATION

Before serial I/O can take place, the CPU informs the UART how the transmitted words should be formed and what sort of words to expect to receive.

The UART parameters, namely word length, parity, and stop bit selection, are all loaded by the CPU through output port D8, which is listed in table 7.2 and shown in figure 7.3. Actually, any port number in the range D0 (decimal 208) to DF (223) will do.

Table 7.2. UART and other signals (output port D8)

Bit	Function
0	SBS (0=one stop bit; 1=two stop bits*)
1	EPE (0=odd parity; 1=even parity; ignored by UART if bit 2 is on.)
2	PI (1=no parity)
3	CLS1 (0=5 or 7 bits; 1=6 or 8 bits)
4	CLS2 (0=5 or 6 bits; 1=7 or 8 bits)
5	not used
6	not used
7	not used

* If the character length is set to 5, SBS=1 yields 1.5 stop bits.

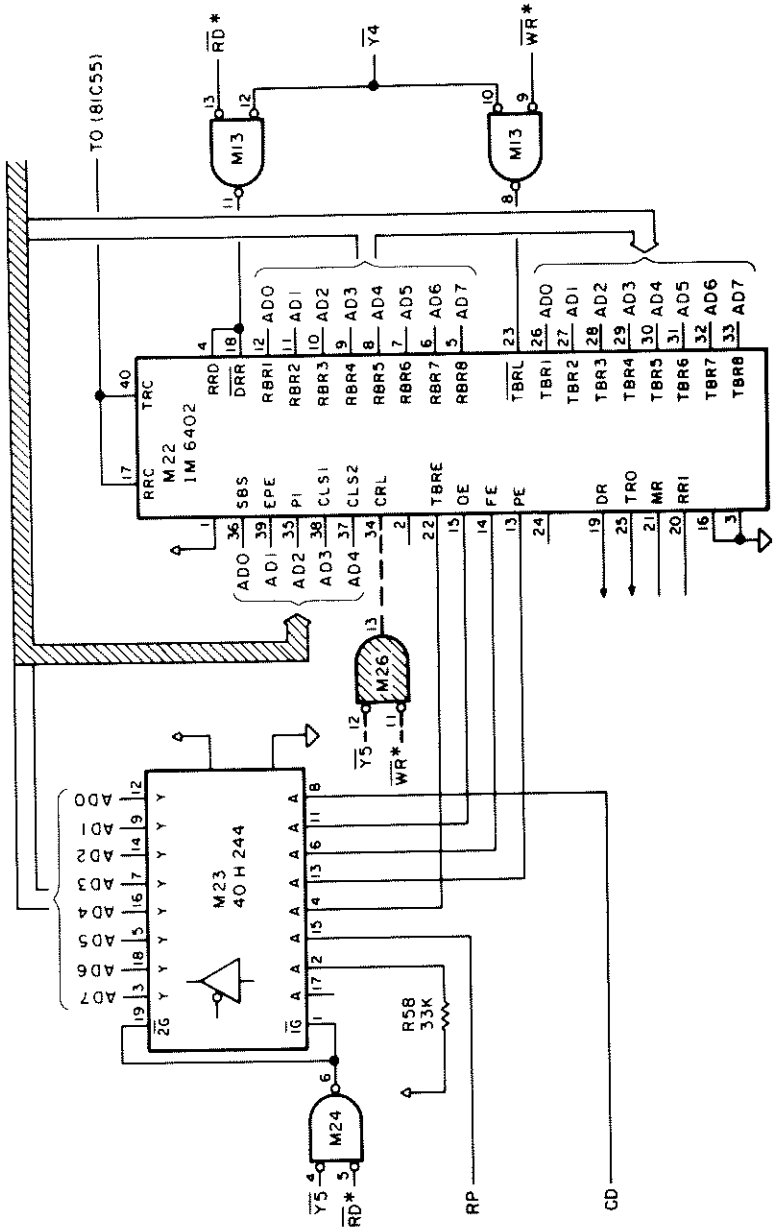

Figure 7.3. UART programming

SETTING THE BAUD RATE

CPU selection of baud rate is accomplished by loading a divisor into the PIO timer register through output ports BC or B4 (decimal 180 or 188) and BD or B5 (decimal 181 or 189).

Baud rate depends ultimately upon crystal X2, located near the CPU, which oscillates at 4.9152 megahertz. This is shown in photo 7.1. The CPU provides a CLK signal of 2.4576 megahertz (half the crystal frequency) to the PIO at its TIMER IN pin. The PIO divides the TIMER IN frequency by the divisor previously loaded into the PIO timer register. It then provides that reduced frequency via the TIMER OUT pin to the UART at both the TRC and RRC pins. (The PIO also provides the TIMER OUT signal to the piezoelectric beeper for sound generation). The UART in turn divides the PIO signal by 16 to arrive at both the transmit and receive baud rates. The divisors necessary to produce commonly used baud rates are shown in table 7.3. Note that while most of the baud rates are exact, the PIO output for 110 baud is in error by about 0.026%. A divisor of 1396 yields a TRC/RRC value of 1760.4585 hertz, which results in a data rate of 110.02865 baud.

Table 7.3. Baud rate divisors (decimal).

Baud Rate	PIO Divisor	Upper Byte Port BD	Lower Byte Port BC	TELCOM Stat Value
75	2048	72	0	1
110	1396	69	116	2
300	512	66	0	3
300	512	66	0	M
600	256	65	0	4
1200	128	64	128	5
2400	64	64	64	6
4800	32	64	32	7
9600	16	64	16	8
19200	8	64	8	9

Note that each value sent to output port BD has bit 6 on and bit 7 off. This is because port BD actually performs two functions. Bits 0 through 5 are the upper byte of the divisor, while bits 6 and 7 determine

the divider mode. As mentioned in chapter 5, the timer may be set by means of bits 6 and 7 for a single cycle of square wave (00), a single pulse (10), a continuous square wave (01), or continuous pulses (11). The UART requires a continuously provided frequency, so only the latter two modes work properly. To avoid any danger of the pulse being too brief for the UART to pick it up, the safest thing is to use the square wave. Thus, the byte sent to port BD has bit 7 off and bit 6 on.

One more CPU action, an output to port B8, is needed to provide the data rate clock signal to the UART. The PIO divisor must be enabled (port value C3) rather than disabled (port value 43).

SERIAL TRANSMISSION

Before sending a character by way of the UART, the CPU must confirm that any character in the TBR has already been received by the TR. The CPU does this by inspecting the condition of bit 4 of input port D8 (or D0 decimal 208 or 216). That bit carries the UART signal TBRE (transmitter buffer register empty) signal (*see* table 7.4). This is shown schematically in figure 7.4a.

The UART is designed so that once a character resides in the TR (and will presumably be transmitted presently), the CPU can load another character into the TBR. Presumably a certain (nonzero) amount of time passes between the moment the "empty" signal is sent to the CPU and the moment the next character is provided to the UART. The presence of the TBR means that the UART can, upon finishing the sending character, have another one ready to send which is "waiting in the wings."

Thus far nothing has been said about how the CPU loads the outgoing data into the UART. This is accomplished through output port C8 (or C0; decimal 192 or 200). The process is shown schematically in figure 7.4b.

SERIAL DATA RECEPTION

When an entire character has been received and loaded to the RBR (receiver buffer register), the UART indicates this by producing a logic "1" at its DR (data received) pin. In the Model 100, this is wired to the RST 6.5 interrupt pin of the CPU. As is discussed in detail in

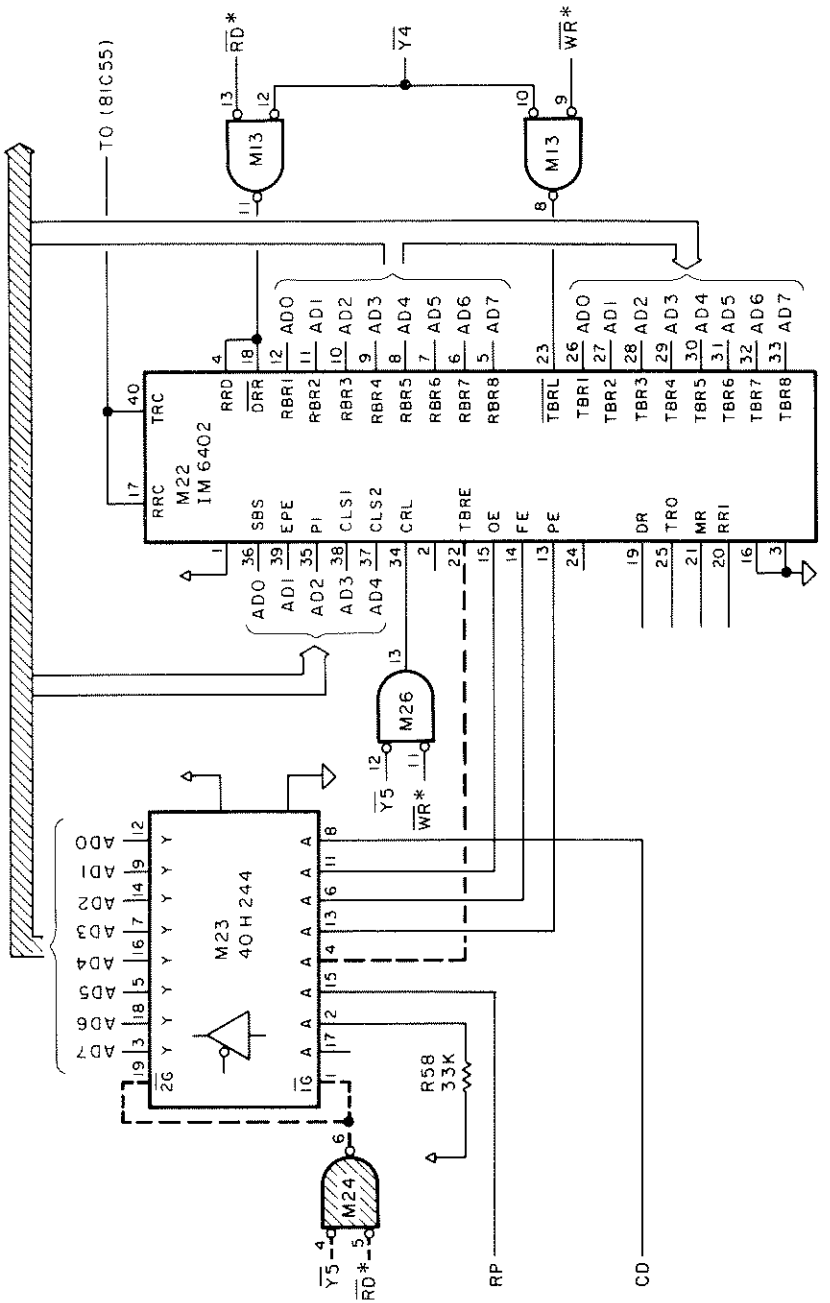

Figure 7.4a. UART data transmission

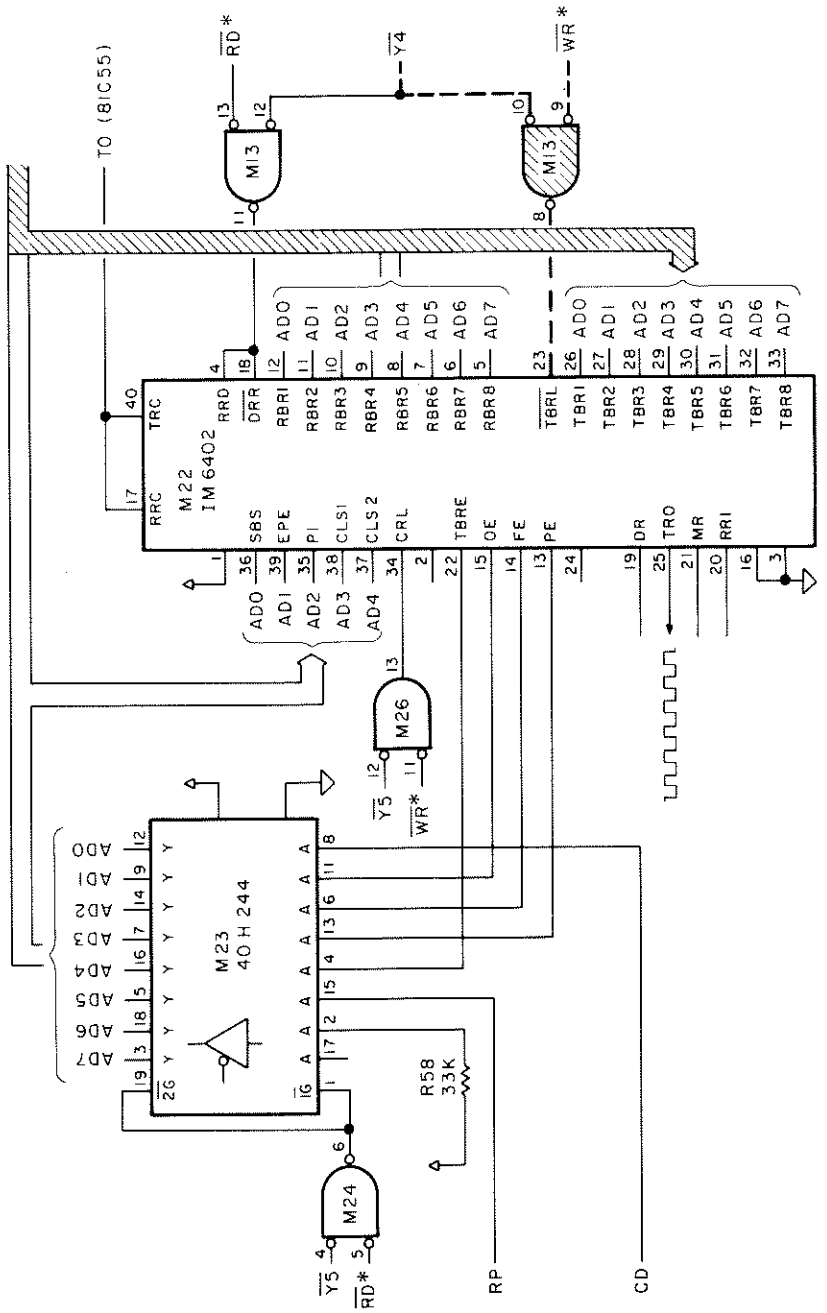

Figure 7.4b. UART data transmission

chapter 15, this causes a subroutine call to ROM location 0034, which disables interrupts and jumps to 6DAC. The CPU obtains the received byte through input port C8 (or C0 decimal 192 or 200). There, the CPU places the received character in a RAM buffer, as shown in figure 7.5.

