6

Hidden Powers of the Keyboard

Concepts

- How a scanning keyboard works
- How to program the keyboard
- Descriptions of ROM routines
- The clock-cursor-keyboard background task
- Keyboard input routines

The keyboard is the main input device for the Model 100. The hardware for the keyboard is quite simple, consisting of an array of switches that correspond to the keys of the keyboard. However, the software that manages the keyboard is quite complex.

In this chapter we'll explore the secrets of the Model 100's keyboard. We will see in general how such a keyboard works; then we will study the specific details of the Model 100's keyboard. We will describe the section of the background task in the Model 100's ROM that runs the keyboard, allowing the computer to continually capture keystrokes as it goes about its other business. We will see how the characters are stored in a buffer as they are typed, and we'll see how other routines pick them up as needed. We will also describe the ON KEY BASIC interrupt routines that allow you to program the Model 100 in an interactive, user-friendly manner.

How a Scanning Keyboard Works

The Model 100 uses a "scanning keyboard" run by the CPU. This kind of keyboard requires that the CPU constantly send pulses into the keyboard through certain signal lines while "sampling" certain other lines coming out of the keyboard. This "pulsing and sampling" must happen quite frequently because the keyboard hardware cannot "remember" what key or keys you hit once you have released them. On the Model 100 the entire keyboard is scanned every 12 milliseconds. That's about 84 times a second.

From the scanning information the CPU can tell exactly what key or key combinations are being depressed. We will see later how the computer can develop a complete "image" of the keyboard in its memory as it scans the keyboard.

Logically, the keys are arranged in the form of a two-dimensional matrix (see Figure 6-1). Each key operates a switch that connects one input line to one output line. The input lines run through the keyboard forming the columns of this matrix, and the output lines form the rows of the matrix. Each key lies at an intersection and thus is uniquely identified by a choice of one input line and one output line. You should be aware that the *logical* wiring of this matrix does not conform exactly to the *physical* arrangement of the keys.

Let's look at how the sampling process works. We'll start with the job of determining if a single specified key is depressed, and then we will see how to scan for *any* key. To look for a particular key, the first step is to turn "on" the input line for its column, turn "off" the input lines for all the other columns, and then examine the output lines. Under this condition, the output lines tell you the state of the keys of just this one column. If a particular output line is "on", then its key is being depressed; otherwise, it's not.

To find out if any key at all has been depressed, you must check all the columns, one at a time, using the above method. The resulting sets of "on/off" patterns of the columns can be arranged side by side to give a "picture" of the keyboard showing exactly which keys were depressed.

How to Program the Model 100 Keyboard

In this section we will develop a BASIC program to display the keyboard matrix on the liquid crystal display.

The Model 100 uses a nine column by eight row matrix for its keyboard (see Figure 6-1). Of the seventy-two possible positions in this matrix, seventy-one correspond to keys on the keyboard, leaving one position that does not correspond to any key.

The nine input lines are connected to eight bits of port B9h = 185d and one bit of port BAh = 186d (bit 0). A bit value of 1 turns off an input line and a bit value of 0 turns it on. This is just the opposite of what you might

expect, but the designers of the Model 100 placed "inverters" in the keyboard electronics that perform this reverse in logic.

The input ports are also used by the LCD and the clock. However, there is no conflict under normal operations because the keyboard output lines are never read when ports B9h = 185d and BAh = 186d are being used for other purposes.

The eight output lines are connected to port E8h = 232d for input to the CPU. This input port is not shared by any other part of the system, so there is never any confusion here.

For the output lines, a bit value of 1 means the corresponding switch was open (key not depressed), and a bit value of 0 means the switch was closed (key was depressed). Again, the logic is reversed relative to what you might expect, but it is consistent with the logic for the input.

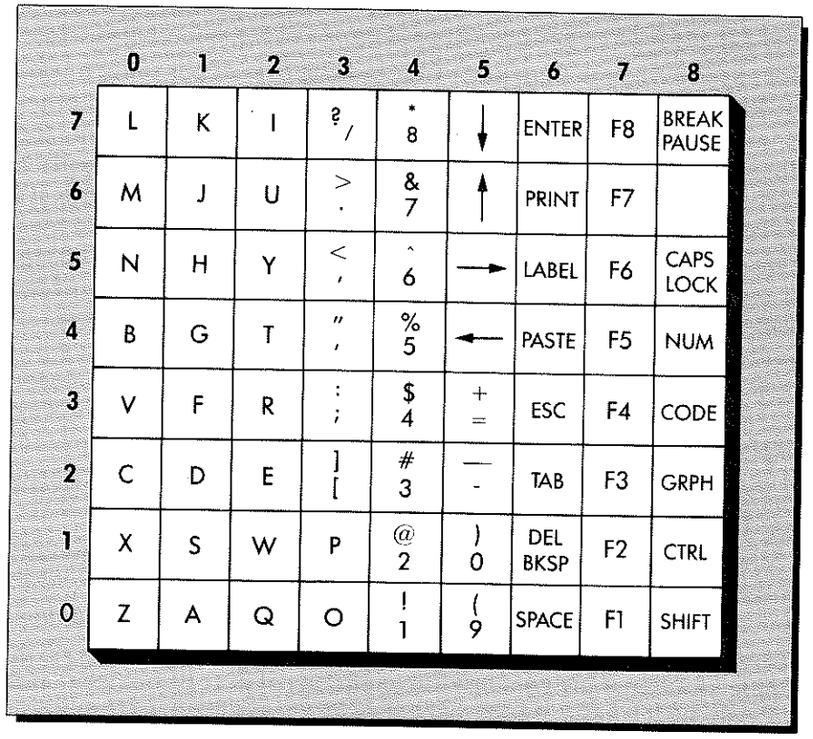

Figure 6-1. The Model 100 keyboard matrix

The following program interactively displays the keyboard matrix for the Model 100. When you run this program, you will see a small rectangular display of pixels in the middle of your screen. If you now press various combinations of keys, you will see the corresponding pixels change from dark to light. To stop the program, just hold down CTRL C or BREAK for a moment.

```
100 ' DISPLAY KEYBOARD MATRIX
110 '
120   CLS
130   PRINT TAB(10);"KEYBOARD MATRIX"
140 '
150 ' MAIN LOOP
160 '
170 ' TURN OFF BACKGROUND TASK
180   CALL 30300
190 '
200 ' SET BIT 0 OF PORT BAh
210   X=INP(186)
220   OUT 186,X OR 1
230 '
240 ' SCAN THROUGH 8 COLUMNS
250   OUT 185,254:A0=INP(232)
260   OUT 185,253:A1=INP(232)
270   OUT 185,251:A2=INP(232)
280   OUT 185,247:A3=INP(232)
290   OUT 185,239:A4=INP(232)
300   OUT 185,223:A5=INP(232)
310   OUT 185,191:A6=INP(232)
320   OUT 185,127:A7=INP(232)
330 '
340 ' CHECK NINTH ROW
350 '
360 ' TURN OFF LOWER COLUMNS
370   OUT 185,255
380 '
390 ' TURN ON JUST THE NINTH
400   X=INP(186)
410   OUT 186,X AND 252
420   A8=INP(232)
430 '
440 ' DISPLAY THE MATRIX ON THE LCD
450 '
460 ' JUST ONE LCD DRIVER
470   OUT 185,128
480 '
490 ' SET UP LCD POSITION
```

```
500   OUT 254,0
510  '
520  ' SEND THE BYTES TO LCD
530   OUT 255,A8
540   OUT 255,A7
550   OUT 255,A6
560   OUT 255,A5
570   OUT 255,A4
580   OUT 255,A3
590   OUT 255,A2
600   OUT 255,A1
610   OUT 255,A0
620  '
630  ' ALLOW FOR A BREAK
640   PRINT CHR$(11);
650  '
660  ' GO BACK FOR MORE
670   GOTO 150
```

Let's look at the code in detail. Except for some initialization consisting of clearing the screen (line 120) and printing the title "KEYBOARD MATRIX" (line 130), the program consists of a loop (lines 150-670) that continually reads the columns of keyboard matrix and sends them to the LCD.

At the beginning of the loop (lines 170-180) the background task is turned off by calling the machine-language routine at 765Ch = 30,300d (see Chapter 4 for description).

In the next part of the loop, the columns are turned on one by one, and the keyboard output port (port E8h = 232d) is read each time into one of the variables A0 through A7. The bytes are not sent directly to the display because the liquid crystal display also uses ports B9h = 185d and BAh = 186d.

Notice how the ninth column is handled. This column is not controlled by port B9h = 185d as are the other columns. Instead, it is controlled by bit 0 of port BAh = 186d. Before the other columns are handled, this bit is set to 1, turning off this column. This has to be done carefully because this bit is on a port (port BAh = 186d) that is shared by other devices, including the power for the machine. You can see how carefully we set this bit by looking at lines 210-220. On line 210 the port is read, and then on line 220 its contents are ORed with 1 as they are put back. It should be clear that this changes only the one bit.

Next, the first eight columns are scanned (lines 250-320). In each case, one bit of port B9h = 185d is set to zero while the others are set to one, and port E8h = 232d is read into the appropriate variable: A0 through A7.

After all the other columns have been scanned, it's time to scan the last column. Again, port BAh = 186d is read before it is written to. Notice that both bits 0 and 1 of port BAh = 186d are made 0 when the port is set. This has two effects: it turns on the last column of the keyboard matrix (with bit 0), and it also disables the last two LCD drivers (controlled by bits 0 and 1). These are used for subsequent display of the matrix. The byte from this column is then picked up and stored in the variable A8.

By line 420 all the raw data from the keyboard are stored in variables A0 through A8, ready to be sent to an LCD driver for display on the screen. But before this data is sent, all but one of the LCD drivers are disabled (line 470), and the starting byte address is sent to that driver through port FEh = 254d. The matrix is then displayed by sending this raw data directly to a LCD driver through port FFh = 255d (lines 530-610). (See Chapter 4 for detailed description of how to program the LCD.)

Before the bottom of the loop there is a command that homes the cursor. Its purpose is to turn on the background task so that a CTRL C can be sensed if the user wishes to terminate the program. Any time you use the PRINT command, the background task is turned on.

Besides illustrating how the keyboard works, this program demonstrates how the LCD can be programmed directly to display complex data in real time.

Descriptions of ROM Routines

The ROM routines for the keyboard fall into three main classes: BASIC interrupt, background scanning task, and keyboard input. With the information presented here you should be able to write your own BASIC or machine-language programs to detect and handle any kind of regular or special combinations of keys. You can set the keyboard up for all sorts of input configurations. As with the case of special programming for the LCD screen, this is especially useful for games and educational software.

The ON KEY Interrupt Routines

Let's start with the interrupt routines for the eight function keys. There are four such commands: ON KEY GOSUB, KEY ON, KEY OFF, and KEY STOP. These provide standard ways to have specially programmed keys.

The code for ON KEY GOSUB is located at A5Bh = 2651d (see box). This code is also shared by the ON TIME$ GOSUB, ON COM GOSUB, and ON MDM GOSUB commands. Here, a routine at 1AFCh = 6908d is called to determine which of these different device types is required. For

the ON KEY interrupt, this routine returns with a value of 0208h in the BC register pair, that is, a value of 2 in the B register and a value of 8 in the C register. This indicates where and how much information for this type of interrupt will be stored in the system's interrupt table, which is located at F944h = 63,812d (see Figure 6-2).

Routine: ON...INTERRUPT/GOSUB — BASIC Command

Purpose: To initialize BASIC interrupts

Entry Point: A5Bh = 2651d

Input: Upon entry, the HL register pair points to the end of an ON...GOSUB command line.

Output: When the routine returns, the location of the subroutine to handle the particular interrupt is set.

BASIC Example:

```
CALL 2651,0,H
```

where H is the address of the end of an ON...GOSUB command line.

Special Comments: None

For the ON KEY GOSUB command, the value 8 in the C register indicates that a maximum of eight locations (one for each function key) will be used, and the 2 in the B register indicates that they will start on the third location of this table (the first two are occupied by the TIME$ and the MDM or COM interrupts). (Note that the MDM and COM have exactly the same interrupt locations.) Each location in this table has three bytes, one for status and two for the location of the interrupt subroutine. The last part of the routine at A5Bh = 2651d reads the list of BASIC subroutines from the end of the ON KEY GOSUB command line and places their locations into the proper slots of the interrupt table (see Figure 6-2).

The KEY ON, KEY OFF, and KEY STOP use the same code and work in the same way as the TIME$ ON, TIME$ OFF, and TIME$ STOP commands described in Chapter 5. That is, they cause transitions of finite state machines (see Chapter 5) whose states are stored as status bytes in the interrupt table. Each function key has its own status byte and hence its own finite state machine.