As is discussed in chapter 15, the usual ROM handling of a received character can be circumvented by disabling interrupts or by changing a RAM vector at F5FC.

DATA RECEPTION ERRORS

Any number of things can go wrong in serial reception of data:

- The CPU could take too long to pick up a received character
- A "0" might occur when a stop bit one was expected
- The number of 1's in the received byte might differ from that indicated by the parity bit.

Such errors are known as *overrun errors* (OE), *framing errors* (FE), and *parity errors* (PE), respectively. These signals are made available to the CPU at input port D8 (or D0 decimal 208 or 216); the connection is shown in figure 7.4a. The various bits of the input port are shown in table 7.4. To obtain the UART error, the value returned at the port should be ANDed with 0E.

When a receive error has occurred, the received value at input port C8 may or may not be correct.

Table 7.4. UART and other signals (input port D8) (the information contained in bit 0 varies depending on the RS-232/modem mode)

Bit	In RS-232C Mode	In Modem Mode
0	Always 0	1=carrier detect
1	OE (1=UART overrun error)	same
2	FE (1=UART framing error)	same
3	PE (1=UART parity error)	same
4	TBRE (1=UART transmitter Buffer register empty)	same
5	RP (0=ground at PHONE pin 8)	same

Continued on following page.

Bit	In RS-232C Mode	In Modem Mode
6	Hobby use (0=ground at M23 pin 2)	same
7	Low-power signal (0= battery low)	same

Significance of ASCII Code

Nothing in the UART hardware, nor in the ROM routines relating to the UART, requires that the information sent or received be coded according to the ASCII (American Standard Code for Information Interchange) standard. In plain language there is no reason that the value 32 (decimal) must represent a space nor that the value 65 (decimal) represent an uppercase A. You can devise your own coding system if you desire using the Model 100 hardware to communicate. For example, many teletype machines use a five-bit Baudot code and many IBM machines use an EBCDIC code (Extended Binary Coded Decimal Interchange Code), each of which assigns values different from ASCII to the various letters of the alphabet, numerals, and so on.

You are perfectly free within software to translate between ASCII (used in the Model 100 keyboard and display screen) and any other code used by a distant device.

The only exception to this is that in the ROM UART routines, the values 17 and 19 decimal are treated differently than the other 254 values, due to their ASCII-assigned meanings as XON and XOFF. Many problems with XON and XOFF can be avoided simply by disabling XON/XOFF, or one can write UART routines (borrowing from the ROM routines) that are neutral regarding the values 17 and 19.

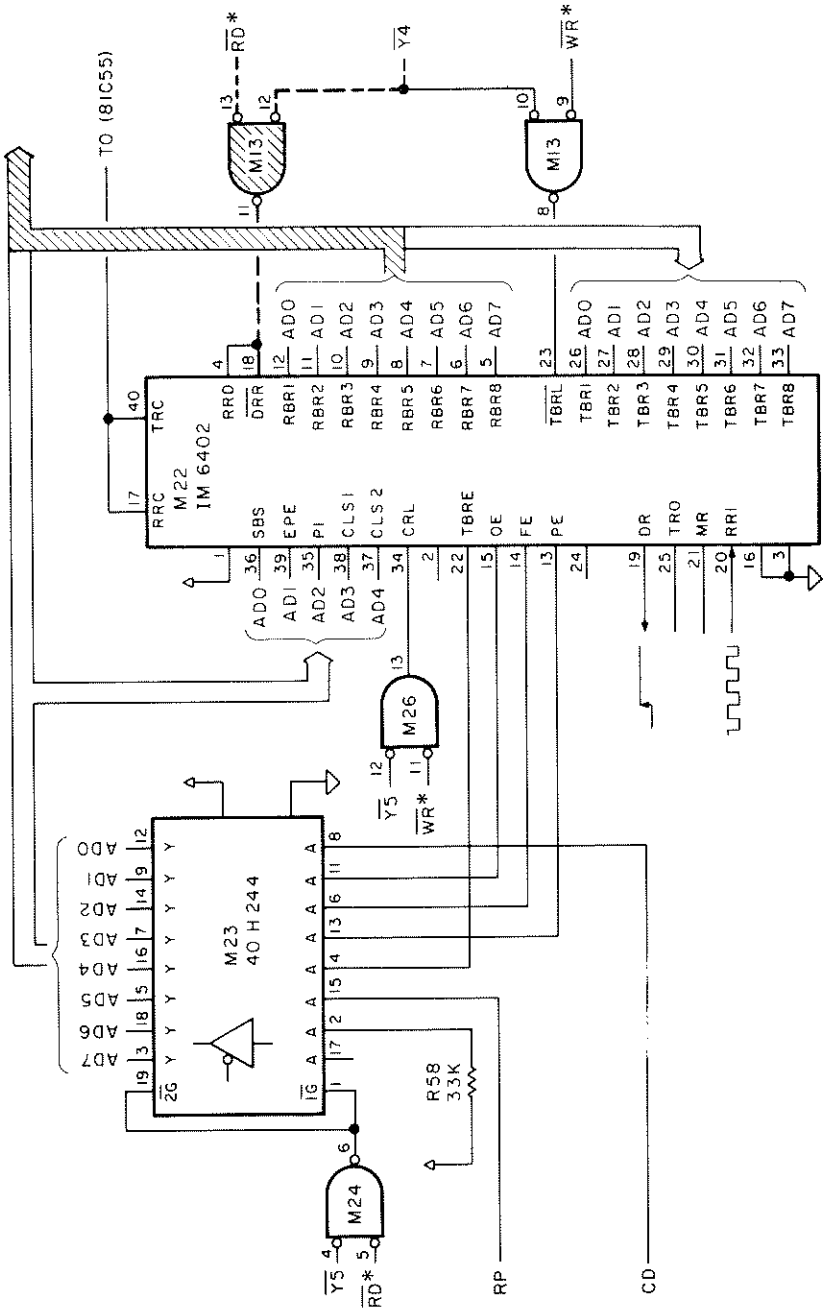

Figure 7.5. Received UART data

The Beeper

The BEEP routine, which can be invoked by sending an ASCII 7 to the screen, works fine, regardless of what the baud rate divider is doing. As a result, an ASCII BELL sent by the remote device beeps at the Model 100. (This is explained in detail in chapter 9.)

ASCII Protocol — XON/XOFF

If XON/XOFF is enabled, the Model 100 will monitor the incoming stream of characters for a CTRL-S. If a CTRL-S is received, the computer will delay sending any characters until such time as a CTRL-Q is received. Three flags are kept relating to XON/XOFF: FF40 indicates whether XON/XOFF was enabled (nonzero) or disabled (zero) during the Stat initialization, FF41 indicates whether the Model 100 most recently transmitted a CTRL-S (nonzero) or a CTRL-Q (zero), and FF42 indicates whether the other device most recently sent the Model 100 a CTRL-S (nonzero) or CTRL-Q (zero).

Mode Selection: RS-232 and Modem

The UART serves as the serial-to-parallel and parallel-to-serial convertor for the RS-232 and telephone modem interfaces. A device termed a *multiplexer* connects the UART either to the RS-232 interface or to the telephone modem interface. The multiplexer will be discussed here. Then, with the assumption that the multiplexer is set in RS-232 mode, the balance of the chapter will be devoted to the RS-232 interface. The modem mode is discussed in chapter 8.

The UART multiplexer is a handful of components set up to act like a big six-pole, double-throw switch. The position of the switch is determined by bit 3 of output port BA. Since that port controls many other functions including power control (*see* chapter 5), one must be careful how port BA is set. A safe setting for selecting the modem is 2D hex, and a safe setting for RS-232 mode is 25 hex. (*See* for example, the ROM code at 6EAA through 6EB8, in which these values are used.) When the computer is powered up, the multiplexer is in *modem* mode. A change to RS-232 mode is caused by any of the following: setting Stat to a baud rate other than M, opening COM: as a file in BASIC, or loading 0 to bit 3 of output port BA directly.

The values switched by the multiplexer are detailed in table 7.5.

Table 7.5. Multiplexer configuration

Signal	RS-232 mode	Modem mode
RTS*-from output port BA, bit 7	RTSR-to RS-232 pin 4	RTSM-to RY-2 (phone line)
TRO-from UART Transmitter Register-output port C8	TXR-to RS-232 pin 2	TXM-to modem transmitter circuitry
RRI-to UART Receiver Register-input port C8	RXR*- from RS-232 pin 3	RXM- from modem receiver circuitry
CTS*-to input port BB, bit 4	CTSR- from RS-232 pin 5	CL/AS- from SW-1 ORIG/ANS
DSR*-to input port BB, bit 5	DSRR- from RS-232 pin 6	CP/TL- from SW-2 ACP/DIR
CD-to input port D8, bit 0	logic zero	RXCAR-through 10K resistor

The items in the column headed RS-232 mode are discussed in this chapter; the items in the column headed Modem are discussed in chapter 8. The circuitry of the multiplexer is shown in figure 7.6. Curiously, the DTR signal is not switched.

It is interesting to note that the multiplexer status is one of the few things not preserved when a running BASIC program is powered down, then back up. To see this, set Stat to a numerical baud rate, and run this program in BASIC:

```
1 A=INP(187) AND 32: PRINT A: GOTO 1
```

The DIR/ACP switch has no effect on the display. Then, turn the computer OFF and back ON. Suddenly the switch will change the value on the screen.

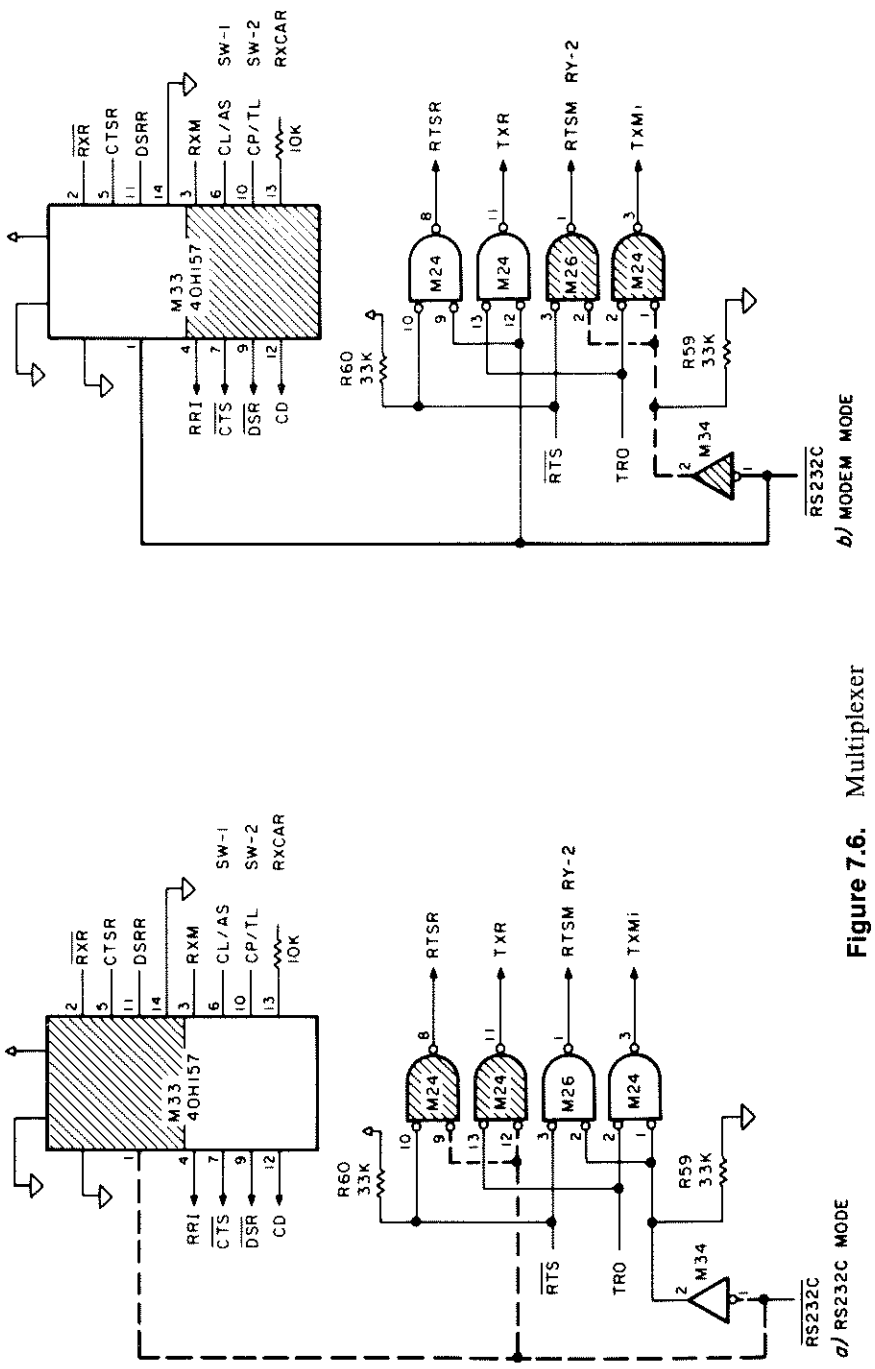

Figure 7.6. Multiplexer

The RS-232 Standard

Decades ago the Electronic Industries Association promulgated the RS-232 standard, designed to facilitate the design of interfaces between data terminal equipment (the Model 100) and data communications equipment (modems). (The letters RS in the designation have nothing to do with Radio Shack.) The standard found widest application in the connecting of teletype-like devices with keyboard and printer (data terminals) to mainframe computers and modems (data sets). Typically, two directions of communication were intended and a variety of *handshaking* signals were designed in so that each device could inform the other whether it was able to receive and transmit data.

Since its origins, the RS-232 standard has undergone three revisions culminating in the present version, RS-232C. (Here the designations RS-232 and RS-232C will be used interchangeably.)

As we shall see, the Model 100 satisfies parts of the standard and fails to satisfy other parts. The same is true of every computer product on the market today.

MECHANICAL REQUIREMENTS

The most visible aspect of the RS-232 standard is the specified connector, a twenty-five-pin plug and jack in the DB style. The terminal is supposed to accept the male connector, so the Model 100 violates that requirement.

Most of the twenty-five pins have defined functions, but only eight still have common use. Plugs, jacks, and cables are commonly available; many RS-232 cables have only a few of the pins wired from one end to the other. The plug is Radio Shack catalog number 276-1547 or 276-1559; this will plug into the Model 100. Unfortunately the hood, 276-1549, may not be used as it will not fit into the case of the Model 100. The jack is catalog number 276-1548 or 276-1565. A complete cable suitable for Model 100 use can be purchased ready-made (26-1408) or assembled by hand (276-1551, 276-1559, and 276-1565).

ELECTRICAL REQUIREMENTS

Each signal in the twenty-five-pin connector has a specified originator and recipient, either the data set or the data terminal. As to each

signal, the originator is obligated to use -5 volts or less to mean a logic 1 (or *denial* of a handshake signal) and +5 volts or more to mean a logic 0 (or *assertion* of a handshake signal). The originator must meet this voltage requirement even under a 3000-ohm load. The originator is forbidden to use voltages between +5V and -5V for any purpose other than during the brief instant of transition from positive to negative, or vice versa.

It is a fact of the Model 100 design that its 0 and 1 states are a mere five and minus five volts, respectively. Under a 3000-ohm load, the voltages drops to well under five volts. As a result at the other end of the cable, the signals can easily be in violation of the RS-232 electrical requirements. This is not usually a problem since devices at the other end are likely to be able to receive the signals with as little as a one-volt swing about zero.

In the absence of a connection to the Model 100 RS-232 connector, the hardware reads all incoming signals as negative voltage, or logic "1's." When an incoming signal on any of those pins becomes more positive than about 1.5 volts, the value presented to the CPU changes to a logic 0. (This betters the threshold promised by the Radio Shack specification, which is 3 volts.) Thus, assuming a reasonably short cable, one Model 100 will have no trouble detecting the plus five volt and minus five volt signals from another Model 100.

The RS-232 standard requires that the inputs withstand as much as positive or negative 25 volts, but according to the Radio Shack Service Manual, the RS-232 inputs are designed to withstand only eighteen volts. The inputs are also required to give off no more than plus or minus two volts in open circuit; in the Model 100 they give off minus five volts.

A further RS-232 requirement is that pin 7, called AB, should be the signal ground used by each device as the zero voltage reference for the signals originating at the other end. Each device is to use that pin as the ground reference for the drivers sending voltages to the other end.

As a separate matter, pin 1, with RS-232 designation AA, is to be used as a protective (earth) ground to minimize the possibility that a person touching the chassis of the two devices would be injured by electric shock due to a voltage difference between the two. The specification requires that it be possible for the user to either tie pins 1 and 7 together or separate them within the computer.

Unfortunately, in the Model 100 the two pins are tied together permanently. This reduces noise immunity for both directions of serial data.

DATA FLOW

According to the standard, the data set sends out data on pin 3, and the data terminal sends out data on pin 2. The signals are named from the terminal's point of view, with the pin 2 signal called BA (transmitted data) and the pin 3 signal called BB (received data). The Model 100 acts like a data terminal, talking on pin 2 and listening on pin 3. (If it is to talk with another Model 100, a so-called *null modem* is required, which switches lines 2 and 3 between two connectors. You can easily make one yourself.)

HANDSHAKING SIGNALS

The terminal indicates its powered-up status and requests that the modem access the phone line by asserting the DTR (data terminal ready, RS-232 designation CD) signal provided to the modem at pin 20.

If the data set were a modem, it would let the data terminal know when it is powered up and connected with the transmission line by activating what is called the DSR (data set ready) line, pin 6, which has the RS-232 designation CD.

Before the terminal would send a character to the modem (to be transmitted down the phone line) it would ask permission of the modem by activating RTS (request to send), pin 4, which has RS-232 designation CA. Assuming the modem is able to send another character, it grants permission by activating CTS (clear to send, pin 5, RS-232 designation CB), which is received by the terminal.

In the Model 100, as we shall see below, hardware provision has been made for the signals named above. The TELCOM software and the ROM routines ignore DSR and CTS, which come in from outside, and assert RTS and DSR to the outside device. (When power is applied to the Model 100, RTS and DSR are not asserted. Then when the TELCOM program is run and the TERM mode is selected (function key f4), RTS and DSR are both brought to a positive voltage (asserted).

HOW THE RS-232 INTERFACE WORKS

Handshake signals and serial data enter and leave the Model 100 through the RS-232C connector, as shown in table 7.6.

Table 7.6. RS232 signals handled by the Model 100

RS232 Pin,Des	Mod 100 Symbol	Source	Port Location
1-AA	(protective ground)		
2-BA	TX	Inside	Out C8
3-BB	RX	Outside	In C8
4-CA	RTS	Inside	Out BA, bit 7
5-CB	CTS	Outside	In BB, bit 4
6-CC	DSR	Outside	In BB, bit 5
7-AB	(signal ground)		
20-CD	DTR	Inside	Out BA, bit 6

These signals, according to the RS-232 convention, are negative for a data one or non-asserted value, and positive for a data zero or asserted value. However, the internal circuitry of the Model 100, like all computers, uses +5 volts for a logic one and ground or 0 volts for a logic zero level. The conversions, three incoming and three outgoing, are performed by integrated circuit M35. The RS-232 interface circuitry is shown in figure 7.7.

The incoming handshake signals, clear-to-send and data-set-ready, are then made available to the CPU at bits 4 and 5 of input port BB. A negative voltage at the input pin appears to the CPU as a zero at the input port, while a positive voltage comes through as a one.

The outgoing handshake signals, data-set-ready and request-to-send, are controlled by the CPU through bits 6 and 7, respectively, of output port BA. In each case, 0 and 1 sent out by the CPU result in positive and negative voltages, respectively.

RECEIVING RS-232 DATA

Serial data enters the Model 100 through RS-232 pin 3 and, after level shifting and signal switching, is fed to the UART.

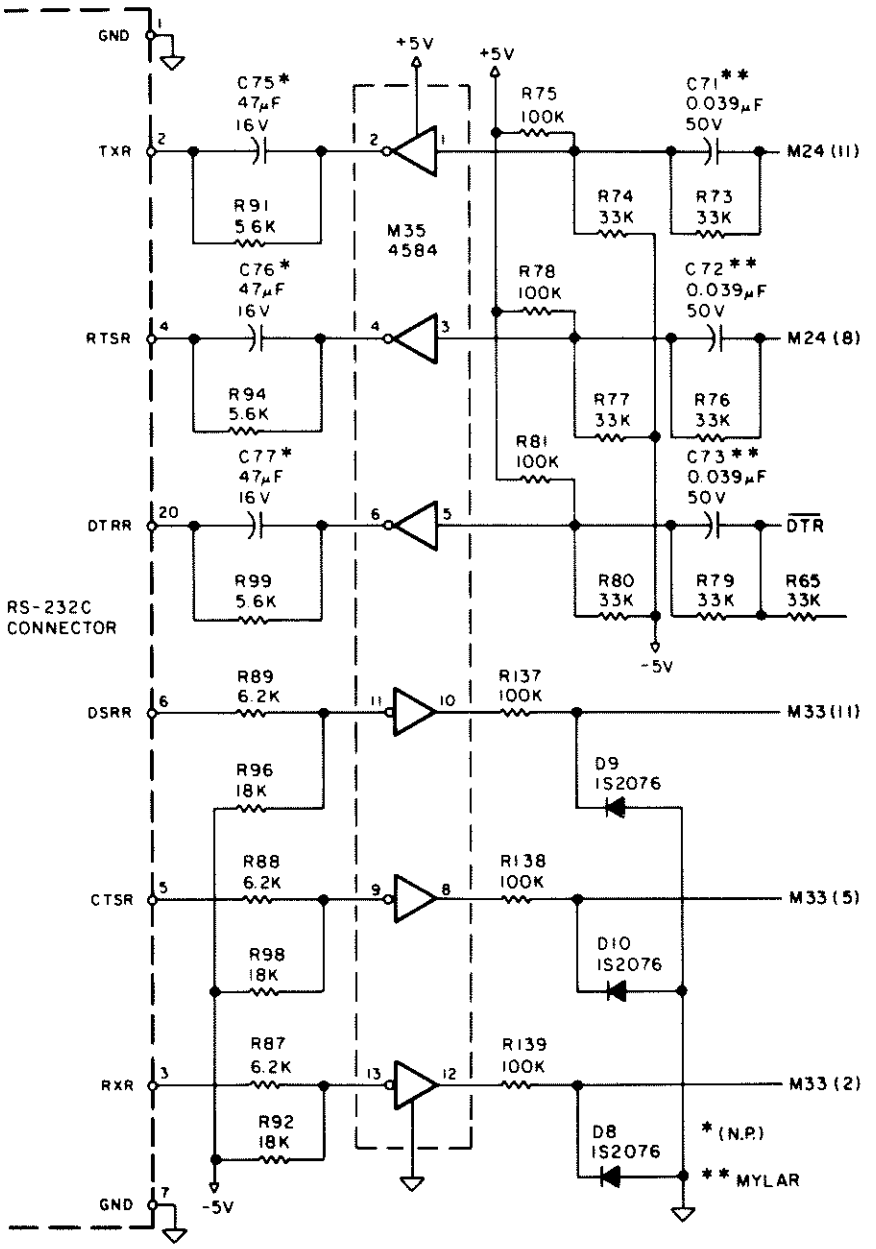

Figure 7.7. RS-232 interface

TRANSMITTING RS-232 DATA

The serial output from the UART goes through switching circuitry to RS-232 level shifter M35, and from there to RS-232 pin 2.

PUBLISHED ROM SUBROUTINES

The UART baud rate can be set by the routine BAUDST, called at 6E75. Prior to the call, H contains a numerical value in the range of 1 through 9, and the routine sets the baud rate using a ROM table at 6E94 through 6EA5. The table address used in the load is stored in FF8B. Note that the routine will not work properly if H contains undefined values, such as ASCII values for characters 1 through 9 or M. The routine extends to 6E93.