The following program lets you "peek" at these status bytes, graphically demonstrating how these interrupt functions work. You can watch the state of the F4 interrupt as you turn it on, off, and stop it. The keys F1, F2, F3 have been assigned to do these tasks for the function key F4 so that you can have direct and convenient control over these functions for one particular interrupt. As you press these keys you will see the fourth status byte change. You should review the discussion of BASIC interrupts in Chapter 5 to see what the various values mean.

```
100 ' DISPLAY INTERRUPT STATES
110 '
120   CLS
130   PRINT "FUNCTION KEY STATES"
140   PRINT:PRINT:PRINT
150   PRINT "F1 = KEY(4) ON"
160   PRINT "F2 = KEY(4) OFF"
170   PRINT "F3 = KEY(4) STOP"
180 '
190 ' TURN ON INTERRUPTS
200   ON KEY GOSUB 320,350,380,410
210   KEY(1) ON:KEY(2) ON:KEY(3) ON
220 '
230 ' PRINT ON THIRD LINE
240   CALL 17020,,2
250 '
```

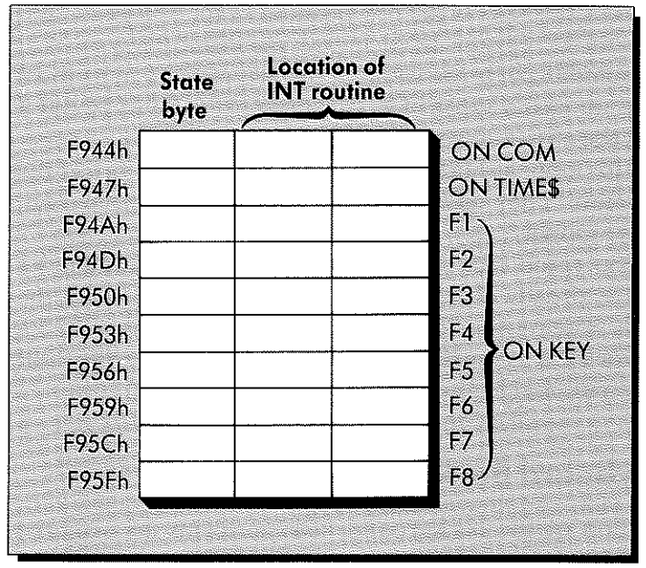

Figure 6-2. The system interrupt table

```
260 ' DISPLAY THE STATUS OF THE KEYS
270   FOR I = 63818! TO 63841! STEP 3
280   PRINT USING "####";PEEK(I);
290   NEXT I
300   GOTO 230
310 '
320 ' KEY 1 INTERRUPT ROUTINE
330   KEY(4) ON
340   RETURN
350 ' KEY 2 INTERRUPT ROUTINE
360   KEY(4) OFF
370   RETURN
380 ' KEY 3 INTERRUPT ROUTINE
390   KEY(4) STOP
400   RETURN
410 ' KEY 4 INTERRUPT ROUTINE
420   RETURN
```

Let's look at the code for this program. In lines 120-170 the background display is set up, showing a title and a description of the function keys. In lines 190-210 the locations of ON KEY interrupt routines are defined and the interrupts for F1, F2, and F3 are turned on.

Lines 230-300 form the main loop, which repeatedly displays the status bytes. At the top of this loop, the cursor is placed at the beginning of the third line by CALLing the cursor positioning routine at 427Ch = 17,020d (see Chapter 4). This routine expects the row position in the L register and the column position in the H register. In each case, the numbering starts at zero. The CALL command expects the value of the HL register in its third parameter. In this case we pass a value of 2, causing zero to be in H and 2 to be in L, thus setting the cursor to row 2, column 0.

In the last part of this loop, a short FOR loop (lines 270-290) prints the status bytes on the screen. They appear on the third line of the screen, where we just placed the cursor.

After the main loop come our BASIC interrupt subroutines for F1 through F4. As we have already observed, the first three keys control the interrupt for the fourth. The routine for F1 is a command to turn on the F4 interrupt, the routine for F2 is a command to turn off the F4 interrupt, and the routine for F3 is a command to stop the F4 interrupt.

The Clock-Cursor-Keyboard Background Task

The keyboard section of the background task performs an essential function for the Model 100 by constantly scanning the keyboard. It detects what keys are hit and puts their character codes into a keyboard character buffer.

The code for the keyboard section of the background task starts at 7055h = 28,757d and runs to about 7241h = 29,249d. It is divided into four main subsections: general management, key detection, key decoding, and character buffer management (see box).

Routine: Keyboard Scanning

Purpose: To scan the keyboard as part of the background task

Entry Point: 7055h = 28,757d

Input: Monitors the keyboard

Output: Puts codes for the keys in the keyboard input buffer

BASIC Example: Not applicable

Special Comments: Not a callable routine — part of background task

Management

The general management subsection controls program flow and timing (see box). It sets up a "graceful" exit from the keyboard section, and it times the keyboard scanning by allowing it to be performed only every third time that the background task is executed.

Routine: Management of Keyboard Scanning

Purpose: To set up timing and exit conditions for the keyboard scanning background task

Entry Point: 7055h = 28,757d

Input: A counter located at FF8Fh

Output: The counter is decremented if not already equal to one. If it is one, the counter is set to a value of three.

BASIC Example: Not applicable

Special Comments: Not a callable routine — part of the background task

The "graceful" exit from this section is accomplished by pushing the address 71F4h = 29,172d onto the stack. This address points to the very last part of the background task; hence any RETurn instruction in this section will automatically cause the CPU to jump there. The code at this "finishing" address of 71F4h = 29,172d has the correct series of "POPs" and RETurn to make a proper exit from the background task.

The waltz-like "every third time" timing is controlled by a countdown counter located at FF8Fh = 65,423d. This counter is loaded with 3 every time it reaches zero. Since keyboard scanning is performed only every third time that the background task is executed, this occurs about every 12 milliseconds (recall that the background task itself is performed about every 4 milliseconds.)

Key Detection

The key detection subsection must determine and record the matrix positions of keys that are pressed down on the keyboard. You can use the ideas described here to develop your own detection routines. For example, you could write a program that detects certain double and triple key combinations used with programs that simulate various instruments or machines.

The routine for key detection begins at 7060h = 28,768d (see box). It contains a short block of code starting at 7066h = 28,774d, which decodes the ninth column of the keyboard matrix, and then a loop (starting at 7080h = 28,800d) for detecting all the other keys. The loop is executed eight times, once for each remaining column of the keyboard matrix. The columns are scanned in reverse order, from last to first.

Routine: Key Detection

Purpose: To determine which keys have been hit

Entry Point: 7060h = 28,768d

Input: Monitors the keyboard

Output: Special tables in RAM contain information about the keyboard matrix.

BASIC Example: Not applicable

Special Comments: Not a callable routine — part of the background task

The job of key detection is complicated by the fact that the various shift keys (SHIFT, CTRL, GRPH, CODE, NUM, and CAPS LOCK) are used to modify the meaning of regular keys. For this reason and others, detecting the shift keys differs from detecting other keys.

The shift keys are all in the last (ninth) column of the keyboard matrix, which is controlled by bit 0 of port BAh = 186d. With the exception of the BREAK/PAUSE key, which is also on this column, all keys are controlled by the eight bits of port B9h = 185d. The ninth column is scanned first, in a separate section from the other columns.

The first few instructions of the keyboard detection routine set up pointers to two buffers, each nine bytes long. In each buffer there is one byte for each column of the keyboard matrix. The first buffer runs from FF91h = 65,425d to FF99h = 65,433d, and the second runs from FF9Ah = 65,434d to FFA2h = 65,442d (see Figure 6-3). The ending address of the first buffer is placed in HL, and the ending address of the second is placed in DE. They are placed this way because the columns of the keyboard matrix are scanned backward from last to first. This order is convenient because it allows the shift keys to be scanned first and it makes the loop counter (B register) equal to the number of the column.

Here is a short program that displays these buffers on the LCD screen. As you type characters, you will see the contents of the buffers change.

```
100 ' KEYBOARD MATRIX BUFFER DISPLAY
110 '
120   PRINT CHR$(11);
130   FOR I = 65425 TO 65433
140    PRINT PEEK(I);
150   NEXT I
160   PRINT
170   FOR I = 65434 TO 65442
180    PRINT PEEK(I);
190   NEXT I
200   GOTO 120
```

This program consists of an infinite loop that displays the keyboard matrix buffers over and over again. At the top of the loop (line 120), the cursor is placed in home position. Lines 130-160 display the first buffer on the top line of the display, and lines 170-190 display the second buffer on the second line of the display.

Two buffers are needed for the keyboard matrix in order to help distinguish among the following conditions: 1) a key has not been held down long enough to register; 2) a key has been just hit, but this is the first keyboard scan that detects it; 3) a key is depressed but has already been detected

during a previous scan; and 4) a key has been depressed long enough to activate the repeat feature.

In general, a key is detected in a two-step process, one step for each buffer. In the first step, the byte for the key column is picked up from the keyboard output port and compared with the current value in the first buffer, and then the current value in the first buffer is replaced by this new value. If the new and old values for the first buffer disagree, this must be the first scan to detect this change in the keyboard. In this case, no further action is taken for this particular column.

If the new and old values for the first buffer agree, the second step is performed. In this step the corresponding byte in the second buffer is checked and updated if it is different. This time no further action is taken if the values agree, because this means that the key has already been detected and should not be counted another time. However, if the values differ, the position of the key in the matrix is computed, ready for the decoding stage.

The key detection process produces several bytes of data that are used by the decoding process. Primary among these are FFA3h = 65,443d, which

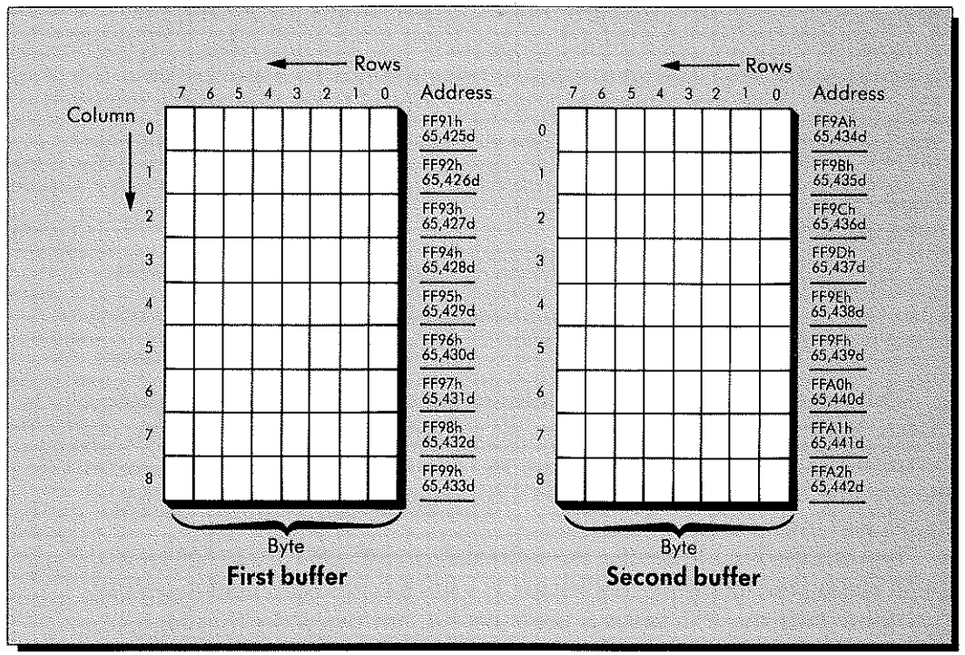

Figure 6-3. Keyboard matrix buffers

contains the shift code byte (for column nine), and FFA6h = 65,446d, which contains the position of the key in the matrix. This last value is obtained by multiplying the column position by eight and adding in the row position. The code for this calculation starts at 70F0h = 28,912d.

The repeat feature for keys is handled in the code starting at about 70B0h = 28,848d. The length of the keyboard buffer is first checked, for if there is more than one character in the buffer, the repeat feature will not be active. A counter at FFA4h = 65,444d counts down 84 cycles of keyboard scanning before the repeat feature is activated. Using the 12-millisecond cycle time, this gives about a one second delay before the key begins to repeat automatically. Once the key begins to repeat, for each cycle of the repeat, the counter is set back to six (at 70BBh = 28,859d), giving a frequency of about fourteen repeats per second. (This checks with the timing results we measured with a stopwatch.)

Key Decoding

In the decoding section the key positions are converted into ASCII codes. You can use the basic principles of this routine to convert key combinations to any sort of code that is required. The main job is in setting up the proper translation tables.

The decoding subsection begins at about 7122h = 28,962d (see box).

Routine: Key Decoding

Purpose: To convert key combinations into corresponding ASCII codes

Entry Point: 7122h = 28,962d

Input: Upon entry, the key position number is in location FFA6h = 65446d and the shift code (what shift keys have been depressed) is in location FFA3h = 65,443d.

Output: ASCII code for the corresponding character

BASIC Example: Not applicable

Special Comments: Not a callable routine — part of the background task

First the position of the key is retrieved from FFA6h = 65,446d. Figure 6-4 shows how the position numbers correspond to the labels on the keys.

The keys in the white area of this diagram are handled first. They have position numbers 0 through 43 and correspond to all the alphabetical, numerical, and punctuation keys. Not included are the keys around the outside of the keyboard, such as [ESC], [TAB], [CTRL], [SHIFT], [SPACE], [ENTER], and the function keys.

The keys with positions 0 through 43 map to ASCII in six different ways, depending upon the shift code. There are six tables in memory to handle these six different cases (see Appendix D). Each table is forty-four bytes long.

The code that manages these tables extends from 7122h = 28,962d to 71B8h = 29,112d. In this code, the BC, HL, and DE are used to help compute the table look-up address. The BC register pair holds the key position, the HL register pair holds the base address for the particular table, and the DE register pair holds the value 44 or 0 for use with the [SHIFT] key. These are all added together to compute the look-up address for the ASCII code.

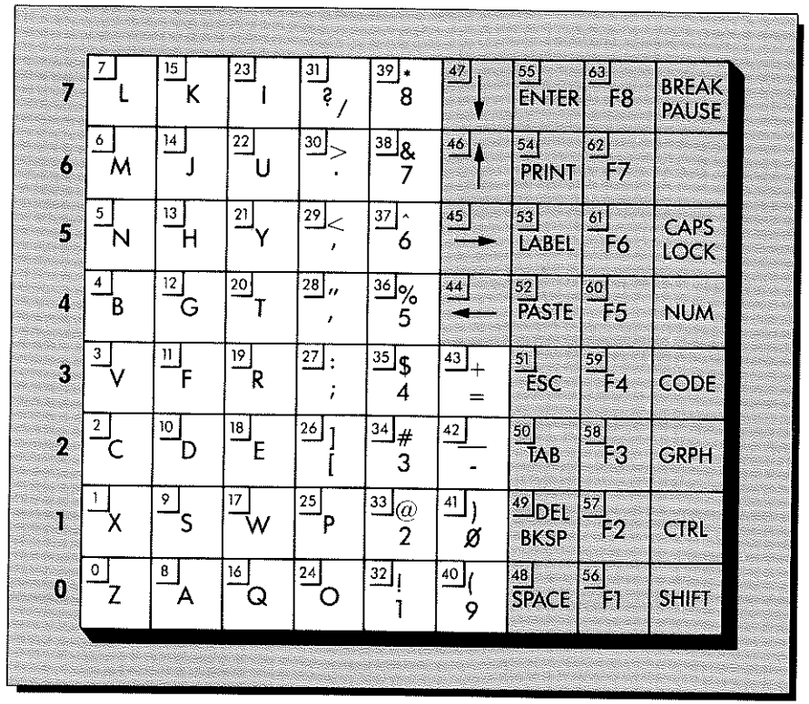

Figure 6-4. Keyboard matrix table

The first table starts at 7BF1h = 31,729d and contains the ASCII codes for the unshifted characters. This includes the lowercase letters, the numbers, and the unshifted punctuation mark keys. In this case HL is set equal to the value 7BF1h = 31,729d, and DE is made equal to zero for the look-up.