The routine INZCOM, called at 6EA6, performs the baud setting function of BAUDST and configures the UART for word length, parity, and so on. Prior to the call, H must contain the baud rate number, just as with BAUDST. In addition, L must contain "1's" and "0's" to select the UART parameters desired, from table 7.2. (Some Radio shack documentation is incorrect on this; table 7.2 should be used.) The condition of the carry flag determines whether the routine sets the multiplexer for RS-232C (if carry is set) or modem mode (if carry is reset). The routine initializes a UART data-received buffer.

It does not matter what bit values are chosen for bits 5 through 7 of the L register; the hardware ignores them and the ROM routine trims them off anyway.

The routine SETSER, called at 17E6, sets the baud rate and UART word length parameters based on an ASCII text string similar to the one which follows the device designator COM: or MDM:. The routine also initializes a UART data-received buffer, and updates a flag located at FF42 controlling XON/XOFF. A zero at FF42 means XON/XOFF is enabled; a nonzero value means it is disabled.

Before calling the routine, HL must point to the ASCII text string. The carry flag determines whether the routine switches the multiplexer to RS-232 or modem mode, just as in the INZCOM routine. If modem mode is to be used, the D register must be loaded with the value of two prior to the call, and the text string must start not with the baud rate "M" character but with the word length digit.

The correctness of the text string is fully checked, and the alphabetic characters can be uppercase or lowercase. When the routine finishes, if the Z flag is set, something was wrong with the text string. Otherwise, you may assume the baud rate and parameters are set. There is no need for the string to end with a zero, as the routine simply reads enough bytes to get what it needs.

This routine updates the storage location STAT, at F65B through F65F, which contains in ASCII format the Stat information, baud rate (M if modem), word length, parity, stop bits, and XON/XOFF switch.

The routine CLSCOM, called at 6ECB, deactivates the RS-232 or modem circuitry. More precisely, it "un-asserts" RTS, if it is in RS-232 mode. If in modem mode, it simply hangs up the phone. In either case, it "un-asserts" DTR.

The routine RCVX, called at 6D6D, checks the UART data-received queue to see if any characters have been received since the queue was last emptied. The A register contains the number of characters in the queue, and the Z flag will be set if no received data is pending, or reset otherwise.

This routine can be used regardless of whether the Model 100 is in RS-232 mode or modem mode.

The routine RV232C, called at 6D7E, retrieves a character from the UART data-received queue. If no character was present in the queue at the time of the call, the routine does not return until a character is received or SHIFT-BREAK is pressed. (CTRL-C will not cause a return; only the precise combination SHIFT-BREAK will do so.)

The call is appropriate regardless of whether the computer is in RS-232 or modem mode. The received character is in the A register, and the conditions of the Z and C flags indicate whether errors occurred. The Z flag is set if the character was received properly, and reset if a PE, FE, or OE occurred. If the routine returned because SHIFT-BREAK was pressed, the carry flag will be set; or reset otherwise.

The routine SENDCQ, called at 6E0B, sends an XON (CTRL-Q, ASCII 11H) character to the remote device, but only if XON/XOFF was enabled at the time of UART initialization. (Recall the XON/XOFF flag at FF42 is zero if XON/XOFF is enabled and nonzero if disabled.)

The routine SENDCS, called at 6E1E, sends an XOFF (CTRL-S, ASCII 13H) character to the remote device, but only if XON/XOFF was enabled at the time of UART initialization. (Recall the XON/X-OFF flag at FF42 is zero if XON/XOFF is enabled and nonzero if disabled.)

The routine SD232C, called at 6E32, sends a character to the UART to be transmitted to the remote device. The character to be sent should be present in the A register prior to the call.

It is possible for the routine to return without successfully sending the character. Suppose the UART takes so long to transmit (or the other device sends the Model 100 a CTRL-S for such a long time) that the user presses SHIFT-BREAK. The routine returns with the carry flag set.

If the UART was originally configured with XON/XOFF disabled (FF42=0) then the routine simply sends the character. If XON/XOFF is enabled, then the routine makes a point of keeping track (the flag at FF41) of which has been sent out by the Model 100 most recently— CTRL-S (flag=-1) or CTL-Q (flag=9).

Before you use the UART, it is a good practice to clear the UART receiver buffer register with an input from port C8. This is done, for example, at 6CE5.

8

The Telephone Modem

The Model 100's built-in modem and autodial capabilities are among its most popular features. This chapter consists of a discussion of the modem and autodial feature. The hardware of the autodial, direct-connect modem is explained first followed by a description of the acoustically-coupled modem. ROM subroutines are included for both types of modem.

Data Flow Overview

Recall from the discussion in chapter 7 that serial-to-parallel and parallel-to-serial conversions take place within the UART, and that a

multiplexer determines whether the UART is connected to RS-232 circuitry or modem circuitry. This is shown in functional block diagram form in figure 8.1.

The multiplexer, shown here in the "RS-232" position, sends incoming RS-232 data to the receiver portion of the UART which is then sent to the CPU. The multiplexer also connects the output of the UART transmitter to the RS-232 output circuitry.

If and when the multiplexer switches to the other position, the data paths change in two ways. As shown in figure 8.1a, the UART receiver gets its signal not from the RS-232 circuitry but instead from the modem receiver (which in turn gets its signal from the modem filter). The UART transmitter is connected to the modem transmitter, which in turn feeds a wave shaper, as shown in figure 8.1b.

Assuming the multiplexer is in "modem" mode, then the DIR/ACP and ORIG/ANS switches also have an effect on the data paths.

The DIR/ACP switch determines whether the modem filter gets its signal from the phone line transformer OT1 or from the electret-condenser microphone in the earpiece cup of the acoustic coupler. In addition, it connects the output of the wave shaper either to OT1 or to a tiny speaker in the mouthpiece of the acoustic coupler.

Finally, the ORIG/ANS switch changes the internal function of the modem filter, the wave shaper, and the transmitter and receiver portions of the modem integrated circuit.

The various circuits shown in the functional block diagram will be explained in detail, but first it will be necessary to explain a bit about the Bell 103 standard, and about telephones generally.

The Bell 103 Standard

All 300 baud data transmission in North America is performed according to Bell Standard 103, which spells out the manner in which computers are to send 1's and 0's to each other.

The technique used is frequency-shift keying (FSK), which means that each computer produces an audio tone, and shifts in the frequency of the tone indicate whether a "1" or "0" is being sent. The Bell 103 standard calls for each computer to transmit a "1 " most of the time, dropping the frequency by 200 Hertz whenever a "0" is to be sent.

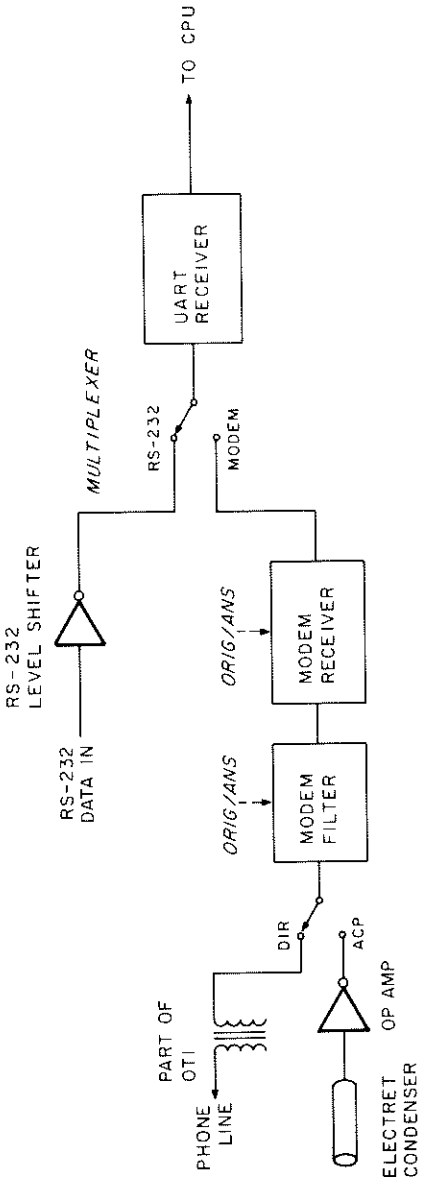

Figure 8.1a. Receiver data path

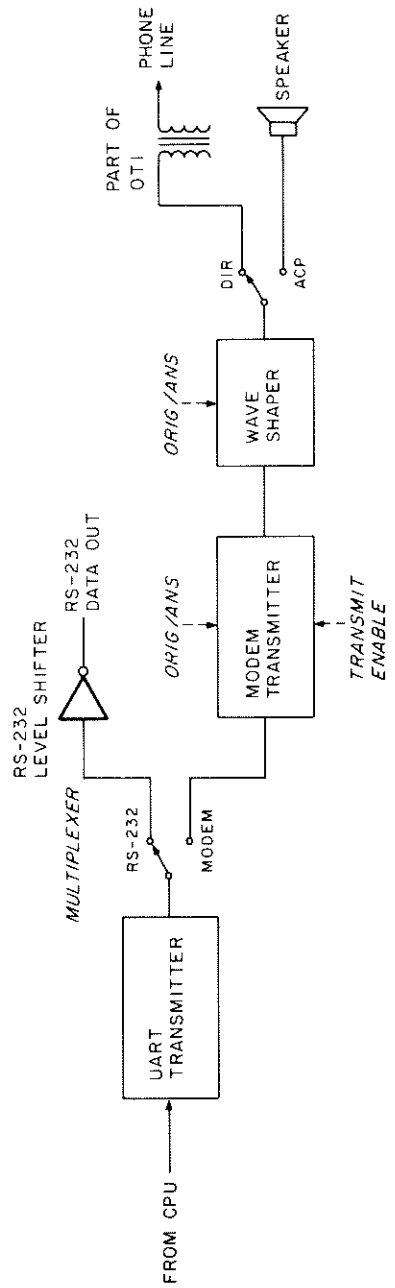

Figure 8.1b. Data path switching functional block diagram

The "1" state is called "marking," and the "0" is called a "space". This terminology originated with the first teletype machines. Historically the usual circuit condition between two teletypes was a current flow of 20 milliamperes, called a *mark*. Brief interruptions in the current were called *spaces*. An analogy can be drawn between frequency-shift keying and the dialing of a phone.

To this day, most telex machines use this so-called *current loop* means of data communication which allows them to be interfaced with any RS-232 device such as the Model 100, by using optoisolators.

THE AUDIO FREQUENCIES

In order for two-way data communication to take place, each computer must be prepared to listen carefully to the audio tone transmitted from the other device, to determine whether a "1" or "0" is being sent. At the same time the computer must generate audio tones to send 1's and 0's to the other device.

Problems arise when both computers use the same tone because each computer will hear itself as well as the other computer. The solution is to assign different frequencies to the two devices.

When the Bell 103 standard was written, most modem communications was quite lopsided. One device was a large computer, which answered calls, while at the other was a small device, usually a terminal, which originated calls. The originating device was assigned a mark frequency of 1270 Hertz, while the answering device was assigned 2225 Hertz. The established protocol was that the answering device, upon answering the phone, would emit the 2225 Hertz tone. Upon hearing it, the originating device would start producing the 1270 Hertz tone.

The two tones, known as *carrier* tones, were expected to be present (at either the stated frequency or 200 Hertz down) during the duration of the phone call. If either computer's tone disappeared for even an instant, the other device would assume it had been hung up on, and would disconnect itself. This is the reason that the call waiting feature causes so much trouble for modems.

Currently in many situations (such as communications between two Model 100's) the selection of one device as the *originate* device and the other as the *answer* device is purely arbitrary.

The other major standard, used in Europe, is the CCITT standard. It is similar in most respects to the Bell 103, except for the particular frequencies used to represent 1's and 0's.

How Telephones Work

THE PHONE WIRES

Although standard modular telephone jacks contain four wires—red, green, yellow and black, most telephones use only two of them, the red and green wires. The red wire is sometimes called the *ring* signal, and the green wire is sometimes referred to as the *tip* signal. The terms ring and tip have nothing to do with the ringing of the phone bell; they come from the physical description of the barrel (ring) and end (tip) of the two-conductor phone plugs traditionally used by switchboard operators in connecting calls.

ELECTRICAL CONSIDERATIONS

If no phone is plugged into the jack, or if a phone is plugged in and *on-hook* or hang-up, the potential on the line will be about forty volts. An on-hook phone represents a very high resistance across the two wires, so that virtually no current flows.

PLACING TELEPHONE CALLS

When the phone is taken *off-hook*, that is, when it is picked up to place or answer a call, the phone presents a low resistance across the two wires. The central office detects this, and presents a dial tone or connects the incoming call, whichever is appropriate. The low resistance is typically 600 ohms. With such a load on the line, the voltage from the central office drops substantially to perhaps ten volts.

Once a dial tone is audible, the next step is generally dialing a number. One of two methods may be used, depending on the nature of the dial tone circuitry provided. With all dial tones, one may use rotary dial pulses; with some dial tones, DTMF (dual tone multifrequency, e.g. Touch-Tone) may also be used.

Rotary dial pulses are sent to the central office by repeatedly removing from the phone line the low resistance path that was present

when the phone was taken off-hook. In a traditional rotary-dial phone, this is accomplished using a simple mechanism of springs, gears and switches. With many modern pulse-type, pushbutton phones, solid-state circuitry mimics the dial mechanism.

There is nothing mysterious about the action of the telephone dial. It merely hangs up the phone (places it on-hook) one or more times for only a fraction of a second. You can do this manually with any phone by simply tapping the hang-up button. (The on-hook and off-hook times should last about 65* and 35* milliseconds, respectively.) After a pause of about 300* milliseconds, the next digit is dialed.

ANSWERING TELEPHONE CALLS

Now that the number has been dialed, let's examine what takes place when the phone rings. The central office causes a phone to ring by sending AC (alternating current) at perhaps forty or fifty volts to the phone jack. The telephone uses a coupling capacitor to allow the AC to ring the bell.

RINGER EQUIVALENCE

The amount of energy absorbed by a device connected to the phone line when the ringing voltage is present is reflected in the ringer equivalence number (REN) of the device. An REN of 1 means the device takes as much power as a traditional Western Electric phone. The Model 100 has a REN of zero, because when relay RY2 is open the computer has no current path capable of absorbing an appreciable amount of the ringing energy.

Simply taking the phone off-hook signals to the central office to stop sending the ringing voltage and to connect the calling party.

THE DIRECT-CONNECT MODEM AND THE TRANSFORMER OT1

The Model 100 uses the red and green wires for direct-connect modem operation. The red wire enters through pin 7, as shown in figure 8.2.

* For 20 pulse-per-second dialing, all these times are divided by two.

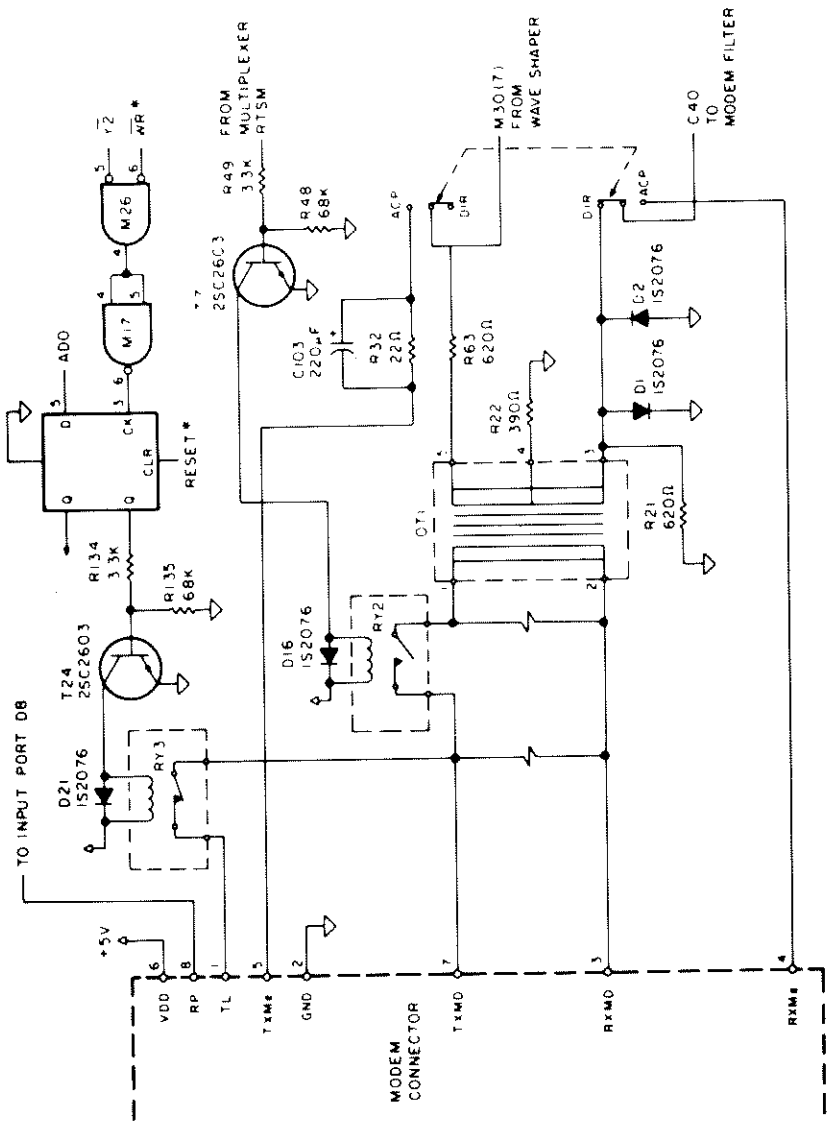

Figure 8.2. Direct connect interface

Two relays control the direct telephone conections of the Model 100. Though it is normally open, relay RY2, when energized, connects the primary side of isolation transformer OT1 to the tip and ring signals. Relay RY3, which is normally closed, makes the connection between the telephone instrument and the telephone line. Relay RY2 accomplishes the resistance transition from infinity down to six hundred ohms to pick up the phone, and to dial phone numbers. Relay RY2 could be termed the *hookswitch* relay.

Recall that modem cable 26-1410 has two modular plugs, with beige and silver cords. The beige cord plugs into a modular jack, bringing the tip and ring signals to pins 3 and 7 as mentioned above. The ring signal at pin 3 goes directly to the phone instrument (if connected) through the red conductor of the silver modular cord. The tip signal from the beige cord enters the Model 100 at pin 7 and is usually fed through relay RY3 to pin 1. (These signals are summarized in table 8.1.) From there the tip signal passes through the silver cord's green conductor to the instrument.

Table 8.1. Phone jack pin designations.

Pin	Designation	Function
1	TL	Green to phone instrument
2	GND	Ground reference
3	RXMD	Red from phone line ("ring" signal) also goes to phone instrument
4	RXMe	From coupler earpiece electret-condenser microphone
5	TXMe	To coupler mouthpiece
6	VDD	+5V for coupler amplifier
7	TXMD	Green from phone line ("tip" signal)
8	RP	Ring pulse signal (see text)

When direct-connect modem data transmission is desired, it is necessary to energize both relays. (This is accomplished by the TELCOM software.) This conects the phone line to the modem circuitry and disconnects the telephone instrument to avoid interference if the phone is picked up. It is impossible, of course, for the Model 100 to protect against interference caused by picking up an extension phone on the same line, or from problems caused by loss of the audio carrier signal such as a call-waiting beep.

When TELCOM is used as an automatic dialer for voice conversations, relays RY2 and RY3 are both energized. Relay RY2 is left open for about a second to insure that any previous call is disconnected and is then turned on again. After allowing a couple of seconds for the dial tone to arrive, RY2 is repeatedly switched on and off to simulate a rotary telephone dial. Then both relays are de-energized, leaving the phone instrument connected to either ringing or a busy signal.

The dialing process is simple. To dial, say, a "4," relay RY2 is switched off and on four times.

RING PULSE

Provision has been made for the Model 100 to be expanded into an autoanswer device. This prospect is discussed further in chapter 17.

FCC CERTIFICATION

On the bottom panel of the Model 100 are labels describing two kinds of FCC certification. The first, which bears FCC identification number AWQ9SB26-3802, indicates that the computer has been tested and found to be sufficiently shielded. This means that it does not radiate in excess of levels of radio frequency (RF) energy set forth in part 15 of the FCC rules. The most noticeable part of the shielding is a foil panel, resting between the main printed circuit board and the black plastic at the bottom of the case.

The other FCC certification, number AWQ9SB-70372-DT-R, pertains to the physical and electrical qualities of the circuitry shown in figure 8.2. The standard used is referred to in part 68 of the FCC rules, which requires that the computer must not interfere with the ability of other phone customers to place their calls and must not generate any voltages that might injure telephone workers.

Neither of these FCC certifications establishes that the computer does a proper job of dialing the phone nor does it even indicate that the computer will function when it is turned on. (An empty box would also satisfy both FCC requirements and would in fact be easier to get certified.)

MODEM DATA FLOW

The incoming modem signal passes through a filter composed of six operational amplifiers, shown in figure 8.3. The filters remove almost everything except the energy in the neighborhood of the frequency of the incoming carrier. In the originate mode, this is 2025-2225 Hertz; in answer mode this is 1070-1270 Hertz. (Transistors T2, T3, and T5 affect the frequency change in the filter.) The resulting signal, designated RXCAR, varies between 0 and 5 volts and wiggles up and down at the same frequency as the received carrier. It goes to the modem chip and appears as a "1" (about half of the time) at bit 0 of input port D8. The CPU can check to see if the carrier is being received by noting whether RXCAR keeps changing (carrier present) or remains constant always 0 or always 1 (carrier absent).

THE MODEM RECEIVER

The output of the modem filter goes to the modem integrated circuit, shown in figure 8.4. It accepts the signal from the modem filter, which ranges from 0 to 5 volts, and which varies in frequency. Based on the position of the ORIG/ANS switch, it interprets the signal to yield serial digital data. For example, in the originate mode, if the RXCAR signal is 2225 Hertz, the modem chip will send a logic "1" to the UART on the RXMi line.

The internal structure of the modem chip is shown in figure 8.5. The *type* input, at pin 14, can configure the chip for CCITT frequencies. For conversion to CCITT, however, the Model 100 would also require changes in the modem filter and wave shaper.

The modem chip includes a pin, TTLD, which reduces power consumption in the chip when, as in the Model 100, it is connected only to CMOS components.