The second table starts at 7C1Dh = 31,773d and holds the ASCII codes for uppercase letters and shifted punctuation marks. In this case HL is loaded with 7BF1h = 31,729d as before, but DE has the value 2Ch = 44d. Thus the computed address will point into this second table. The CPU instructions at 7133h = 28,979d, 7136h = 28,982d, 7185h = 29,061d, and 7188h = 29,064d examine the [SHIFT] key bit and set DE accordingly.

The third table starts at 7C49h = 31,817d and holds ASCII codes generated while holding down the [GRPH] key, and the fourth table starts at 7C75h = 31,861d and holds ASCII codes for characters generated while holding down both the [GRPH] and the [SHIFT] keys. In both cases the HL register is loaded with the value 7C49h = 31,817d, but as above the DE register is used to "shift" between the two cases.

The fifth table starts at 7CA1h = 31,905d and holds ASCII codes generated while holding down the [CODE] key, and the sixth table starts at 7CCDh = 31,949d and holds ASCII codes for characters generated while holding down both the [CODE] and the [SHIFT] keys. Again, the HL register is loaded with a base value (this time 7CA1h = 31,905d), and the DE register is used to "shift" between the two cases.

There are other parts to the decoding routine to handle the [CTRL] key (at 720Ah = 29,194d), the [CAPS LOCK] key (at 722Ch = 29,228d), the [NUM] key (at 7233h = 29,235d), the function keys (at 715Bh = 29,019d), and the arrow keys (at 7222h = 29,218d). There are special tables in the ROM to handle some of these. If the [NUM] key is depressed, the table at 7CF9h = 31,993d (see Appendix P) is used to search for the values of the keys in the Model 100 keyboard's number pad area.

Two tables contain ASCII codes for the keys in positions 44 through 63 (see Appendix Q). The first table starts at 7D07h = 32,007d and contains ASCII codes for these keys when [SHIFT] is not depressed. The second table starts at 7D1Bh = 32,027d and contains ASCII codes for these keys when [SHIFT] is depressed. The routine for decoding these keys runs from 7148h = 29,000d to 7158h = 29,016d and from 7222h = 29,218d to 7229h = 29,225d.

Character Buffer Management

This section illustrates one way to build a buffer to receive characters from an I/O process. In Chapter 7, on serial communications devices, we will see another way.

In the buffer management subsection, starting at 71C4h = 29,124d, the ASCII codes for the keys are put into the keyboard input buffer (see box).

Routine: Keyboard Character Buffer Management

Purpose: To put key codes into the keyboard input buffer

Entry Points: 71C4h = 29124d, 71D5h = 29,141d, and 71E4h = 29,156d

Input: Upon entry, the ASCII code for the character is in the A register and/or C register (depending upon which entry point is used).

Output: The key codes are put in the keyboard input buffer.

BASIC Example: Not applicable

Special Comments: Not a callable routine — part of the background task

There are several entry points. The one at 71D5h = 29,141d is for placing characters in an empty buffer, and the one at 71E4h = 29,156d is for storing subsequent characters in the buffer.

The number of characters currently in the buffer is stored at location FFAAh = 65,450d. The buffer can store a maximum of thirty-two characters, and each character takes two bytes of storage. The buffer is a queue: That is, it is a list in which new entries are added to the end of the list and old ones are taken from the beginning.

For regular ASCII characters, the first byte contains the ASCII code and the second byte is zero. Certain other keys, namely the function keys and the [LABEL], [PRINT], [PRINT] with [SHIFT], and [PASTE], are stored with a numeric code in the first byte and a value of 255 in the second byte. The numeric codes for these keys are 0 through 11, respectively.

Here is a BASIC program that displays the keyboard input buffer. You can use it to watch this buffer as you hit various keys. Just run the program and watch the screen as you hit some keys. You should type slowly, because it takes a while for this BASIC program to make a full cycle. The first number displayed is the number of characters currently in the buffer. After that every two bytes is where a character is stored in the buffer. You should compare the display on the screen with the foregoing description of the buffer.

```
100 ' KEYBOARD BUFFER
110 '
120   PRINT CHR$(11);
130   FOR I = 65450 TO 65514
140    PRINT USING "####"; PEEK(I);
150   NEXT I
160   GOTO 120
```

On line 120 of this program, the cursor is put into the home position (upper left corner). On lines 130-150, the contents of the buffer are displayed. On line 160, the program loops back to line 120 to display the buffer again.

The Keyboard Input Routines

The keyboard input routines provide a way to grab characters from the keyboard character buffer. This is the "official" way to get characters from the keyboard. It is the only way that the rest of the Model 100's ROM routines can get characters from the keyboard.

The KYREAD Routine

There are several routines for getting characters from the buffer. For a few more than are presented here, see the *Model 100 ROM Functions* (700-2245) from Radio Shack.

Let's look first at a routine called KYREAD (see box), located at 7242h = 29,250d. It checks whether or not there is a character in the buffer. If there are no characters, it returns with the Z flag set (Z). If there is a character, it returns with the ASCII code for the key in the A register and the Z flag clear (NZ). However, if the C flag was also set, then the character corresponds to one of the special keys [F1] through [F8], [LABEL], [PRINT], [SHIFT][PRINT], and [PASTE]. In that case the A register contains a corresponding numeric code, 0 through 11.

Routine: KYREAD

Purpose: To read a key from the keyboard input buffer

Entry Point: 7242h = 29,250d

Input: From the keyboard

Output: When the routine returns, the Z flag indicates if there are any characters in the keyboard input buffer. If the Z flag is set (Z), there are no characters. If the Z flag is clear (NZ), the ASCII code of the next character is in the A register. If the C register is also set (C), then the character corresponds to one of the special keys (F1) through (F8), (LABEL), (PRINT), (SHIFT)(PRINT), and (PASTE). In that case the A register contains a corresponding numeric code, 0 through 11.

BASIC Example: Not applicable

Special Comments: None

The routine first turns off the background task and then checks the variable at FFAAh = 65,450d, which holds the number of characters in the buffer. If this is zero, then it returns, turning the background task back on.

If there is at least one character in the buffer, then its code is loaded into a register, and the other entries in the buffer are moved one character position toward the front of the buffer. Then the routine returns, turning on the background task.

The BRKCHK Routine

The next routine is called BRKCHK (see box). It checks for a break or pause character ((CTRL) C or (CTRL) S). These are also generated by the (BREAK)(PAUSE) keys. If a break character was hit, it returns with the zero flag clear (NZ) and the ASCII code for the break or pause character in the A register; otherwise it returns with the zero flag set (Z).

Routine: BRKCHK

Purpose: To check for break or pause character

Entry Point: 7283h = 29,315d

Input: From the keyboard

Output: If a break character was hit, it returns with the zero flag clear (NZ) and the ASCII code for the break or pause character in the A register; otherwise it returns with the zero flag set (Z).

BASIC Example: Not applicable

Special Comments: None

The BRKCHK routine is located at 7283h = 29,315d. It first checks location FFEBh = 65,515d. This location is set by the key detection routines. It is normally zero, but if a break or pause character is detected, the ASCII code for that character is stored in this location. Special codes with the eighth bit turned on are also stored here when a function key is detected.

The BRKCHK routine clears location FFEBh = 65,515d after checking it. It first checks the eighth bit to see if a function key has been detected. If this is nonzero, a function key must have been detected, and it processes an interrupt for that key. If the eighth bit is clear (zero), then the routine returns with the zero flag set according to the contents of location FFEBh = 65,515d (nonzero or zero according to whether or not a break or pause was detected).

The KEYX Routine

The last input routine we'll look at in this chapter is called KEYX (see box). It checks the keyboard queue for normal characters or a break. It does not actually return any characters, only CPU flags. If there is at least one character in the buffer, it returns with the Z flag clear (NZ); otherwise the zero flag is set (Z). If a break was hit, then the carry is set (C); otherwise it is clear (NC).

Routine: KEYX

Purpose: To check for a character from the keyboard

Entry Point: 7270h = 29,296d

Input: From the keyboard

Output: If there is at least one character in the buffer, the routine returns with the Z flag clear (NZ); otherwise the zero flag is set (Z). If a break was hit, then the carry is set (C); otherwise it is clear (NC).

BASIC Example: Not applicable

Special Comments: None

The routine is located at 7270h = 29,296d. It first calls the BRKCHK routine (described previously). If there is no break or pause character, it checks location FFAAh = 65,450d to see if there are any characters in the buffer, returning with the zero flag set accordingly. If there was a break or pause, it checks the ASCII code in FFEBh = 65,515d for a break (ASCII 3), as opposed to a pause (ASCII 13h = 19d). If indeed there was a break, the routine sets the carry and returns; otherwise it checks to see if the buffer has "pause" characters pending.

Summary

In this chapter we have explored the Model 100's keyboard. We have seen how to scan it using BASIC and how the ROM routines in the Model 100 interface between the keyboard and the rest of the computer. We have seen that the actual hardware keyboard is fairly simple, but the software is complicated by the fact that the SHIFT, CTRL, BREAK and PAUSE keys have to be handled in special ways.

7

Hidden Powers of the Communications Devices

Concepts

How an RS-232 serial communications line works
The RS-232 connector and the modem

ROM Routines for the Communications Devices

BASIC interrupt commands
Initializing and shutting down the UART
Dialing the telephone
Reading from the serial communications line
Writing to the serial communications line

The serial communications line provides a vital link between the Model 100 and other computers, allowing it to become part of a larger information handling system. Through the serial communications line, the Model 100 can be used as a terminal for other computers and as a detachable unit that can be used to carry files from a larger computer to places where larger computers are unavailable. For example, you can download a file into the Model 100 at work, edit it at home, and then bring it back to the main computer the next day. Through the modem connection, you can also connect to other computers over telephone lines.

In this chapter we will explore the secrets of the Model 100's modem and RS-232 serial communications devices. We will see how these devices work and how they are set up in the Model 100. Then we will study the ROM routines that control them.

How an RS-232 Serial Communications Line Works

There are two major ways to send computer data: parallel and serial. Parallel transmission is normally used within the computer and for short distances outside the computer such as between the computer and a printer. Serial transmission is the rule for longer distances such as over the phone lines or from one building to another.

With parallel transmission, all the bits of a byte of data are transmitted at the same time, each over its own separate signal line. With serial transmission, in contrast, the bits are sent one at a time over a single signal line (see Figure 7-1).

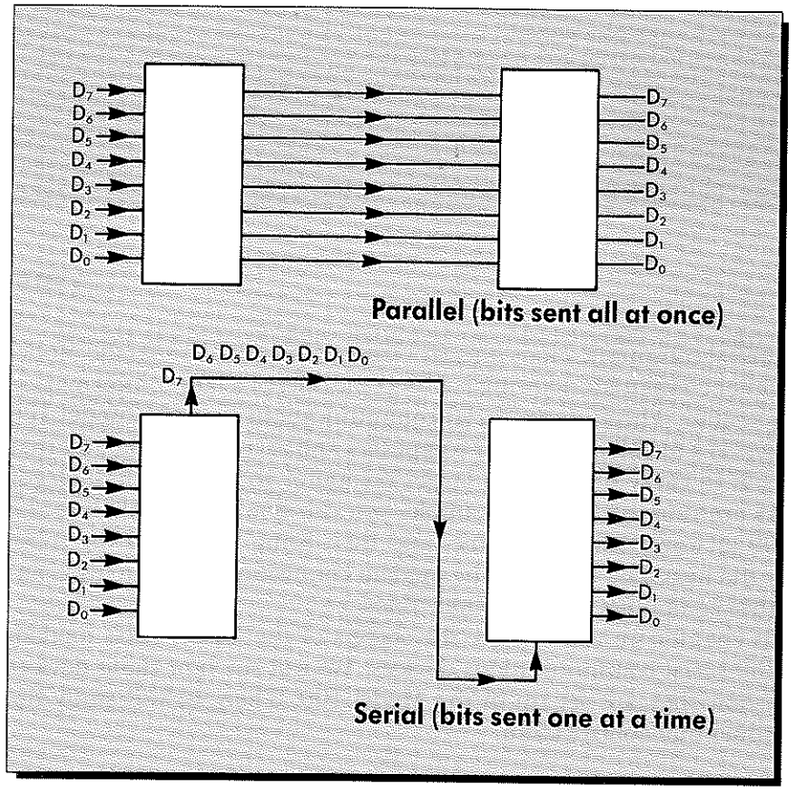

Figure 7-1. Parallel and serial transmission

Advantages of Serial Transmission

The main advantages of serial transmission are that fewer wires are needed and standards for this type of transmission are well established. The main disadvantages are that special hardware is needed to convert between the computer's internal parallel format and the external serial format and that serial transmission is limited to slower speeds than parallel transmission.

Let's look in more detail at the advantages, starting with the fact that fewer wires are needed for serial transmission. For a two-way serial communications line, a minimum of three signal lines (wires) is needed: one for each of the two directions and one as a ground line. Other signals can be added to provide hardware control of the flow of information. The serial communications port on the Model 100 contains six different signal lines plus a ground, but it's quite possible to use only three of these for many applications (see Figure 7-2).

The second advantage of serial transmission is its well-established standards. Standards increase the transportability of data and program files and thus increase overall productivity. The main standard for serial communications is called RS-232C; this is what the Model 100 uses. The RS-232C standard encompasses a number of different speeds (called baud rates) and a number of different formats (see Figure 7-3). When you set up your communications on the Model 100 you have to select the baud rate, word length, parity, and number of stop bits. Once these have been properly selected, you should be able to communicate with any other computer that has a RS-232C serial line with the same choices of speed and format. Since most computer systems have the option of communicating in this way, your Model 100 computer can communicate to most other systems.

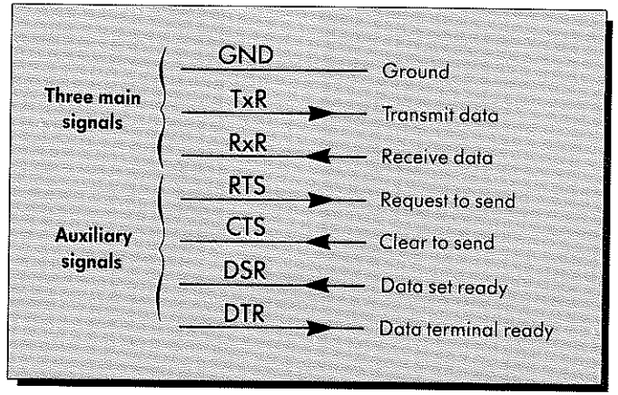

Figure 7-2. Signals for serial transmission

Disadvantages of Serial Transmission

Now let's look at the disadvantages of serial communications. The first is the extra hardware required. Fortunately, this hardware has been neatly packaged in computer chips called UARTs. This stands for *Universal Asynchronous Receiver Transmitter*. The word "Asynchronous" refers to the fact that bytes of data can be sent one by one as they become available, instead of in large blocks as with "synchronous" transmissions. There are a number of different UARTS available today. The Model 100 uses a UART chip called an IM6402.

The second disadvantage of serial transmission is speed. The difficulty is that the bits are sent one at a time. When a byte is to be sent, it is loaded into the UART in parallel. Then the UART peels off the bits, one by one, to be sent over the data signal line (see Figure 7-4). Sending one byte of information requires that a series of about ten bits be sent over the serial communications line. The extra bits are used to separate the bytes from each other. The opposite process happens when data are received. In this

Baud rate (speed)	75 baud 110 baud 300 baud 600 baud 1200 baud 2400 baud 4800 baud 9600 baud 19200 baud
Word length	6 bits 7 bits 8 bits
Parity	Ignore parity Odd parity Even parity No parity
Stop bits	1 stop bit 2 stop bits

Figure 7-3. Typical speeds and formats for serial transmission

case, the bits from the serial communications line accumulate in the UART until a byte is complete and then are transferred in parallel into the computer (see Figure 7-4). Special status lines from the UART to the computer indicate when each of these processes is complete and the UART is able to handle more characters.

Standard speeds for serial transmission range from 75 to 19,200 baud (bits per second) or 7.5 to 1920 bytes per second. Data bytes are transmitted inside the computer in parallel at speeds of up to several million bytes per second, but transmission between computers at such high speeds simply is not practical. The Model 100 can handle the full range of baud rates, but not without problems. For example, the LCD screen can display characters at a rate of only about 90 characters per second, and the practical limit for downloading files (reading them into the Model 100's memory) is about 240 characters per second.

To overcome differences in transmission speeds between devices, serial transmission lines send signals back and forth to control the flow of data. The Model 100 uses what is called an XON/XOFF protocol. With this method, the receiving device sends a CTRL S (XOFF) when it can no longer safely handle more data from the transmitting device. It sends a CTRL Q

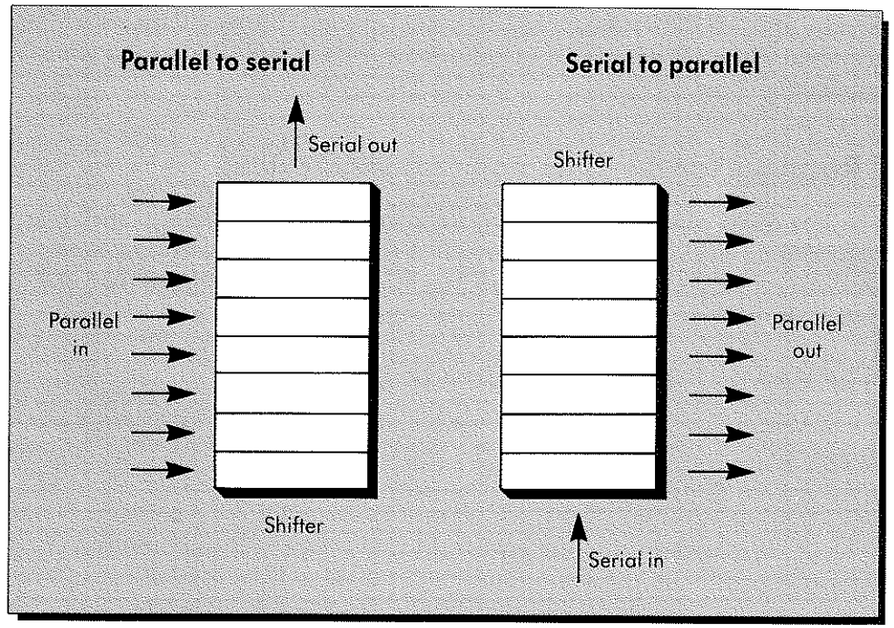

Figure 7-4. Converting from parallel to serial and from serial to parallel

(XON) when it can handle more. Later in the chapter we will study the ROM routines in the Model 100 that handle this protocol.

The RS-232C Connector and the Modem

The UART in the Model 100 is connected to the outside world in two ways: through the RS-232C connector and through the modem. The RS-232C connector provides a standard way of connecting two devices directly, and the modem converts back and forth between the serial bits of data of the UART and audio signals suitable for transmission over the telephone lines (see Figure 7-5).

Bit 3 of port BAh = 186d switches between the RS-232C connector and the modem. When this bit is 0, the UART is connected to the RS-232C connector, and when this bit is 1, the UART is connected to the modem (see Figure 7-6).

The modem and the RS-232C connector behave much the same to a programmer. There are, however, several differences. The modem has the automatic dial function, and the RS-232C connection has some extra status and control signals. As we shall see, dialing is done by taking the telephone rapidly on and off its hook. The extra signals for the RS-232C connector are RTS (Ready to Send), DTR (Data Terminal Ready), CTS (Clear to Send), and DSR (Data Set Ready). The Model 100 is set up as a data terminal; thus, the first two signal lines are output, and the second two are input.

ROM Routines for the Communications Devices

The ROM routines for the communications devices perform such tasks as handling BASIC interrupts, initializing the UART, switching between the modem and RS-232C connector, reading from and writing to the UART, and dialing the phone.

The BASIC Interrupt Commands

BASIC has interrupt commands for the communications line that are similar to those for the clock and the keyboard. The interrupt control commands are ON MDM GOSUB, ON COM GOSUB, MDM ON, COM ON, MDM OFF, COM OFF, MDM STOP, and COM STOP. The keyword MDM refers to the modem, and the keyword COM refers to the communications line in general, but, the same routines are used to handle both cases. This is because the interrupts happen in the UART, which lies between the computer and both these methods of transmitting serial data.

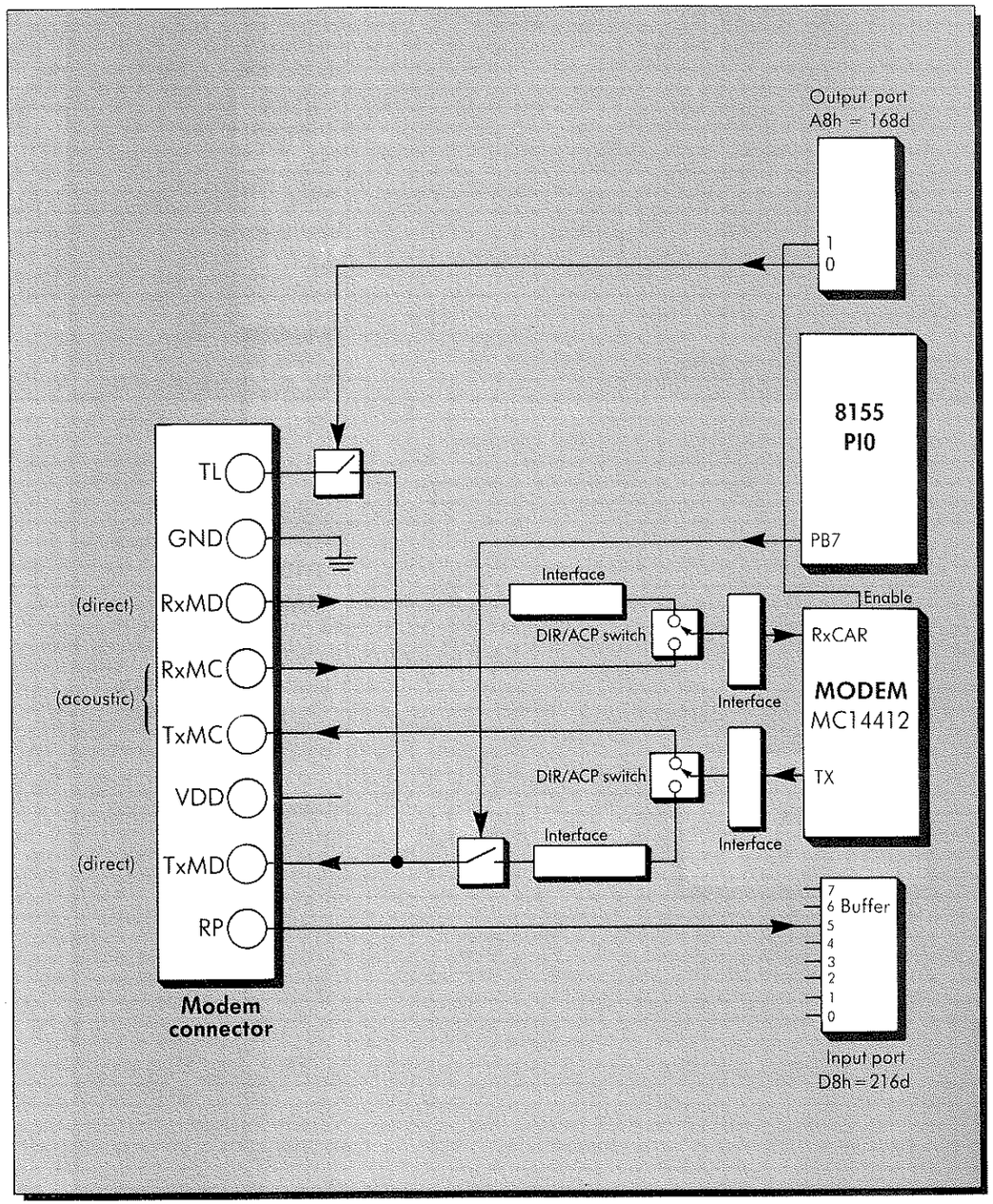

Figure 7-5. The modem

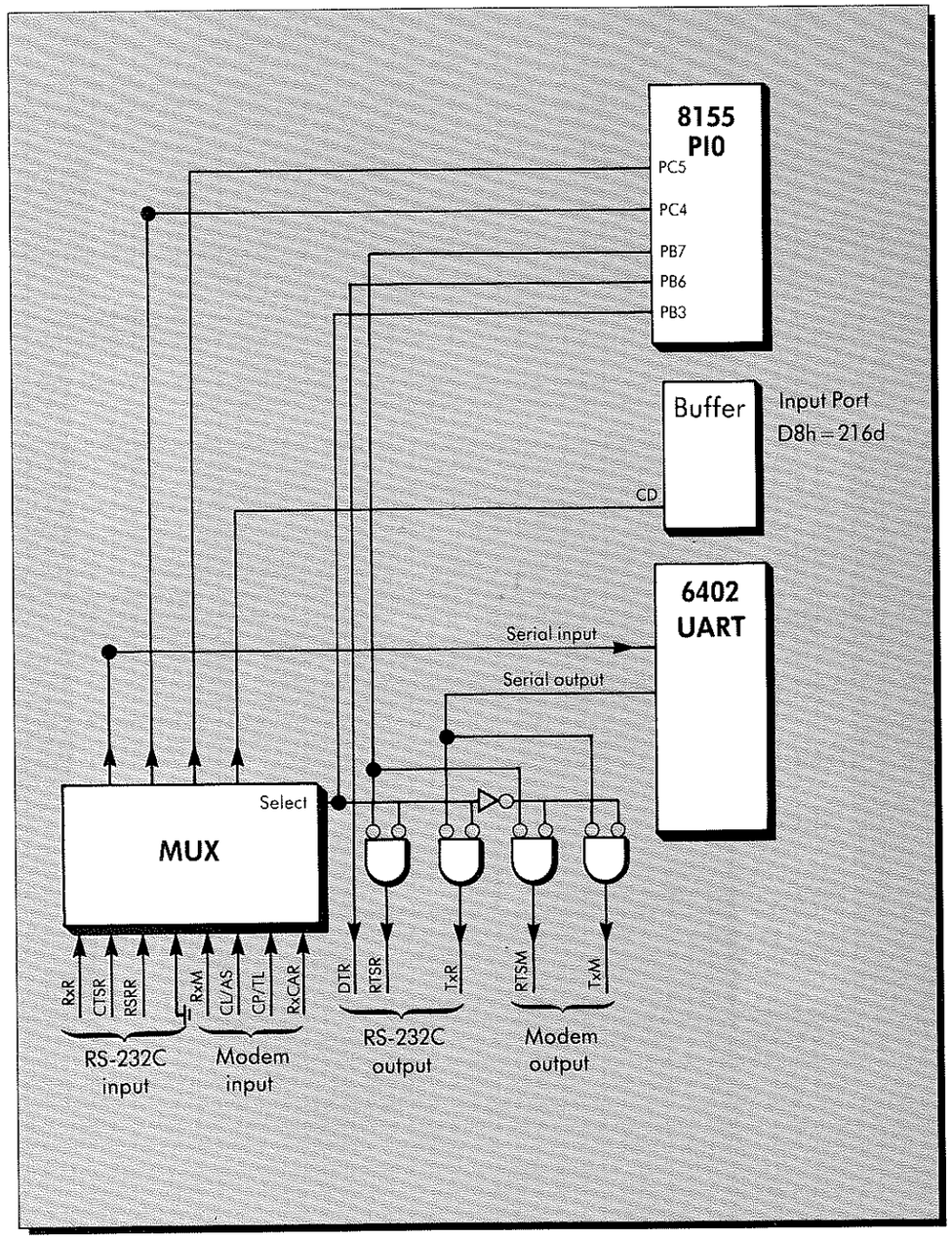

Figure 7-6. Switching between the modem and the RS-232C

In Chapters 5 and 6 we discussed how BASIC interrupt routines operate on the BASIC interrupt table (starting at F944h = 63,812d). The first three bytes of this table are reserved for the COM or MDM interrupt and contain the interrupt status word and the location of the BASIC interrupt subroutine for the communications devices.