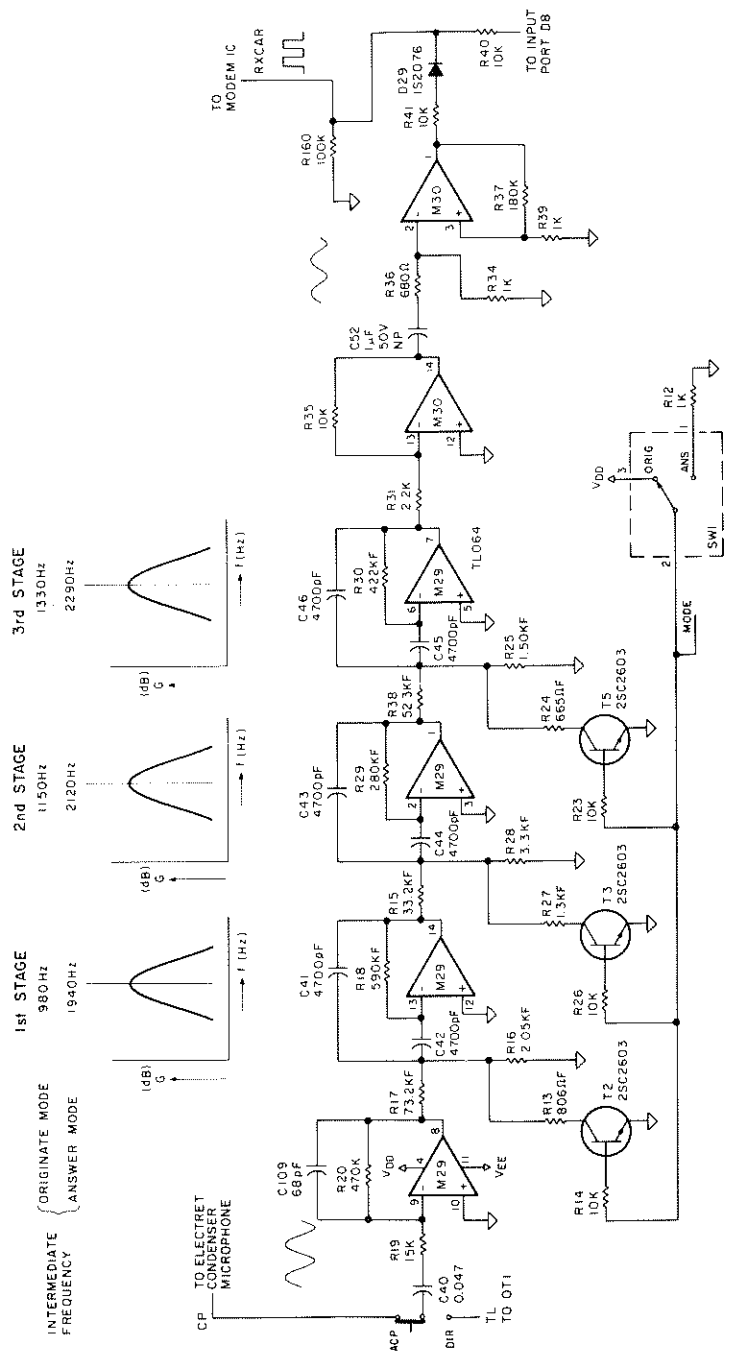

Figure 8.3. Modem incoming data filter

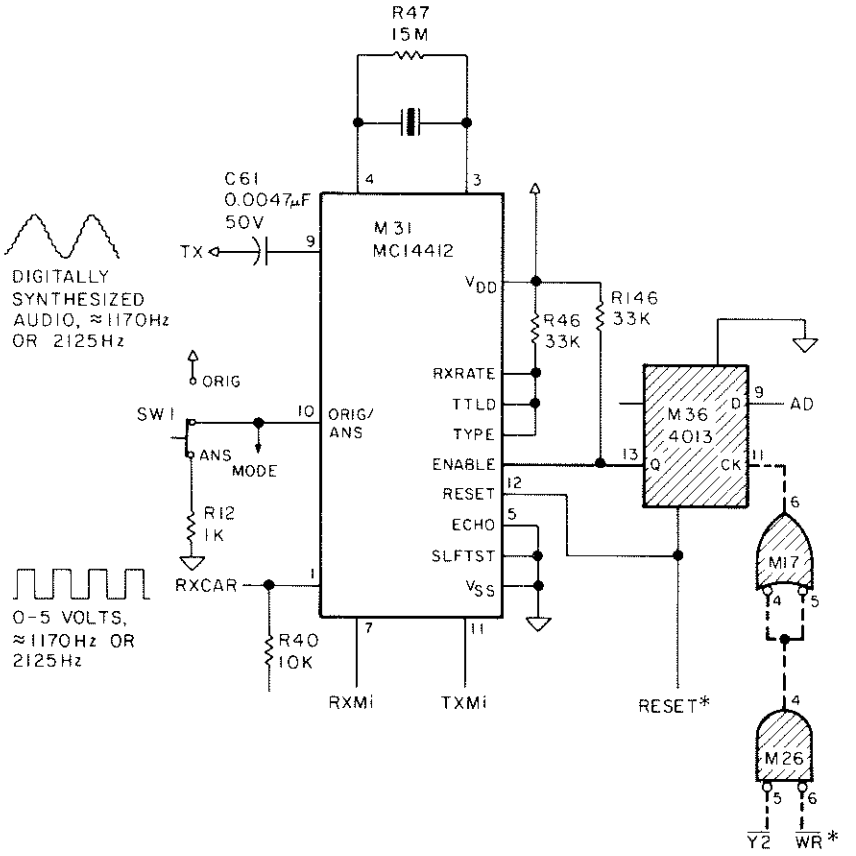

Figure 8.4. Modem IC connections

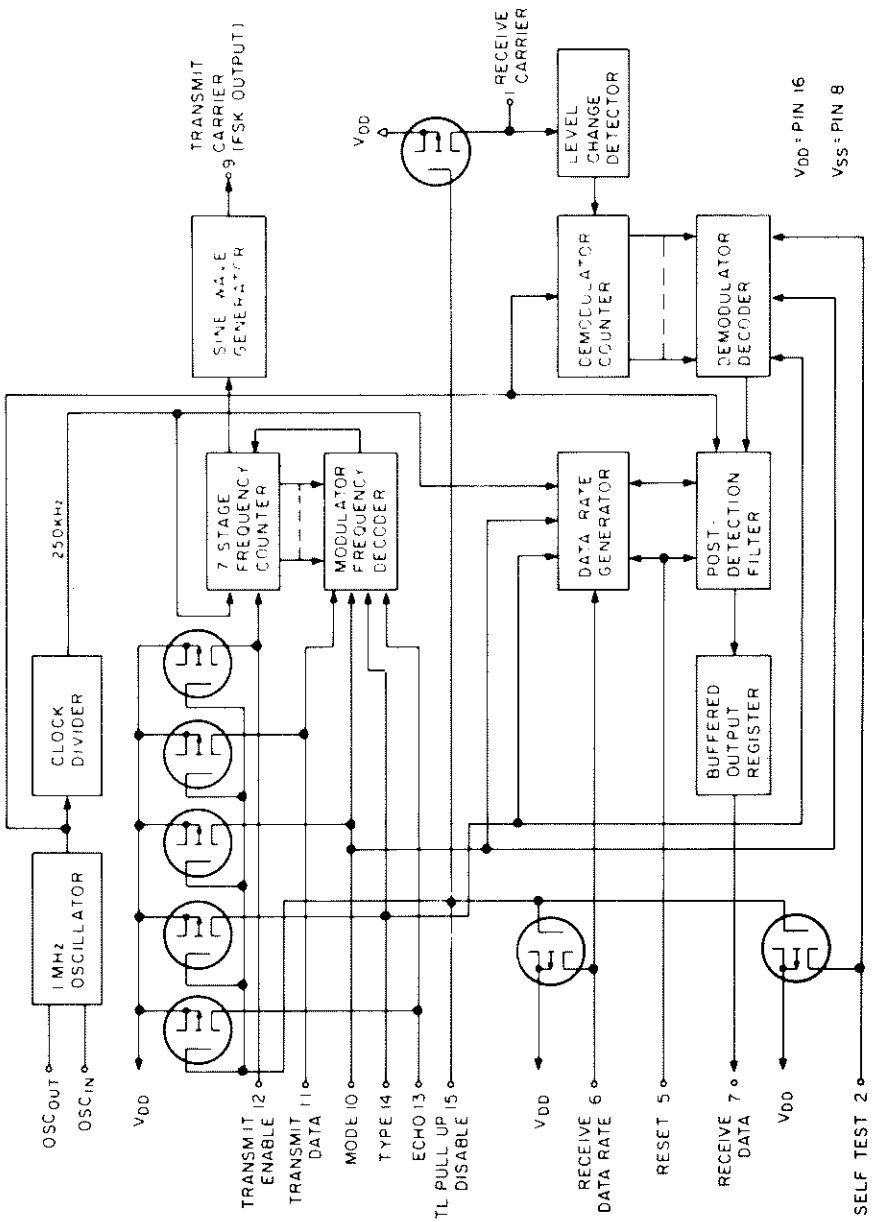

Figure 8.5. Modem IC functional block diagram

The resulting digital signal RXMi goes to the UART, where it is treated just the same as in the RS-232 mode discussed in the previous chapter.

Before modem output can be performed, the CPU must enable carrier output by means of a "1" at bit 1 of output port A8. (For most modems in the originate mode, the usual practice is to send out the originate carrier signal only after detecting the incoming carrier signal.)

OUTGOING DATA PATH

Outgoing modem data is sent by the CPU just as it was in the RS-232 mode. From the UART the serial digital signal TXMi goes to the modem chip, as shown in figure 8.4. The output, a synthesized audio signal TX, is fed to the wave shaper circuit shown in figure 8.6. There the audio volume level is set by potentiometer VR2 and is amplified and sent down the telephone line through OT1 or the coupler mouthpiece, depending on the position of the DIR/ACP switch.

ACOUSTICALLY-COUPLED MODEM

For acoustic-coupler operation, acoustic coupler 26-3805 is connected to the Model 100 instead of the direct-connect modem cable. When the DIR/ACP switch SW-2 is moved to the ACP position, three things happen.

First, assuming the Model 100 is in the RS-232C mode, bit 5 of input port BB will be found to be 1 rather than 0. It is this bit that allows the CPU to know that the DIR/ACP switch has been moved to the ACP position, although the Model 100 ROM routines never put this information to use.

Second, the computer "talks" to the coupler rather than to the direct-connect cable. The modem audio output signal TX is removed from the direct-connect matching transformer OT1 and goes instead to pin 5 of the phone jack, and from there to the acoustically-coupled speaker that clamps to the mouthpiece of the telephone handset. The modem cable connects to the mouthpiece cup by means of a 3/32" plug (similar to Radio Shack cat. no. 274-289) and jack. This is shown in figure 8.7.

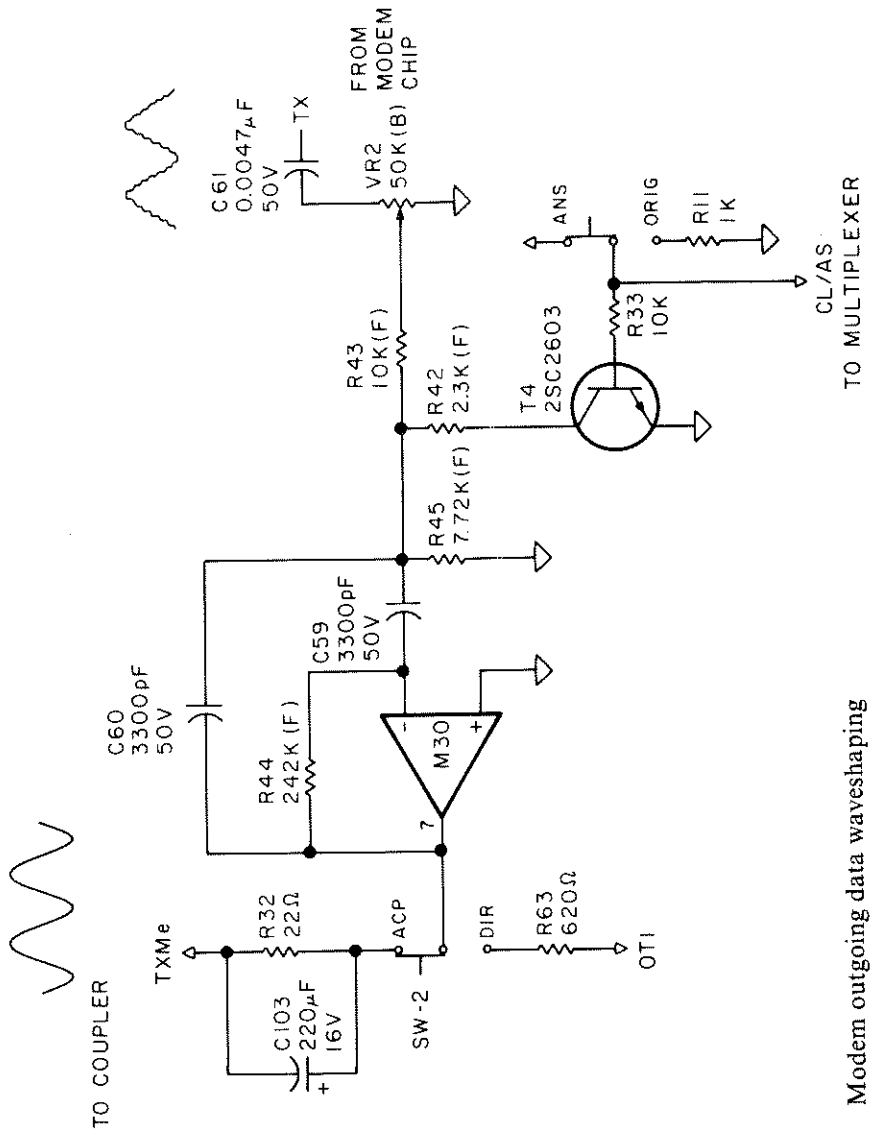

Figure 8.6. Modem outgoing data waveshaping

Finally, the computer "listens" to the coupler rather than to the direct-connect cable. The modem audio filter receives its input from the earpiece of the handset via an acoustically-coupled electret-condenser microphone, shown in figure 8.7, and not from OT1. The microphone signal is amplified by an operational amplifier (located in the earpiece cup) which draws upon the 5 Volt power available at phone jack pin 6. Capacitor C3 filters ripples from the 5 volt supply, while C2 removes high frequency pickup. Resistor R9 provides DC power to the microphone itself, while C4 capacitively couples the audio signal to the op amp.

The amplifier audio output is provided for the Model 100 at phone jack pin 4. Both the speaker and microphone are grounded at pin 2.

The acoustic coupler plug uses pins 2, 4, 5 and 6. By comparison, the direct-connect modem cable uses pins 1, 3 and 7. In each case, the cable requires an uncommon 8-pin DIN plug which differs from a standard 5-pin DIN plug such as Radio Shack carries (*see* catalog number 274-003).

USE OF THE COUPLER

When the acoustic coupler is plugged into the Model 100, the line-control relays RY2 and RY3 are not connected to anything. Thus the autodialing features of TELCOM cannot be put to use, and dialing a computer access number must be performed manually. It would have been more efficient if TELCOM had been written to sense the position of the DIR/ACP switch, so that when ACP was selected, the computer would skip the autodialing process and go directly to the login sequence. Instead in the ACP mode, TELCOM still goes through the futile routine of dialing the phone number.

DIALING PROCEDURES WITH THE COUPLER

When using the coupler, one dials the phone number and then connects the acoustic cups. When the carrier tone is audible, one choice is to push the "Term" key of TELCOM, which establishes the connection of the Model 100's modem circuitry to the cups. However, it would be nice to take advantage of TELCOM's ability to send the login

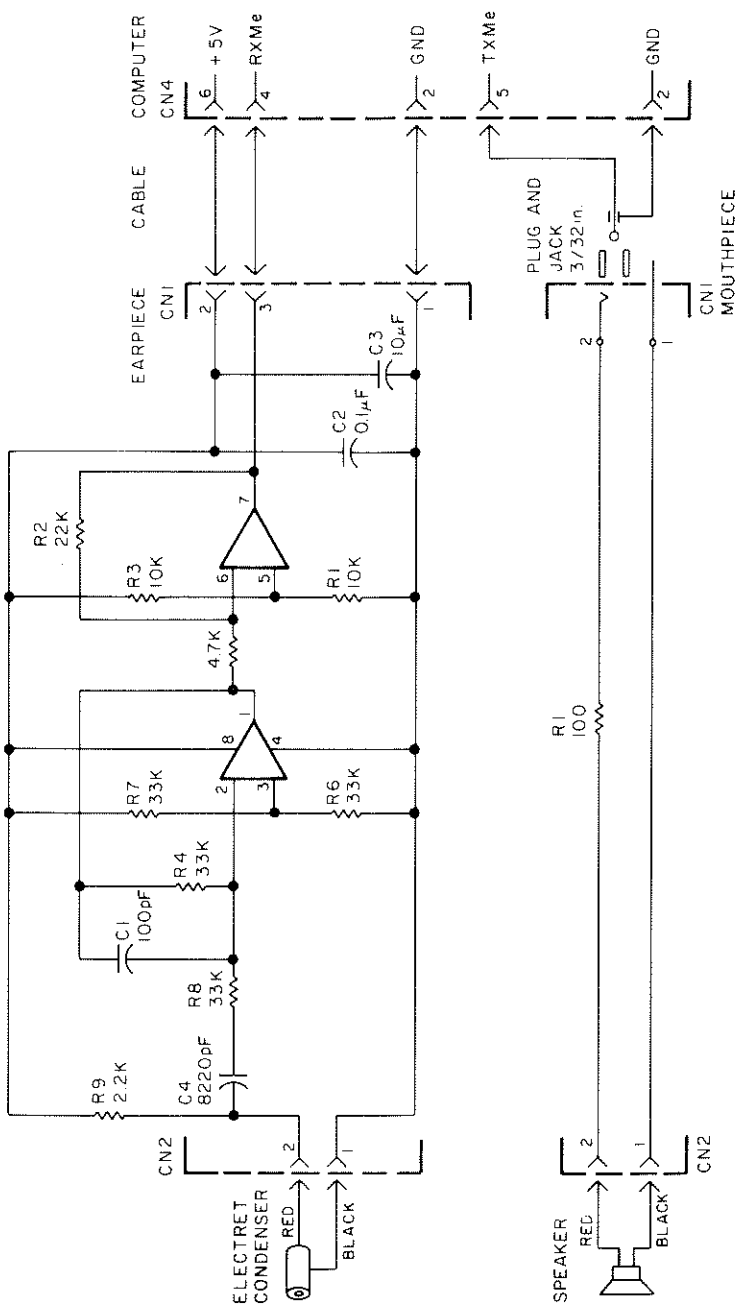

Figure 8.7. Acoustically-coupled Modem

sequence. This can be accomplished by using TELCOM's "Call" feature even though the dialer does not function. When TELCOM detects the carrier signal, it goes to the login sequence contained in the angle-brackets "<>" just as it would if the direct-connect cable had been attached.

I/O PORTS

Most of the modem functions are in ports shared with other functions. One port, A8, is used exclusively for modem functions. *See* table 8.1. RAM location FAAE contains the present contents of the port, to facilitate changing one bit without affecting the other. Other modem I/O functions are listed in table 8.2.

Table 8.1. Telephone relay/modem control (output port A8; contents at FAAE)

Bit	Function
0	Telephone instrument relay (1=disconnect)
1	modem transmit (1=enable)
2-7	not used

Table 8.2. Modem I/O port functions

Port	Bit	Function
out BA	7	phone line off-hook
in BB	4	1=ANS, 0=ORIG
in BB	5	1=ACP, 0=DIR
in D8	0	Carrier Detect
in D8	5	Ring Pulse

ROM SUBROUTINES

Four subroutine addresses have been published for modem operations. Two, CARDET and DIAL, are of substantial value, while the other two are simple implementations of the I/O port functions discussed earlier.

The routine CARDET, called at 6EEF, returns with the Z flag set and A=00 if a carrier is detected. It returns with Z reset and A=FF if no

carrier is detected. CARDET, which lies in ROM at 6ED6-6F30, and uses some rather tricky techniques. At 6EF2, for example, the value 6F2C is pushed to the stack, and if the search for carrier is unsuccessful, a RET instruction results in a jump to that address, which is halfway through another opcode. (Normally, every PUSH has an associated POP that is always executed.)

6EE5-6EED contains code which toggles the beeper during the carrier search if SOUND is ON (i.e. (FF44) is zero).

Other published routines are as follows:

DISC Called at 52BB, disables transmission of carrier, reconnects the phone instrument, and puts the phone line back on-hook.

CONN Called at 52D0, takes the phone line off-hook, disconnects the phone instrument, and enables carrier transmission.

DIAL Called at 532D, dials a phone number and follows a login sequence, just as does the "Call" button in TELCOM. Before the call, HL must point to the phone number sequence.

If the sequence has a CTRL-Z, CR or LF before an angle bracket ("<") the routine finishes by connecting the phone instrument; thus DIAL may be used as an autodialer.

Upon return from the routine, if the carry flag is set, the routine was unsuccessful, probably because the SHIFT-BREAK key was pushed.

The dialing rate, 10 or 20 pps, is a function of the "pps" flag at F62B.

9

Piezoelectric Beeper

Located directly under the TRS-80 top panel logo is a piezoelectric beeper. It provides an audio monitor of cassette data input and TELCOM dialing progress and performs the BASIC commands SOUND and BEEP.

How Piezo Beepers Work

Since the 19th century it has been known that pressure applied to certain crystals generates electricity. This piezoelectric effect is named after the Greek word "piezein" meaning to press. Years ago the effect was used in crystal microphones and crystal phonograph cartridges; one common present-day consumer application is the flintless butane lighter, in which mechanical energy from the user's thumb is converted to a voltage high enough to create a spark to light the vaporized fuel.

A lesser-known aspect of the piezoelectric effect is the fact that application of electrical potential to such a crystal causes physical deformation, such as expansion, contraction, or twisting, depending on the shape of the crystal and the location at which the potential is applied.

The most common consumer applications of the electrical-to-mechanical principle are the little earphones provided with inexpensive transistor radios and the high-pitched chirpers used in smoke detectors and one-piece telephones.

It is this latter aspect of the piezoelectric effect that is used in piezo beepers. When stimulated by a varying voltage, a quartz crystal deforms, moving a metal disc to which it is attached. For example, a square wave electrical signal applied to the disc produces something approaching a square wave audio signal in the air, because of the movement of the disc.

The conversion of electrical energy to mechanical energy in a piezo beeper is quite efficient — far more efficient than the energy conversion in a permanent-magnet audio speaker. This efficiency helps to conserve the battery power of the computer.

Some piezo beepers contain driving circuitry so that they can operate from direct current, yielding a fixed-frequency audio tone. In its simplest form a piezo beeper is composed of little more than the crystal and the metal disc; the beeper installed in the Model 100 is of this type. Figure 9.1 shows the beeper mounted on the inside top panel of the computer, and figure 9.2 shows the beeper fully disassembled. Wires are connected to two surfaces of the crystal.

Because the Model 100 beeper has no built-in driver circuitry, the CPU must ensure that appropriate signals are sent to it. One result is that the CPU can cause the beeper to produce a wide variety of audio signals.

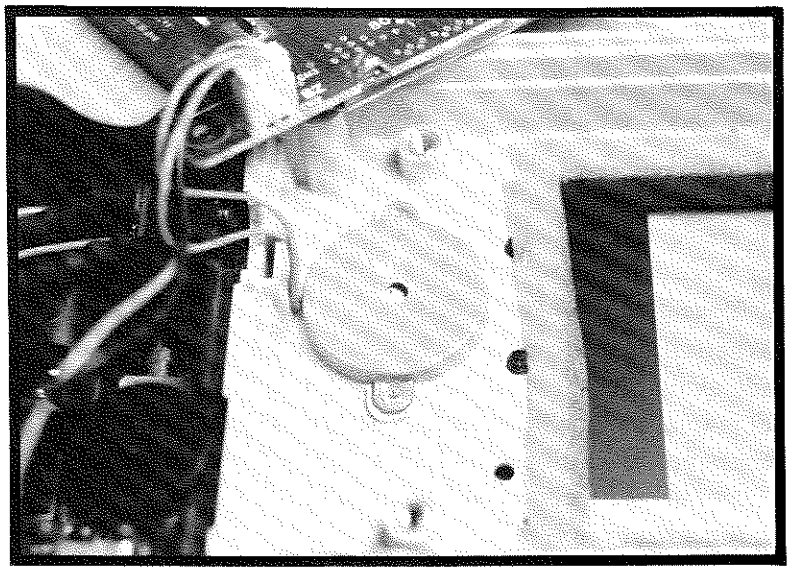

Figure 9.1. Beeper position inside the computer

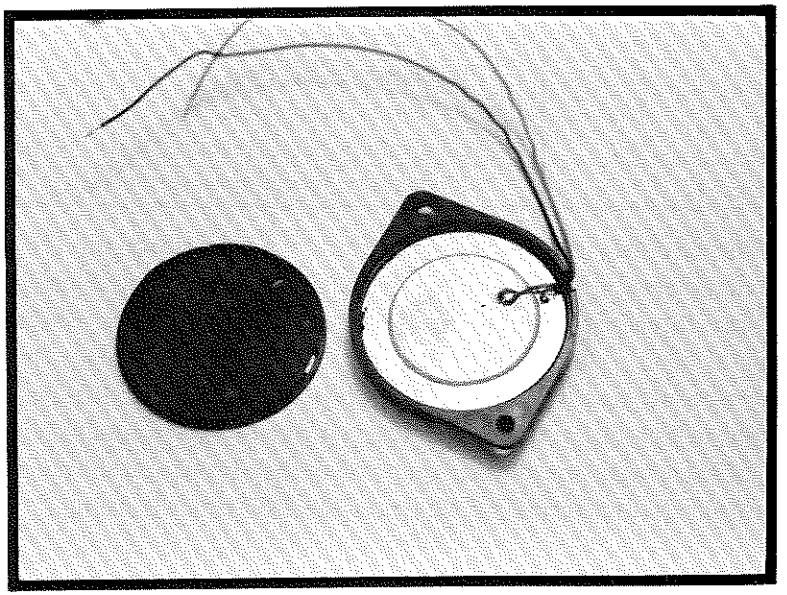

Figure 9.2. Beeper fully disassembled

Hardware Theory of Operation

The beeper module, mounted on the underside of the top panel, bears reference designation P-26 and plugs into connector CN2, reference designation A-8, on the LCD printed circuit board. This is shown in figure 9.3. A flat cable runs from the LCD board to the main printed circuit board, where it is controlled by transistor T1. One side of the beeper is connected to +5 volts, while the other side goes to the transistor and a pull-down resistor.

When the transistor emitter-collector junction conducts, +5 volts is made available to the BZ line; the voltage difference seen by the beeper is small. When the transistor turns off, however, the BZ line is effectively grounded through R113; the beeper sees about 5 volts.

Transistor T1 is controlled by bits 2 and 5 of CPU output port BA or B2; (decimal 178 or 186). Table 9.1 lists the various port addresses associated with the beeper.

Table 9.1. Port addresses associated with beeper.

Port	Bit	Function
BA	2	Disconnects beeper from divider
BA	5	Direct beeper control line
BC	0-7	PIO divisor- lower byte
BD	0-5	PIO divisor- upper byte
BD	6-7	PIO divider mode (01=square wave, 11=pulses)
B8	7	PIO divider control (01=stop, 11=start)

The circuitry associated with bits 2 and 5 allows the CPU to control the beeper in either of two ways. With bit 2 on continually, the CPU can directly toggle the transistor and through it the beeper by means of bit 5. Alternatively, with bit 2 low and bit 5 high, the TO (timer output) signal from the PIO controls the beeper. Bit 2 can be thought of as a timer mask. Because it is one of two inputs to NAND gate (M26), a logic one at that input causes the output and the beeper to be completely unaffected by whatever the PIO timer is doing.