Initializing and Shutting Down the UART

The routine to initialize the UART is called INZCOM and is located at 6EA6h = 28,326d (see box). This routine will help you to write your own code to set up the UART any way you want.

Routine: INZCOM

Purpose: To initialize the UART that controls the communications line

Entry Point: 6EA6h = 28,326d

Input: Upon entry, it expects the H register to specify the baud rate (1 through 9 as used in the STAT program), the L register to contain the UART configuration code, and the carry flag to indicate whether the modem or the RS-232C connector is to be used (set if RS-232C and clear if modem). In the configuration code, bit 0 specifies the number of stop bits (0 = 1 stop bit and 1 = 2 stop bits), bits 1 and 2 specify the parity (00 = none, 01 = even, and 10 = odd), and bits 3 and 4 specify the word length (00 = 6, 01 = 7, and 10 = 8 bits).

Output: When the routine returns, the UART is properly initialized.

BASIC Example:

```
CALL 28326,0,H
```

where H is a 16-bit number that contains the configuration information as specified above.

Special Comments: This routine will not update the configuration information that is stored in memory; thus, it is not permanent.

If the carry flag is clear, the modem was selected. In this case, the routine loads a 3 into the H register and 2Dh = 45d into the B register. This sets the baud rate to 300 and turns the modem on. Then it disables interrupts using the DI command and calls a routine called BAUDST to set the baud rate.

The BAUDST routine is located at 6E75h = 28,277d (see box). It expects the H register to contain a number from 1 to 9 that specifies the baud rate, as described above. This routine uses a table located at 6E94h = 28,308d to look up the correct patterns to send to ports BCh = 188d and BDh = 189d. These ports program a timer in the 8155 PIO chip to produce a square wave, which is sent to the UART to control the baud rate.

Routine: BAUDST

Purpose: To set the baud rate for the serial communications line

Entry Point: 6E75h = 28277d

Input: Upon entry, the H register contains a number from 1 to 9 that specifies the baud rate, using the same correspondence as is used by the STAT program.

Output: When the routine returns, the baud rate is set as specified.

BASIC Example:

```
CALL 28277,0,H
```

where H is 256 times (because it's the H register) the baud rate number decribed above.

Special Comments: None

Bits 6 and 7 of port BDh = 189d are always set to 01. This specifies that the output of the timer is a continuous square wave. The other possibilities are 00 for a single cycle of square wave, 10 for a single pulse, and 11 for continuous pulses. The other bits of these ports form a fourteen-bit binary number. Bits 0 through 7 of port BCh = 188d form the lower eight bits, and bits 0 through 5 of port BDh = 189d form the upper six bits. In a moment we will explain how this number helps to determine the baud rate, but first, let's trace the clock signals that control the baud rate.

The timing signal for the baud rate originates with the 4,915,200 cycles per second crystal that runs the CPU and provides the timing for the main circuits of the computer. The CPU divides this frequency in half and sends it on to drive other devices such as the timer in the 8155 PIO. This timer further divides the frequency by the fourteen-bit binary number previously referred to. The result is passed to provide a baud rate clock for both the receiver and the transmitter circuits in the UART. The actual baud rate produced by the UART is one sixteenth of the frequency of this last signal.

For example, if you select 300 baud, the fourteen-bit binary number will have a value of 512 in the table. The baud rate will be 4,915,200 divided by 2, divided by 512, divided by 16, which is exactly 300.

After ports BCh = 188d and BDh = 189d are programmed, port B8h = 184d is set to a value of C3h = 195d. The upper two bits of this byte start the timer, and the lower six bits define how ports B9h = 185d, BAh = 186d, and BBh = 187d are to be used. They are not changed from the way that they are originally initialized. The BAUDST routine then returns.

The INZCOM routine continues by sending the contents of the B register out port BAh = 186d. If the modem is specified, 2Dh = 45d is sent, and if the RS-232C connector is selected, 25h = 37d is sent. The difference is bit 3, which controls the switch selecting between these two modes of operation for the serial line.

Next the INZCOM routine sends the configuration code to the UART via port D8h = 216d. The upper three bits are masked off because they are not used.

Next the routine at 6F39h = 28,473d is called (see box). This routine clears three locations: FF40h = 65,344d, FF86h = 65,414d, and FF88h = 65,416d. The first of these locations controls the XON/XOFF protocol, the second specifies how many bytes are in the buffer for incoming characters from the serial communications line, and the third is a pointer to the position in the buffer for incoming characters.

Routine: Initialize Serial Buffer Parameters

Purpose: To initialize parameters that manage the serial communications line.

Entry Point: 6F39h = 28,473d

Input: None

Output: When the routine returns, three locations — FF40h = 65,344d, FF86h = 65,414d, and FF88h = 65,416d — are cleared.

BASIC Example:

```
CALL 28473
```

Special Comments: None

Next the INZCOM places a value of FFh = 255d in location FF43h = 65,347d and returns. This location tells the system when the serial communications line is active. The return is made by jumping to a place where there is the proper number of POPs and then a RETurn. This just happens to be the very last part of the background task discussed in Chapters 4, 5, and 6.

The function of the CLSCOM routine (see box) is the opposite of that for the INZCOM routine: CLSCOM deactivates the communications line. It is located at 6ECBh = 28,363d. It sets bits 6 and 7 of port BAh = 186d. Bit 7 hangs up the telephone, and bit 6 places the DTR (Data Terminal Ready) line of the RS-232C connector in the "not ready" state.

Routine: CLSCOM

Purpose: To deactivate the serial communications line

Entry Point: 6ECBh = 28363d

Input: None

Output: When the routine returns, the telephone connection is broken and the RS-232C DTR line is placed in the "not ready" state.

BASIC Example:

```
CALL 28363
```

Special Comments: None

Dialing the Telephone

The Model 100 dials the telephone by a method that in effect, rapidly takes the telephone on and off the "hook". A relay in the modem circuit acts as the "hook". This relay is controlled by bit 7 of port BAh = 186d. When the telephone is hung up, this bit is 1, and when the telephone is off the hook, it is 0.

After we present the ROM routines for dialing the telephone, we will provide a program that shows how you can dial the telephone from BASIC.

The routine for dialing the telephone is called DIAL and is located at 532Dh = 21,293d (see box). Upon entry the HL register pair contains the address of the telephone number. The routine first sets bit 3 of port BAh = 186d, selecting the modem instead of the RS-232C connector. Next the routine calls a routine at 5359h = 21,337d, which does the actual dialing (see box).

Routine: DIAL

Purpose: To dial the telephone

Entry Point: 532Dh = 21,293d

Input: Upon entry, the HL register points to where the telephone number is stored in memory.

Output: The routine dials the telephone number

BASIC Example:

```
CALL 21293,0,H
```

Special Comments: None

Routine: Dialing Routine

Purpose: To dial the telephone

Entry Point: 5359h = 21,337d

Input: Upon entry, the HL register pair points to where the telephone number is stored in memory.

Output: The routine dials the telephone.

BASIC Example:

```
CALL 21337,0,H
```

where H is the address of the telephone number

Special Comments: None

The dialing routine at 5359h = 21,337d calls various other routines and then gets down to the business of dialing the telephone. The dialing loop runs from 5370h = 21,360d to 539Bh = 21,403d. At the top of the loop, a check is made for the BREAK key, and the dialing routine is exited if BREAK is detected. Next, if the routine continues, a digit or other symbol is fetched from the telephone number string. It is checked to see if it is a special symbol such as CR, LF, or CTRL Z. If it is a CR/LF sequence or a CTRL Z, the routine jumps to a section of code in which the auto log on sequence is performed. If it is not, the routine skips any spaces in the telephone number and sets the carry if it finds a digit. If there was a digit, it calls a routine at 540Ah = 21,514d to dial the digit (see box).

Routine: Dial a Digit

Purpose: To dial a digit of a telephone number

Entry Point: 540Ah = 21,514d

Input: Upon entry, the digit (ASCII code or actual value) is in the A register.

Output: When the routine returns, the digit is dialed.

BASIC Example:

```
CALL 21514,A
```

where A contains the digit

Special Comments: None

The digit dialing routine at 540Ah = 21,514d prints the digit on the screen, then converts it from ASCII to a numerical value by masking off all but the lower four bits and converting zero values to ten. This numerical value is used as a loop counter to control the number of pulses that are to be sent out. Note that a zero on the telephone dial is really a ten when sent over the line.

The pulse loop begins by getting the timing delay selected by STAT (10 or 20) and using it to set a counter in the DE register pair for a pair of delay loops that control the pulse timing. If a rate of 10 pulses per second was selected, a value of 2440h = 9280d is chosen, and if a 20 pulse per second rate was selected, a value of 161Ch = 5660d is chosen. Next the pulse loop disconnects the phone by calling a routine at 52C1h = 21,185d. This is part of the "official" routine for disconnecting the phone line that begins at

52BBh = 21,179d (see box). The "disconnect" is made by setting bit 7 of port BAh = 186d equal to 1, leaving the other bits undisturbed. Next the E register of the DE pair is used to count a delay with the telephone on the hook. The pulse loop then reconnects the telephone by calling a routine at 52B4h = 21,172d, which clears bit 7 of port BAh = 186d. This is part of the "official" routine for connecting the phone line that begins at 52D0h = 21,200d (see box). Finally, the D register is used to count a delay with the telephone off the hook (connected). During the pulse loop, interrupts are disabled (using the DI instruction) because the timing loops are critical and must not be interrupted.

Routine: DISC

Purpose: To disconnect the telephone line

Entry Point: 52BBh = 21,179d

Input: None

Output: When the routine returns, the telephone line is disconnected.

BASIC Example:

```
CALL 21179
```

Special Comments: None

Routine: CONN

Purpose: To connect the telephone line

Entry Point: 52D0h = 21,200d

Input: None

Output: When the routine returns, the telephone line is connected.

BASIC Example:

```
CALL 21200
```

Special Comments: None

After the pulse loop there is a short delay to separate the digit from other digits of the telephone number. Then the dialing routine continues by checking for more special characters, including "<" for beginning of the auto log on sequence and "=" for a two-second delay.

The routine for the auto log on sequence begins at 539Eh = 21,406d (see box). It calls serial communications line routines to send and receive characters through the modem as specified by the user. The auto log on sequence then returns to the main DIAL routine.

Routine: Auto Log On

Purpose: To send auto log on sequence

Entry Point: 539Eh = 21,406d

Input: Upon entry, the HL register pair points to the descriptor for the auto log on sequence.

Output: The routine sends the auto log on sequence out the serial communications line.

BASIC Example:

```
CALL 21406,0,H
```

where H is the address where the auto log on sequence is stored in memory.

Special Comments: None

In DIAL, bit 3 of port BAh = 186d is restored to its original value, and various other business is taken care of depending upon whether you are in terminal mode or just using the computer to dial a voice call for you.

Dialing from BASIC

Here is a BASIC program that uses the DIAL routine to dial a telephone number. It prompts you for the telephone number and then dials the number for you. You can use this program as a starting point for developing more elaborate telephone routines. For example, you could write a program that waits until a specified time, dials a certain number, logs onto a computer at the other end of the line, sends some data, and then hangs up the telephone.


```
100 ' DIAL A TELEPHONE NUMBER
110 '
120   INPUT "TELEPHONE NUMBER";T$
130   S$ = T$+CHR$(13)
140   S = VARPTR(S$)
150   H = PEEK(S+1)+256*PEEK(S+2)
160   CALL 21293,0,H
170   CALL 21179
180   PRINT
190   GOTO 120
```

On line 120 of this program, the telephone number is input by the user. On lines 130-150, the address where this number is stored in memory is computed. On line 160, the DIAL routine is called with the address of this telephone number in the HL register pair. On line 170, the program hangs up the telephone, assuming that the user is on the line with a regular handset. On line 190, it loops back to try another number.

Reading from the Serial Communications Line

The serial communications line is interrupt driven. That is, whenever characters are ready to be received, the UART actuates an interrupt to a special routine to put the character in a buffer (see Figure 7-7). Whenever the computer needs a character from the communications line, it checks the buffer, not the UART directly. Let's look at the routines to handle the interrupt and to fetch characters from the buffer.

The Serial Communications Interrupt Service Routine

The interrupt service routine for the serial communications line takes in characters from the line as they are generated. Whenever the UART receives another character, it triggers an interrupt.

The Model 100 uses interrupt 6.5 for input to its serial communications line. This means that whenever the interrupt is triggered, the CPU stops what it is doing (if this interrupt is enabled) and calls location 34h = 52d. On the Model 100, the code starting at 34h = 52d disables further interrupts and jumps to location 6DACh = 28,076d.

The interrupt service routine for the serial communications line continues at location 6DACh = 28,076d (see box). It first calls a routine in RAM at F5FCh = 62,972d. This normally consists of just a RETurn instruction. Since it is in RAM, it provides a way for you to take control of the interrupt routine, perhaps placing a jump at F5FCh = 62,972d to your own interrupt routine for the serial communications line. For example, you could use such a routine in a communications program that transfers files in special formats.

Routine: Serial Interrupt Service Routine

Purpose: To read a character from the serial communications line

Entry Point: 6DACh = 28076d

Input: Upon entry, a character is ready for input from the serial communications line.

Output: When the routine returns, the character is placed in the serial communications input buffer.

BASIC Example: Not applicable

Special Comments: None

The main part of the interrupt routine manages input to a circular buffer from the serial communications line. A circular buffer is a buffer that wraps around on itself (see Figure 7-8). This particular circular buffer is 64 bytes long, starting at location FF46h = 65,350d. A pointer at FF88h = 65,416d gives the position within the buffer for bytes as they are input from the serial line. A pointer at FF87h = 65,415d gives the position

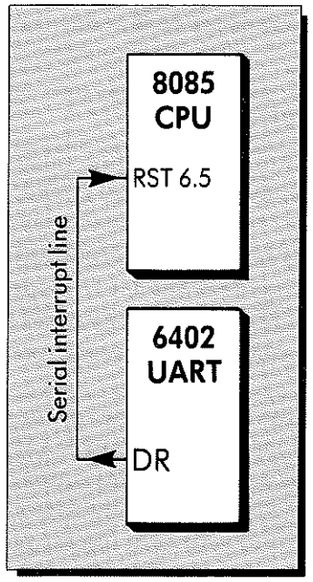

Figure 7-7. The UART interrupt

for bytes as they are removed from the buffer by other routines. Another variable, located at FF86h = 65,414d, keeps track of the number of bytes currently stored in the buffer.

The buffer management routine starts by pushing the HL, DE, BC, and PSW registers onto the stack, saving what the CPU was doing just before the interrupt occurred. Then the address 71F7h = 29,175d is pushed onto the stack, providing a return address with the proper sequence of POPs to restore the registers upon return from the interrupt. Again, this code is borrowed from the background task.

Next, port C8h = 200d is read into the accumulator. This is the data byte from the serial communications line. It will be placed in the circular buffer. First, however, it must be processed according to the choice of parity.

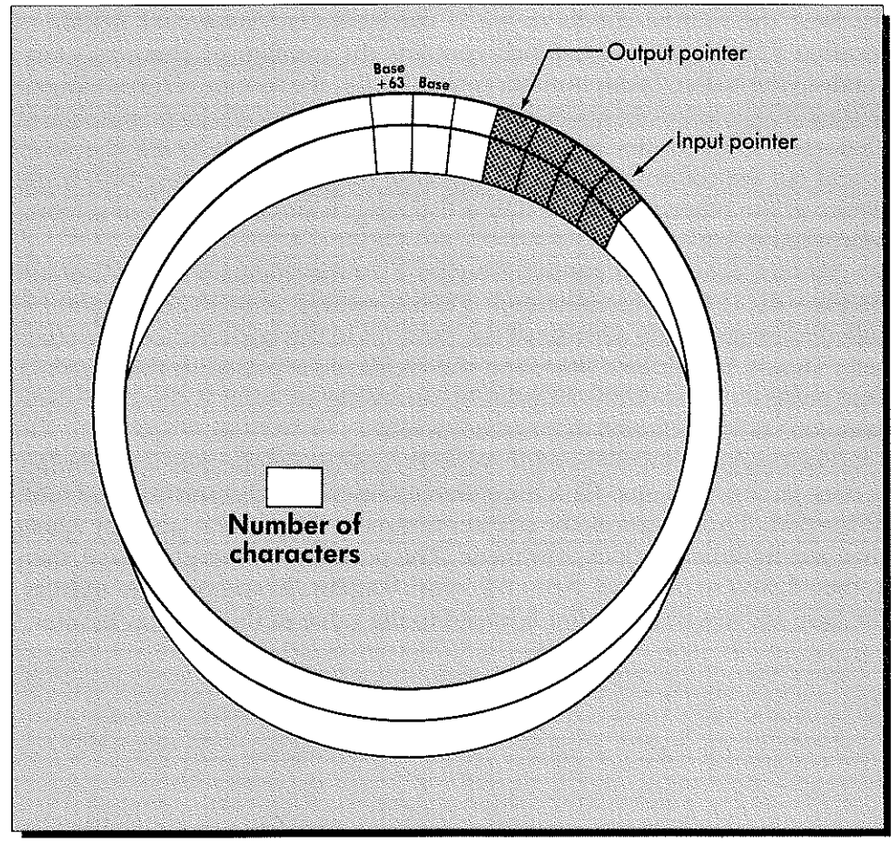

Figure 7-8. A circular buffer

The byte is ANDed with the contents of FF8Dh = 65,421d. If parity is ignored, this location contains 7Fh = 127d; otherwise, it contains FFh = 255d. In the first case it masks off the parity, and in the second case it has no effect. The result is moved to the C register.

Next, the routine checks for errors from the communications line. It reads port D8h = 216d to get the status byte of the UART. Only bits 1, 2, and 3 are used. The rest are masked off. They carry the following error signal lines: OE (Overrun Error), FE (Framing Error), and PE (Parity Error). The byte containing these bits is saved in the B register for later use.

If all of the error bits are zero, the incoming character is checked for XON/XOFF protocol characters. If the routine finds CTRL Q (XON), location FF40h = 65,344d is cleared, and if it finds CTRL S (XOFF), that location is made nonzero. Next, the routine checks location FF42h = 65,346. If this is nonzero, it returns without further action.

Next the routine checks to see if the buffer is already full. It checks location FF86h = 65,414d, which contains the number of characters currently in the buffer. If this number is equal to 64, the routine returns without further action because this indicates that the buffer is full. If the number of characters in the buffer is greater than or equal to 40, an XOFF character is sent out the communication line because the buffer is getting full. If the device at the other end of the line is listening, it should soon stop sending more characters.

Now the character is put in the buffer. First the character count (location FF86h = 65,414d) is incremented. Then a routine at 6DFCh = 28,156d is called to compute the address of the position in the buffer where the character should be put (see box), and the character is placed in the buffer at that position. The routine finishes by recording an error if the status byte sampled earlier indicates that one occurred.

The computation of the address of the position of the byte in the buffer is done in several steps. First the pointer is incremented and ANDed with 3Fh = 63d. This advances the pointer around a 64-position number wheel that uses modular or clock arithmetic. The position numbers start at 0, then 1, then 2, and so on to 63. After 63 comes 0 again. To compute the address, the base address of the buffer is added to the value of this circular pointer.

Routine: Address Computation for Circular Buffer

Purpose: To compute the address for a pointer into the circular buffer

Entry Point: 6DFCh = 28,156d

Input: Upon entry, the HL register pair points to the location in memory where the pointer is stored. The pointer is contained in a byte and has a value from 0 to 63.

Output: When the routine returns, the HL register pair contains the actual address of the buffer entry, and the DE register pair contains the address of the pointer. The value of the pointer is incremented unless it was 63, in which case it is set back to 0.

BASIC Example: Not applicable

Special Comments: None

Serial Communications Input Routines

Next, let's look at the routines that fetch characters from the buffer for use by the rest of the computer. There is a routine called RCVX to return the number of characters in the buffer and a routine called RV232C to fetch a character from the buffer. An assembly-language programmer can use these routines to send data to the serial communications line.

The RCVX routine is located at 6D6Dh = 28,013d (see box). This routine returns the number of characters currently stored in the serial communications input buffer in the A register. It also sets the Z flag accordingly. That is, if A is zero, then Z is set; otherwise, it is clear.

Routine: RCVX

Purpose: To get the number of characters currently in the serial communications input buffer

Entry Point: 6D6Dh = 28,013d

Input: None

Output: When the routine returns, the number of characters currently stored in the serial communications input buffer is in the A register, and the Z flag is set accordingly.

BASIC Example: Not applicable

Special Comments: None

The RCVX routine gets the number of characters currently in the buffer from location FF86h = 65,414d, ORing it with itself to set the zero flag if the buffer is empty (Z means empty and NZ means not empty). It also checks locations FF40h = 65,344d and FF41h = 65,345d for special conditions involving the XON/XOFF protocol.

The RV232C routine is located at 6D7Eh = 28,030d (see box). Its job is to get a character from the serial communications input buffer. Upon exit the A register contains the ASCII code of the character from the buffer. The zero flag is set (Z) if there is no error and clear (NZ) if there was an error. The carry is set (C) if the BREAK key was hit and clear (NC) otherwise.

Routine: RV232C

Purpose: To get a character from the serial communications input buffer

Entry Point: 6D7Eh = 28,030d

Input: None

Output: When the routine returns, the A register contains the ASCII code of the character from the buffer. The zero flag is set (Z) if there is no error and clear (NZ) if there was an error. The carry is set (C) if the BREAK key was hit and clear (NC) otherwise.

BASIC Example: Not applicable

Special Comments: None

The RV232C routine pushes the HL, DE, and BE registers on the stack and pushes the address 71F8h = 29,176d on the stack for the return address. This is where the proper number of POPs and a RETurn are located.

Next the RV232C routine goes into a loop in which it waits for a character from the buffer. The loop first calls a routine at 729Fh = 29,343d to check for a BREAK key, returning with the carry set if it detects this key (see box). If there was no BREAK, the RCVX routine is called to check the queue. If the queue is empty, the loop keeps looping. The loop will exit normally as soon as there is a character in the buffer.

Routine: BREAK check

Purpose: To check for BREAK key

Entry Point: 729Fh = 29,343d

Input: None

Output: The routine returns with the carry flag set if it detects the BREAK key.

BASIC Example: Not applicable

Special Comments: None

If there are fewer than three characters in the buffer, a routine called SEND Q, at 6E0Bh = 28,171d, is called to send CTRL Q (XON) out the communications line (see box later in the chapter).

The RV232C then gets the next character out of the buffer, using the routine at 6DFCh = 28,156d (described earlier) to compute its address within the buffer. It checks for an error condition left by the interrupt service routine and then returns.

Writing to the Serial Communications Line

There are two routines for sending characters out the serial communications line: SNDCOM and SD232C. The first simply sends characters, while the second uses the XON/XOFF protocol. If you are an assembly-language programmer, you can use these routines to send bytes out the serial communications line.

The SNDCOM routine is located at 6E3Ah = 28,218d (see box). It sends a single byte out the serial communications line, either to the modem or to the RS-232C connector, whichever is currently selected. It expects the character in the C register.

Routine: SNDCOM

Purpose: To send a character out the serial communications line

Entry Point: 6E3Ah = 28,218d

Input: Upon entry, the ASCII code of the character is in the C register.

Output: The routine sends the character out the serial communications line.

BASIC Example: Not applicable

Special Comments: None

The SNDCOM routine contains a loop that waits for the communications line to be ready to output the next character. This loop calls the BREAK detector routine at 729Fh = 29,343d (described previously) and then reads port D8h = 216d, checking bit 4 to see if the UART is free to accept the next character. If the bit is zero, the loop continues looping; otherwise, it does a normal exit, and the routine continues and sends the character out port C8h = 200d before it returns.

The SD232C routine is located at 6E32h = 28,210d (see box). It sends a character out the communications line using the XON/XOFF protocol.

Routine: SD232C

Purpose: To send a character out the communications line using the XON/XOFF protocol

Entry Point: 6E32h = 28,210d

Input: Upon entry, the ASCII code of the character is in the A register.

Output: The character is sent out the serial communications line.

BASIC Example:

```
CALL 28210,A
```

where A is the ASCII code of the character.

Special Comments: None

The SD232C routine calls a routine at 6E4Dh = 28,237d, which does the protocol. This routine waits if the communications line is to be held up by an XOFF. If a BREAK is detected, it returns with the carry flag set. In this case, the SD232C routine exits without sending the character; otherwise, it runs on into the SNDCOM routine to send the character.

Serial Transmission from BASIC

Here is a BASIC program that sends characters out the serial communications line. It prompts the user for a string, sends it out the serial communications line, and then loops back for another string.

```
100 ' SEND BYTES TO SERIAL PORT
110 '
120   PRINT "STRING TO SEND"
130   INPUT T$
140   T = VARPTR(T$)
150   X0 = PEEK(T)
160   X1 = PEEK(T+1)+256*PEEK(T+2)
170   FOR X = X1 TO X1+X0-1
180    CALL 28210,PEEK(X)
190   NEXT X
200   GOTO 120
```

On lines 120-130 of this program, the string is input. On lines 140-160, the length and address of the string are computed. The variable X0 contains the length of the string, and the variable X1 contains its address. On lines 170-190, the bytes of the string are sent by getting them one by one from memory and calling the SD232C routine each time. On line 200, the program loops back for the next string.

BASIC also has ways of sending bytes to the communications line without directly calling machine language. The following program shows one such method.

```
100 ' SEND BYTES TO COM LINE
110 '
120   OPEN "COM:68N2E" FOR OUTPUT AS #1
130   PRINT "STRING TO SEND"
140   INPUT T$
150   PRINT #1, T$;
160   GOTO 130
```

This program is shorter and easier to follow than the previous one, but of course it does not reveal much about the inner workings of the ROM.

Protocal Routines

Now let's look at the main protocol routine. This routine first checks location FF42h = 65,346d. If this location is zero, then the communications line is in a "go" condition, and the protocol routine returns ready to let the output of the character occur. If this location is nonzero, the line is in the "stop" condition. In that case, the character is checked to see if it is a CTRL Q (XON). If it is, the routine zeros locations FF8Ah = 65,418d and FF41h = 65,345d and returns. If not, it checks for CTRL S (XOFF). If the character is found, a value of FFh = 255d is stored in location FF41h = 65,345d, and the routine returns. If not, it goes into a loop that waits for either the BREAK key to be detected (returning with the carry set) or location FF40h = 65,344d to be zero.

Let's finish with two other protocol routines: SENDCQ and SENDCS. The first sends an XON (CTRL Q), and the second sends an XOFF (CTRL S) to the serial communications line.

The SENDCQ routine is located at 6E0Bh = 28,171d (see box). It first checks location FF42h = 65,346d. If this is zero, it returns. If not, it checks location FF8Ah = 65,418d. If this is not one, it returns without further action. If this location is one, the routine stores a zero in FF8Ah = 65,418d and returns, sending out an XON character.

Routine: SENDCQ

Purpose: To turn on the XON/XOFF protocol for incoming characters from the serial communications line

Entry Point: 6E0Bh = 28,171d

Input: None, except for the XON/XOFF control variables FF42h = 65,346d and FF8Ah = 65,418d

Output: The routine turns on the XON/XOFF protocol, sending an XON character (ASCII 11h = 17d) out the serial communications line if needed.

BASIC Example:

```
CALL 28171
```

Special Comments: None

The SENDCS routine is located at 6E1Eh = 28,190d (see box). Like SENDCQ, it first checks location FF42h = 65,346d. Again, if this is zero, it returns. Otherwise, it checks location FF8Ah = 65,418d. This time it returns if this location is not zero. If it is zero, it sets this location to one and returns, sending out an XOFF character.