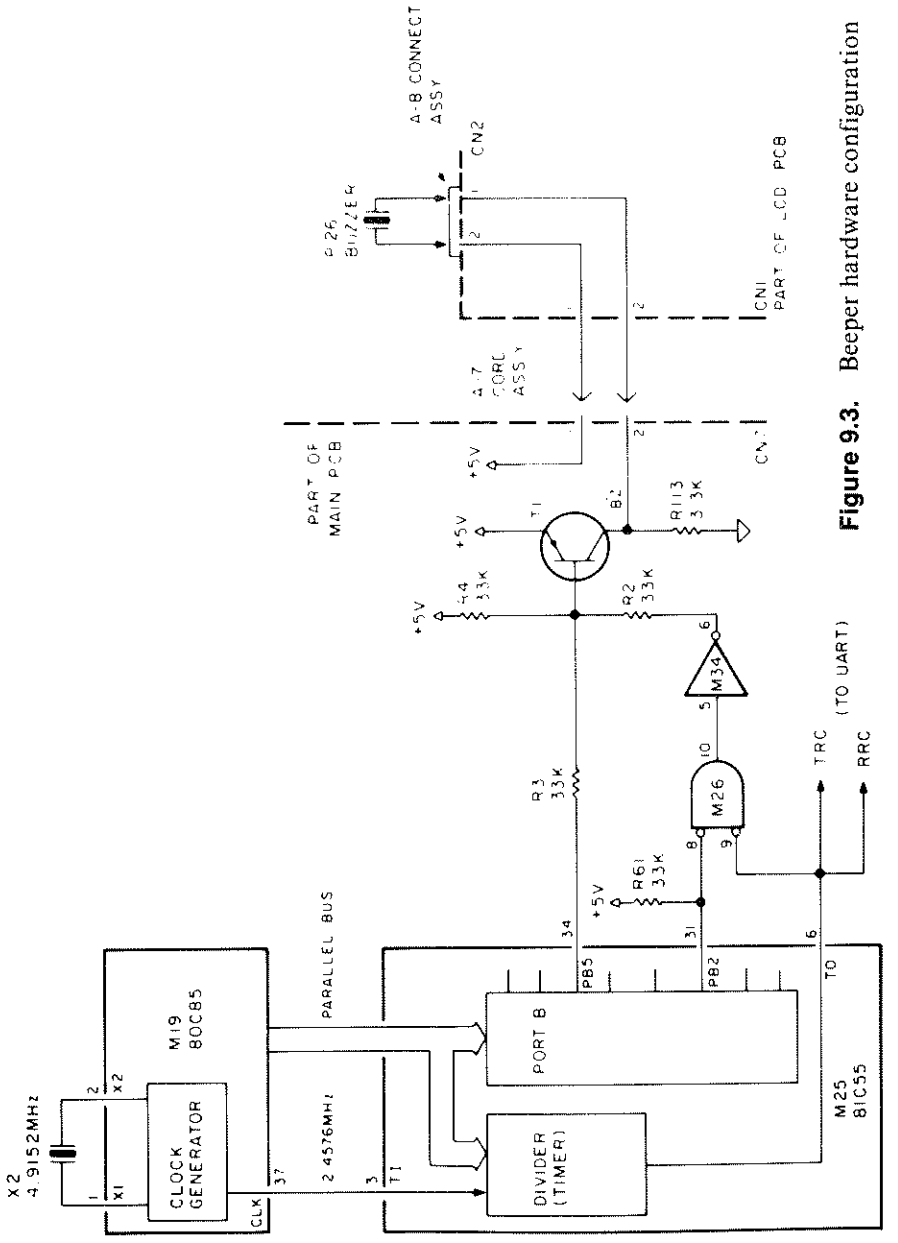

Figure 9.3. Beeper hardware configuration

CPU output port BA, however, controls many functions other than the beeper. For example, setting bit 4 cuts off all electric power to the computer. Because of this, it is important not to inadvertently change any bits other than the beeper bits.

Two aspects of the PIO hardware make control of individual bits very easy. First, the present logic levels of the output port are always available to the CPU simply by reading input port BA. Second, the design of the PIO port is glitch-free. Any bits that are logically unchanged as a result of loading a new port value, will remain electrically unchanged, throughout the loading of the new data.

CPU Toggling

Consider the simple ROM code at 7676-767C, shown in figure 9.4, which toggles bit 5. If bit 5 is on, it is turned off and vice versa.

```
7676    DB BA    IN BA     ;IN A,(BA)
7678    EE       XRI 20    ;XOR 20
767A    D3 BA    OUT BA    ;OUT (BA),A
767C    C9       RET       ;RET
```

Figure 9.4. ROM beeper-toggling routine

The other bits of the output port are completely unaffected. Try calling this routine from BASIC. In the immediate mode, type CALL 30326. You should hear a faint click from the beeper. Now enter and run the following BASIC program:

```
1 CALL 30326 : GOTO 1
```

You should hear a low buzz. The time interval between clicks is determined by the amount of time BASIC takes to accomplish the GOTO and parse the CALL command. If the equivalent routine were executed in machine language, the pitch would be very high, since assembly language is so fast in relation to BASIC.

One way this subroutine is used in ROM is by generating the BASIC command BEEP, which you can call from a machine language program with CALL 7662. Or, in BASIC, type CALL 30306.

Disassemble the code from 7657 to 767C. You will see the equivalent of nested FOR loops — an outer loop which determines how long the beep will last (about 117 milliseconds), an inner loop which determines how much time passes between togglings of the transistor, and the frequency, which is about 1 kilohertz. (The term "Hertz" is synonymous with "cycles per second".)

Let's see how these times may be calculated. The majority of the time consumed in the inner loop is in this subroutine:

```
7657    0D       DCR C     ;DEC C
7658    C2 57 76 JNZ 7657  ;JP NZ,7657
```

Register decrements require four clock cycles. Conditional jumps take seven cycles if the condition fails or ten cycles if the jump actually occurs. In this case, since C has been loaded with 50 hex (80 decimal), many decrements and jumps occur, totaling a time interval expressed as:

```
80 * (4+10 cycles)/(2.4576 MHz)
```

This evaluates to about 0.456 milliseconds. (In this expression "cycles" refers to CPU clock cycles.)

For the sound wave emitted by the beeper to complete one cycle (one period), two togglings must occur. The audio frequency is the inverse of the period, or about:

```
(1 cycle)/(2 * 0.456 milliseconds)=1100 cycles per second.
```

In this expression, "cycles" refers to the audio signal produced by the beeper.

The duration of the beep is determined by the outer loop. The B register is loaded with zero and decremented once for each toggle until it once again equals zero. This means it is decremented 256 times. The duration is:

```
(256 toggles)*(0.456 Msec/toggle)=0.117 sec.
```

The actual BEEP frequency is somewhat lower than the calculated value. The duration is longer, because the length of the toggling period is longer than the 0.456 seconds calculated above. The subroutine at 7657 is, after all, being called by a higher routine with instructions of its own. It takes time even to accomplish the toggling.

For this application, approximate values work fine. Chapter 12 describes some activites, such as reading and writing a high-density magnetic tape, that require a careful counting of every machine cycle.

Toggling the beeper transistor is also used to monitor the cassette loading process and the Telcom carrier-detection process. Consider, for example, the code at 700D through 7011, deep in the heart of the cassette input routine:

```
LDA     FF44    ;LD A,(FF44)
ANA     A       ;AND A
CZ      7676    ;CALL Z,7676
```

This code is reached whenever the cassette input circuitry detects a properly timed plus-to-minus or minus-to-plus transition in the incoming cassette audio data. It first inspects the contents of FF44, the location of the SOUND ON/OFF flag. Assuming SOUND is ON, the toggling routine is called.

From this you can see how SOUND ON and SOUND OFF can be accomplished in assembly language. SOUND ON is the same as loading zero to FF44, and SOUND OFF is the same as loading a nonzero value.

The audio waveform given off by the beeper is a fair copy of the waveform provided to the computer by the cassette.

A similar routine is used in TELCOM at 6EEA. The beeper is toggled in response to changes in the carrier detect bit (bit 0), of input port D8. The carrier detection filter, discussed in detail in the previous chapter, does allow noises other than a bona fide carrier tone to pass through. These noises show up in the carrier detect bit, and allow you to hear such things as a ringing phone number when you place a call to a distant modem.

It is possible to use the beeper to synthesize, albeit crudely, the human voice. Try writing a program that repeatedly samples a recorded voice played to the cassette input signal, storing in RAM the "1" or "0" that is found each time. The "1"s and "0"s are loaded to the beeper at intervals equal to the sampling intervals.

The playback interval can be varied to change the pitch of the reconstructed voice.

A fundamental rule of digital synthesis is that the number of samples per second must be more than double the desired bandwidth to be reproduced. Since the Model 100 cannot reproduce the waveform, but only the zero crossings, intelligible synthesis requires a far faster sampling rate. Synthesis of a one-second phrase might require five or eight thousand samplings.

Eight thousand samples, of course, need not fill up eight thousand bytes. Since each sample is a single binary digit, rotate instructions can be used to pack them into just 1K.

PIO Timer Use

The PIO chip, as discussed in chapter 7, contains a divider (sometimes called a timer), which is used during UART input and output to generate the transmit and receive baud rates. The CPU crystal frequency of 4.9152 megahertz is halved by the CPU and provided for the TI (timer input) pin of the PIO. There, depending on divisor and mode data loaded on the PIO, a lower frequency can be produced at the TO (timer output).

When the UART is not in use, the divider can be connected to the beeper and loaded with a divisor to produce a desired audio frequency.

To do this in assembly language, select a divisor based on the desired frequency:

```
divisor=(2.4576 MHz)/(desired frequency).
```

For example, to produce a concert A (=440 hertz), the divisor should be about 5585. The low-order part (5585 AND 255, which is 209) belongs in output port BC, the least-significant byte of the PIO divisor. The high-order part (5585-209)/256, which is 21, belongs in output port BD, the most significant byte of the PIO divisor.

In addition, the timer mode must be set as a "square wave", which requires that the the word sent to port BD has bit 6 on and bit 7 off. The value sent to BD is 21+64, or 85.

Next, a command word is sent to the PIO telling it to start the divider running. As described in chapter 5, this is accomplished by sending a C3 hex to output port B8. *See* table 9.1, listing the beeper output ports.

Finally, the beeper must be connected to the divider. Output port B of the PIO (CPU output port BA) needs to have bit of 2 off and bit 5 on. This can be done by reading the value of input port BA, ANDing it with FB hex (which turns off bit 2), ORing it with 20 hex (which turns on bit 5), and sending that value to output port BA. After doing all this, the beeper should sound.

A ROM subroutine, MUSIC, is available to make the beeper sound, and it is the same routine as that used by the BASIC SOUND command. The pitch is determined by the divisor in the double register DE, and the duration is determined by the byte in B; it is invoked by a CALL to 72C5. Disassemble the code at 72C5 through 7303, and use the following comments to understand it:

72C6-72C7	Send low-order byte of divisor
72C9-72CC	Send high-order byte with mode bit
72CE-72D0	Turn on divider
72D2-72D8	Connect beeper to divider
72DA-72F6	Let the tone continue, but respond to BREAK key
72F9-72FF	Disconnect beeper, resume previous beeper activity

Musical Tones

As mentioned in the Model 100 user's manual, the BASIC SOUND command can be used to make musical tones. (Of course, from assembly language, the same routine is used by CALLing 72C5.) Unfortunately, the divisor values given there are incorrect. Figure 9.5 is a BASIC program that calculates the correct values.

The method used is simple. A concert A pitch is assumed to be 440 Hertz, although other frequencies have been used. The program could easily be modified to some other "A" frequency. For a note of any given

frequency, the note one octave above it is defined simply as the note whose frequency is double the given frequency.

The definition of the octave, by itself, does not suffice to determine the frequencies of the notes constituting the scale in between. For the last century, though, Western musicians have used a so-called equal-tempered scale. Each pair of notes, going up the scale, has the same ratio of frequencies. From this, it follows that the ratio must be the twelfth root of two, so that a change of twelve steps doubles the frequency.

Knowing this, it is easy to write a program in BASIC which calculates the frequencies and appropriate divisors.

```
5 DIM N$(11):FOR I=0 TO 11:READ N$(I):NEXT
6 DATA"A","A#","B","C","C#","D","D#","E","F","F#","G","G#"
10 FOR I=-18 TO 26
20 F=440*(2^(1/12))^I
30 D=(2.4576*10^6)/F
40 PRINT USING"      ##### #####";N$((I+12*100)MOD12),F,D
45 SOUND D,50
50 NEXT
```

Figure 9.5. Calculation of divisors for notes of even-tempered scale.

The resulting tones are shown in table 9.2.

Table 9.2. Divisors for even-tempered tones

Note	Frequency	Divisor
D#	156	15798
E	165	14911
F	175	14074
F#	185	13285
G	196	12539
G#	208	11835
A	220	11171
A#	233	10544
B	247	9952
C	262	9394
C#	277	8866
D	294	8369
D#	311	7899
E	330	7456
F	349	7037
F#	370	6642
G	392	6269
G#	415	5918
A	440	5585
A#	466	5272
B	494	4976
C	523	4697
C#	554	4433
D	587	4184
D#	622	3950
E	659	3728
F	698	3519
F#	740	3321
G	784	3135
G#	831	2959
A	880	2793
A#	932	2636
B	988	2488
C	1047	2348
C#	1109	2217
D	1175	2092
D#	1245	1975
E	1319	1864
F	1397	1759
F#	1480	1661
G	1568	1567
G#	1661	1479
A	1760	1396
A#	1865	1318
B	1976	1244

10

The Printer Interface

The Model 100 communicates with a printer according to the Centronics interface standard, which defines mechanical, electrical, and software characteristics of the interface.

Mechanical Requirements

The first requirement of the Centronics standard is the connector, a 36-pin device usually made by AMP or Amphenol. At the rear of the Model 100 is a 26-pin connector labeled "PRINTER", with the hardware designation CN5. This connector, with square pins spaced 1/10 inch apart, was probably chosen to save precious space on the Model 100 case. The printer cable 26-1409, the connections of which are shown in table 10.1, plugs into CN5 and has a connector at the other end that conforms to the Centronics standard.