Routine: SENDCS

Purpose: To turn off the XON/XOFF protocol for incoming characters from the serial communications line

Entry Point: 6E1Eh = 28,190d

Input: None, except for the XON/XOFF control variables FF42h = 65,346d and FF8Ah = 65,418d

Output: The routine turns off the XON/XOFF protocol, sending an XOFF character (ASCII 13h = 19d) out the serial communications line if needed.

BASIC Example:

```
CALL 28190
```

Special Comments: None

Summary

In this chapter we have studied the two serial communications channels for the Model 100 computer: the modem, which connects the Model 100 to the telephone, and the RS-232C connector, which provides a standard method for connecting the Model 100 directly to other computers.

We have seen that both communications channels are handled by the same UART (Universal Asynchronous Receiver Transmitter) chip. We have shown how to switch between the two channels, how to initialize the UART, and how to send and receive characters through it. We have also studied the circular buffer that manages the flow for incoming characters and the XON/OFF protocol that is used to prevent the buffer from overflowing.

8

Hidden Powers of Sound

Concepts
- How sound works in the Model 100

ROM Routines for Sound
- The BEEP command
- The SOUND command

Sound makes computers more friendly. On the simplest level, for instance, a "beep" sound can inform you that you've made an error while entering data. In games, sound can provide reinforcement for winning plays. Sound is likely to be an integral part of the input/output systems for personal computers in the future.

In this chapter we will explore the secrets of sound on the Model 100. You will see how to turn the sound on and off through bits on ports of the 8155 PIO and how to program the timer that produces a tone for the sound circuit. These techniques have the potential of providing much more versatile control of sound than can be obtained using BASIC. We will also look at how the BEEP and SOUND commands work.

How Sound Works in the Model 100

The sound circuits in the Model 100 consist of a timer and two switches (see Figure 8-1). The timer is the same one that generates the baud rate for the UART, as explained in the previous chapter, and it is programmed in the same way.

The first of the two switches is controlled by bit 5 of port BAh = 186d. It turns the sound on and off. A value of 1 turns it on and a value of 0 turns it off.

The second switch is controlled by bit 2 of port BAh = 186d. It connects and disconnects the output of the timer from the sound circuit. A value of 0 makes the connection and a value of 1 breaks the connection.

The following BASIC program demonstrates how the switches and the timer can be programmed directly to make sounds of various frequencies. It asks you for the frequency divisor D, which is the value loaded into the timer. The actual frequency is given by the formula:

2,457,600/D

where D is the divisor that you specify.

```
100 ' MAKE A SOUND
110 '
120   INPUT "FREQUENCY DIVISOR";D
130 '
140 ' PROGRAM THE TIMER
150   OUT 188,(D MOD 256)
160   OUT 189,((D/256) AND 127) OR 64
170   OUT 184,195
180 '
```

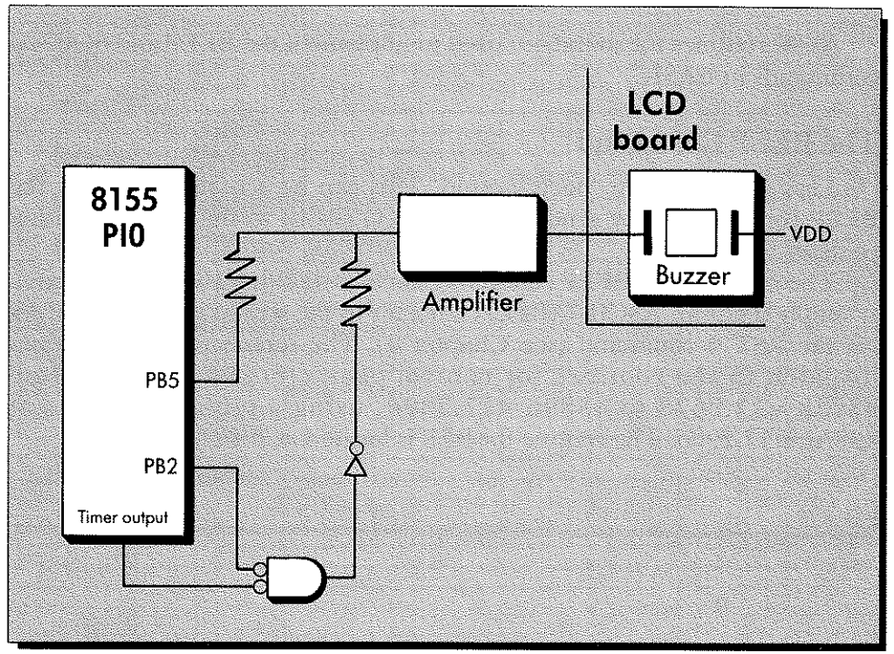

Figure 8-1. The sound circuit of the Model 100

```
190  ' TURN THE SOUND ON
200    X = INP(186)
210    OUT 186, (X AND 219) OR 32
220    GOTO 120
```

Let's look at this program in more detail. The frequency divisor D is input on line 120. On line 150, the lower eight bits are sent to port BCh = 188d, and on line 160, the upper six bits are sent to port BDh = 189d. In addition, bits 6 and 7 of port BDh = 189d are set to binary 01. This ensures that a square wave is produced.

In line 170, the timer is started by sending C3h = 195d to port B8h = 184d. This port programs the way the timer and the ports on the 8155 PIO chip will be used. The upper two bits program the timer, and the lower six bits program the ports. The pattern C3h = 195d ensures that the parallel ports of the PIO will remain as they are.

In lines 200-210, bit 5 is set to one and bit 2 is cleared in port BAh = 186d. This makes the connections from the timer to the sound circuit.

In line 220, the program loops back to get the next tone.

The ROM Routines for Sound

ROM routines for sound are used to implement the BEEP and SOUND commands in BASIC.

The BEEP Command

Let's start with the "beep" sound. This can be actuated either by the BEEP command or by sending a CTRL G (BEL) to the screen printing routines.

The code for the BEEP command starts at 4229h = 16,937d (see box). It simply sends an ASCII 7 (BEL) character to the screen printing routines via the RST 4 command (see Chapter 4). The screen printing routines dispatch to the routines for control characters in the code from 4373h = 17,267d to 4389h = 17,289d, using a table that starts at 438Ah = 17,290d. The routine for BEL is located at 7662h = 30,306d.

Routine: BEEP

Purpose: To sound a beep

Entry Point: 4229h = 16,937d

Input: None

Output: To the sound system

BASIC Example:

```
CALL 16937
```

Special Comments: None

The physical routine to generate the Beep is at 7662h = 30,306d. Here the routine at 765Ch = 30,300d is called to turn off interrupts (see box). Then the B register is cleared to set up a loop count of 256 for the BEEP loop, which is next. This loop first calls a routine at 7676h = 30,326d to flip bit 5 of port BAh = 186d (see box). Then it calls a routine at 7657h = 30,295d (see box) to produce a short delay (the C register is loaded with a timing count of 80). The loop executes 256 times and then returns, turning on the interrupts.

Routine: Flip the Sound Bit

Purpose: To change the sound switch

Entry Point: 7676h = 30,326d

Input: None

Output: To the sound system

BASIC Example:

```
CALL 30326
```

Special Comments: None

Routine: Sound Delay

Purpose: To produce a delay

Entry Point: 7657h = 30,295d

Input: Upon entry, the C register contains a delay count.

Output: The routine returns about 5.7*C + 10.2 microseconds after it has been called, causing a delay of that length.

BASIC Example: Not applicable

Special Comments: None

Note that this method of producing a tone does not use the timer.

Bit flipping, which rapidly opens and closes the sound circuit, provides an alternate method of generating sounds. Here is a BASIC program that illustrates this bit-flipping method. The result sounds almost like a machine gun. The slightly irregular pattern occurs because the background task has not been turned off. You can gain better control of the sound system if you write in machine language. Starting with the ideas in this BASIC program, you can develop machine-language programs to produce much more interesting and sophisticated sounds.

```
100 ' FLIP THE BITS
110 '
120   FOR I = 0 TO 20
130    FOR J=1 TO 50:CALL 30326:NEXT
140    FOR J=1 TO 10:NEXT
150   NEXT I
```

The program consists of a nested loop structure. The large loop extends over lines 120-150. This loop produces twenty pulses of sound. Each pulse is generated by line 130, which calls the bit-flipping routine fifty times. A short pause between the pulses is generated by line 140.

The SOUND Command

Now let's look at the SOUND command. The code for this command begins at 1DC5h = 7621d (see box). Here the commands SOUND ON and SOUND OFF are checked for. If neither is indicated, the frequency and length parameters are loaded, and the routine jumps to location 72C5h = 29,381d (see box). At this location you can find a routine called MUSIC.

Routine: SOUND

Purpose: To control the sound or make a note (depending upon the syntax)

Entry Point: 1DC5h = 7621d

Input: Upon entry, the HL register pair points to the end of the tokenized SOUND command line (right after the token for SOUND).

Output: Depending upon the syntax of the command line, one of the following BASIC commands is executed: SOUND, SOUND ON, or SOUND OFF.

BASIC Example:

```
CALL 29381,0,H
```

where H is the address of the end of the tokenized command line for a SOUND command.

Special Comments: None

Routine: MUSIC

Purpose: To play a note of given frequency and duration

Entry Point: 72C5h = 29,381d

Input: Upon entry, the DE register pair contains a pitch number as described on page 180 of the Model 100 owner's manual, and the B register contains the duration in approximately 1/50ths of a second.

Output: To the sound system

BASIC Example: Not applicable

Special Comments: None

The MUSIC routine expects the frequency divider in the DE register pair and the duration in the B register. The routine first disables interrupts with the DI instruction. This is needed because of the timing loop. The contents of the DE register pair are then loaded into ports BCh = 188d and BDh = 189d to set the timer, with the E register going to BCh = 188d and the D register (ORed with 40h = 64d) going to BDh = 189d. This is much

like lines 150-160 of our "Make a Sound" BASIC program. Also as in our BASIC program, the value C3h = 195d is sent to port B8h = 184d to start the timer. Next, bit 5 is set and bit 2 is cleared in port BAh = 186d, again as in the BASIC program. A break check is made by calling the routine at 729Fh = 29,343d (see box). Then a timing loop at 72EAh = 29,418d counts the specified delay. Finally, the tone is turned off by setting bit 2 of port BAh = 186d and resetting the baud rate into the timer.

Summary

In this chapter we have shown how to program the sound circuits of the Model 100, including two BASIC example programs. We have also shown how the BEEP and SOUND commands are implemented in ROM routines.

9

Hidden Powers of the Cassette

Concepts
- How the cassette interface works
- The SIM instruction and the SOD line
- The RIM instruction and the SID line

ROM Routines for the Cassette System
- Motor control
- Read routines
- Write routines

The tape cassette interface provides an inexpensive way to save and retrieve programs and other kinds of files on the Model 100. In this chapter we will explore the powers of the cassette interface. We will show how to turn the cassette motor on and off through bits on ports of the 8155 PIO chip. We will also explain how to read and write data to the cassette.

How the Cassette Interface Works

The hardware that interfaces the Model 100 to a tape cassette player has three major components: motor control, writing data, and reading data.

The motor control circuit consists of a relay driven by an amplifying transistor, which in turn is controlled by bit 3 of port E8h = 232d (see Figure 9-1). A value of 0 in this bit turns off the cassette motor, and a value of 1 turns it on.

The circuit for writing data to the cassette player converts two-level digital information from the SOD (Serial Out Data) pin on the 8085 CPU

to a waveform suitable for recording on cassette tape (see Figure 9-2). This signal is sent out the cassette connector to the AUX input of the cassette recorder.

The SOD pin is activated by the SIM instruction of the 8085 CPU. SIM is a dual-purpose instruction. In addition to controlling the SOD pin, it is used to set interrupts 7.5, 6.5, and 5.5 (see Chapter 3). Before the SIM instruction is used, the accumulator must be loaded with a bit pattern. If bit 3 is set, the instruction is used to control interrupts; if bit 6 is set, it is used to control the SOD line, sending the value of bit 7 there.

The circuit for reading data from the cassette player converts the signal from the cassette player back into two-level digital information for input into the SID (Serial In Data) pin of the 8085 CPU (see Figure 9-2).

The SID pin is monitored by the CPU's RIM instruction. After this instruction is executed, the value of the SID line is found in bit 7 of the accumulator.

In the next section we will explore the routines that the Model 100 uses to write and read data bytes serially through these circuits.

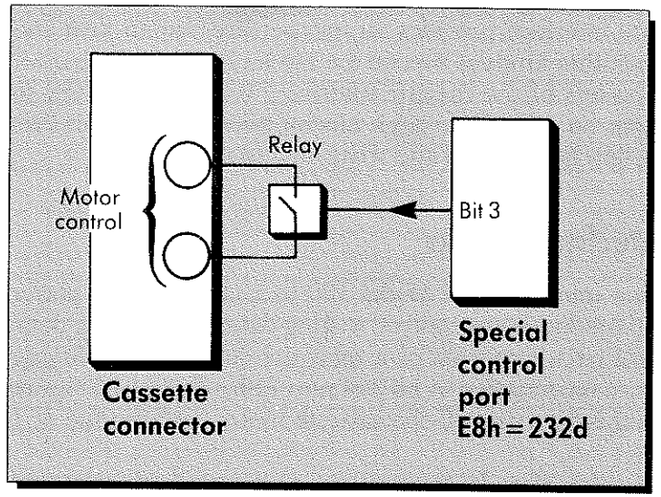

Figure 9-1. Motor control circuit

The ROM Routines for the Cassette System

Let's start with motor control. Then we'll move on to the read and write routines.

Turning the Cassette Motor On and Off

The cassette motor circuit is normally used to do what its name implies, namely, to turn the cassette motor on and off while reading or writing data

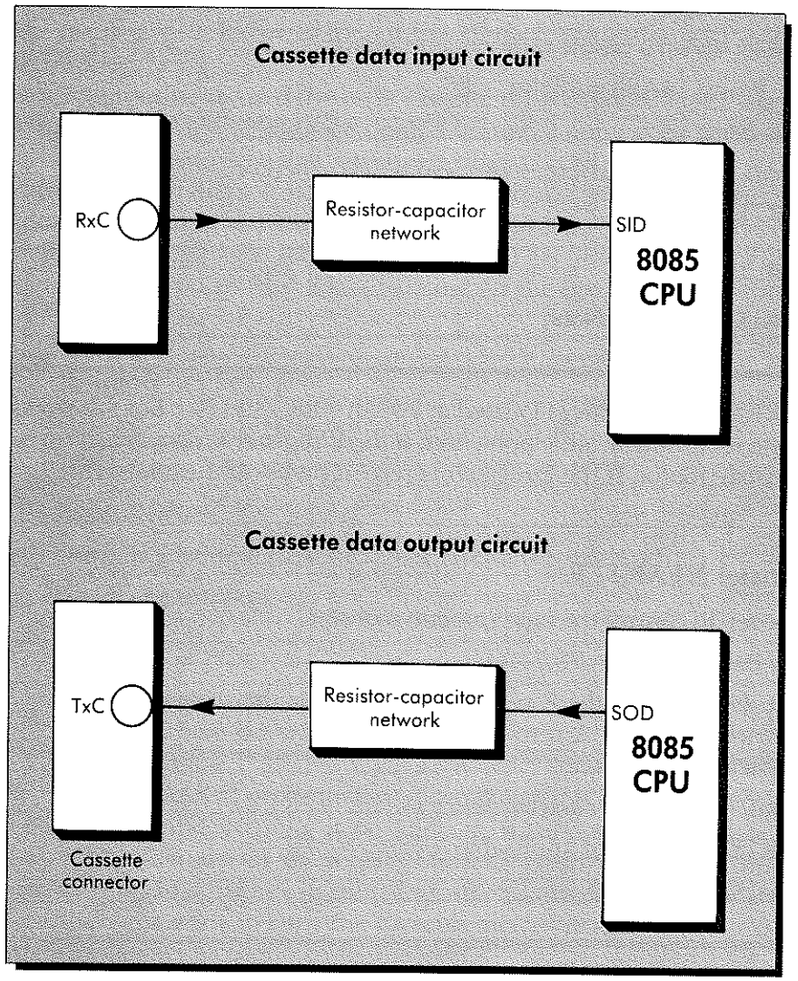

Figure 9-2. Cassette data output and input circuits

to the cassette player/recorder. However, this motor circuit can be used to control other devices as well, such as relays that turn on and off appliances or lab equipment.

The routine to turn on the tape cassette motor is called CTON and is located at 14A8h = 5288d (see box). This routine disables interrupts using the DI instruction, loads a nonzero value into the E register, and then jumps to a routine called REMOTE, which does the action.

Routine: CTON

Purpose: To turn on the tape cassette motor

Entry Point: 14A8h = 5288d

Input: None

Output: The tape cassette motor circuit is turned on.

BASIC Example:

```
CALL 5288
```

Special Comments: None

The REMOTE routine is located at 7043h = 28,739d (see box). It turns the cassette motor on or off. Upon entry, if the E register is nonzero, it turns on the cassette motor. If the E register is zero, it turns off the cassette motor.

Routine: REMOTE

Purpose: To control the tape cassette motor

Entry Point: 7043h = 28,739d

Input: Upon entry, the E register indicates whether the motor is to be turned on or off. A zero value in E indicates off, and a nonzero value in E indicates on.

Output: To the tape cassette motor circuit

BASIC Example: Not applicable

Special Comments: None

The REMOTE routine uses bit 3 of port E8h = 232d to control the cassette motor. Other bits in this port control other devices such as the optional ROM (bit 0), the printer (bit 1), and the clock (bit 2). To make sure that the REMOTE routine changes only this one bit, a copy of the contents of port E8h = 232d is maintained in memory at location FF45h = 65,349d. The REMOTE routine first reads the contents of this location into the accumulator and then clears bit 3 of the accumulator. Next, it checks the E register. If register E is nonzero (as it is coming from the CTON routine), the routine ORs the accumulator with 8, setting bit 3. If register E is zero, it leaves the accumulator as it was, with bit 3 clear. In either case, it sends the result to port E8h = 232d and also stores it in location FF45h = 65,349d before returning.

The routine to turn the tape cassette off is called CTOFF and starts at location 14AAh = 5290d (see box). This entry point is right in the middle of a CPU instruction that belongs to the CTON command, so the code differs, even though it is shared by the two routines. This time it enables interrupts using the EI instruction and then loads zero into the E register before jumping to the REMOTE routine. This time the REMOTE routine clears bit 3 of port E8h = 232d, turning off the cassette motor.

Routine: CTOFF

Purpose: To turn the tape cassette motor off

Entry Point: 14AAh = 5290d

Input: None

Output: The tape cassette motor circuit is turned off.

BASIC Example:

```
CALL 5290
```

Special Comments: None

Here is a BASIC program that directly controls bit 3 of memory location FF45h = 65,349d, turning it on and off. BASIC sends this byte out port E8h = 232d as part of its normal housekeeping; thus we do not have to explicitly do so ourselves in this program. You should note that BASIC has built-in commands to turn the cassette motor on and off; so it is not normally necessary to get down to this level.

```
100 ' CASSETTE MOTOR CONTROL
110 '
120   INPUT "BIT VALUE FOR MOTOR";X
130   Y = PEEK(65349)
140   Y = (Y AND 247) OR 8*(X AND 1)
150   POKE 65349,Y
160   GOTO 120
```

On line 120 of this program, we get the bit value (0 or 1) for the motor control bit. On line 130, we get the contents of memory location FF45h = 65,349d. This is the Model 100's record of port E8h = 232d. On line 140, we insert the new bit value in the byte, and on line 150, we put it back into location FF45h = 65,349d. On line 160, we loop back for another bit value.

Writing to the Cassette

The routines to write to the cassette are at several levels and involve two kinds of activity: 1) sending ordinary data bytes and 2) sending special header bytes.

Ordinary Data

Let's start with the first case, sending ordinary data bytes to the cassette recorder. The lowest-level routine for doing this is called DATAW and is located at 6F5Bh = 28,507d (see box).

Routine: DATAW

Purpose: To send a byte to the cassette recorder

Entry Point: 6F5Bh = 28,507d

Input: Upon entry, the A register contains the byte to be written.

Output: The byte is sent to the tape cassette recorder.

BASIC Example:

```
CALL 28507,A
```

where A contains the data byte that is to be sent to the tape cassette recorder.

Special Comments: None

The DATAW routine expects the outbound byte in the A register. If the BREAK key was hit during the routine, it returns with the carry set; otherwise it returns with the data byte sent and the carry flag clear.

For each outbound byte, the DATAW routine sends nine cycles of an electrical signal to the cassette recorder. The first cycle is a synchronizing signal corresponding to a bit value of zero, and each of the remaining eight cycles corresponds to one of the eight bits in the outbound byte. For each bit, a value of zero is sent as a single cycle that lasts about 837 microseconds, and a value of one is sent as a single cycle that lasts about 418 microseconds. Notice that the cycle for a value of 1 is just about one half the length of the cycle for the value 0. The corresponding frequencies are about 1,195 cycles per second for 0 and about 2,391 cycles per second for 1.

The individual bits are sent via a routine located at 6F6Ah = 28,522d (see box). This routine rolls the accumulator left one position, bringing what was the leftmost bit into the carry. The carry then contains the next bit to be sent. If the carry is clear, a value of 4349h = 17,225d is loaded into the DE register pair; otherwise, a value of 1F24h = 7972d is left in DE. These values control timing loops governing how long the SOD line is kept low and how long it is kept high. In other words, these values control the shape of the waveform that is sent through the SOD line.

Routine: Write Cassette Data Bit

Purpose: To send an individual bit to the tape cassette recorder

Entry Point: 6F6Ah = 28,522d

Input: Upon entry, bit 0 of the A register contains the bit to be sent to the tape cassette recorder.

Output: The bit is sent to the tape cassette recorder.

BASIC Example:

```
CALL 28522,A
```

where A contains the value of the A register.

Special Comments: None

Let's look at this routine in more detail. During the first timing loop (controlled by the D register), the SOD signal is assumed to be low. Right after this loop, the SIM instruction is used to raise the SOD line to a high value of 1. Next, the second timing loop (controlled by the E register) delays

while the SOD line retains the value 1. Then the SIM instruction is used to bring the SOD line low again. Finally, the routine jumps to the BREAK check routine at 729Fh = 29,343d (see Chapter 6) for its return.

The timing values put into the D and E registers have been carefully chosen to account for delays caused not only by the timing loops themselves but by the surrounding code. We found that for a bit value of 0, the SOD line was held low for 1027 CPU clock cycles and high for 1030 CPU clock cycles. For a bit value of 1, the SOD line was held low for 516 cycles and high for 512 cycles. Although the high/low times vary somewhat irregularly, the total number of cycles for a bit value of 0 is 2057, which is just about twice the total of 1028 cycles that we found for a bit value of 1. Since the CPU's clock runs at 2,457,600 cycles per second, you can compute how long each cycle is. As mentioned before, the cycle times are about 837 and 418 microseconds, respectively.

The CSOUT routine is the next higher level routine for sending data bytes to the cassette recorder. It is located at 14C1h = 5313d (see box). In addition to sending out bytes to the cassette, it manages a special error checking byte called a *checksum*.

Routine: CSOUT

Purpose: To send a data byte to the tape cassette recorder and update the checksum

Entry Point: 14C1h = 5313d

Input: Upon entry, the data byte is in the A register, and the current checksum is in the C register.

Output: The data byte is sent to the tape cassette recorder and the checksum is updated.

BASIC Example: Not applicable

Special Comments: None

Let's see how the checksum works. Bytes are sent to the cassette in blocks. For a BASIC program saved in regular non-ASCII form, the whole program is sent as one block. For ASCII files, however, the bytes are packaged in 256-byte blocks. The checksum is computed for each block as the lower eight bits of the sum of all the bytes in the block. This is not as complicated as it sounds, because the sum is computed using an eight-bit register; thus, only the lower eight bits of the answer are retained.

The negative of the checksum byte is placed at the end of the block. When the block is read, all the bytes of the block, including the byte from the checksum, are summed into a byte. The result should be zero; if it is not, an error must have occurred. If the result is zero, there is only a small chance that there is an error.

The CSOUT routine expects the outbound byte in the A register and the previous value of the checksum in the C register. It returns with the updated checksum in the C register.

The CSOUT routine pushes both the DE and HL register pairs on the stack. It then adds the value of the outbound byte to the the previous value of the checksum in the C register, and the result is placed back in the C register. Next, the DATAW routine is called to put the byte out to the cassette. If upon return from this routine the carry is set, the cassette motor is turned off by calling CTOFF and an IO error is declared. This happens only if a BREAK is detected. If the carry is clear, the CSOUT routine returns, POPping register pairs so that the BC, DE, and HL registers are all preserved.

It is technically possible to write BASIC programs that send bytes to the cassette recorder by calling the DATAW routine. However, BASIC is too slow to attain the close timing of bytes that would be required for recording actual data.

Synchronizing Header

Since bits are recorded serially on tape, there has to be a way for the cassette-reading logic to find and lock onto the first and then the subsequent bits of each byte. Some computers use UARTs and modems to create synchronizing information for recording each byte serially on tape. However, the Model 100 uses much simpler hardware and synchronizes entire blocks of bytes.

Each block of data begins with a series of special bit patterns that align the timing of the Model 100's cassette tape reading software so that it hits the right place in the code to accept the first bit of the first byte of each block of data. These synchronizing bit patterns make up the header bytes.

Let's look at the routine to write the special header bytes. This routine is called SYNCW, and it is located at 6F46h = 28,486d (see box). It consists of a loop that puts out 512 bytes with the value 55h = 85d and one byte with the value 7Fh = 127d. It calls a routine at 6F5Eh = 28,510d to send out the bytes (see box). This routine is essentially the DATAW routine without the first synch bit. The effect is a rapidly alternating pattern of zero and one bit values followed by a byte value of 7Fh = 127d.

Routine: SYNCW

Purpose: To write the synchronizing header to the tape cassette recorder

Entry Point: 6F46h = 28,486d

Input: None

Output: The synchronizing header is written to the tape cassette recorder.

BASIC Example:

```
CALL 28486
```

Special Comments: None

The actions of turning on the cassette motor and sending the special header are combined in one routine located at 148Ah = 5258d (see box). This routine first calls the CTON routine and then calls the SYNCW routine. If the SYNCW routine returns with an error (carry set because of a BREAK during DATAW), it calls the error routine at 45Dh = 1117d with the code for "IO error" in the E register. If there was no error, the routine simply returns.

Routine: Start Cassette Write

Purpose: To start writing to the tape cassette recorder

Entry Point: 148Ah = 5258d

Input: None

Output: The tape cassette motor is turned on, and the synchronizing header is written to the tape cassette recorder.

BASIC Example:

```
CALL 5258
```

Special Comments: None

Reading from the Cassette

The routines to read data from the cassette, like those for writing, consist of those for ordinary data bytes and those for the synchronizing header.

Ordinary Data

Let's start with the ordinary data bytes. The DATAR routine, with entry point at 702Ah = 28,714d, is at the lowest level (see box). It returns the incoming byte in the D register. If the BREAK key was hit, DATAR returns with the carry set; otherwise, it returns with the carry clear.

Routine: DATAR

Purpose: To read a byte from the tape cassette player

Entry Point: 702Ah = 28,714d

Input: None

Output: When the routine returns, the D register contains the data byte from the tape cassette player.

BASIC Example: Not applicable

Special Comments: None

The DATAR routine consists of two loops. The first loop waits for the synch bit, and the second loop picks up the eight data bits.

The DATAR routine calls a routine at 6FDBh = 28,635d to pick up the individual bits from the cassette player (see box). This bit-reading routine returns a count in the C register that measures the length of an incoming square-wave pulse. A count of 21 or more indicates a 0 bit value, and a count of less than 21 indicates a 1 bit value.

Routine: Read Cassette Data Bit

Purpose: To read a bit from the tape cassette player

Entry Point: 6FDBh = 28,635d

Input: None

Output: When the routine returns, the C register contains a number that measures the length of an incoming square-wave pulse. A count of 21 or more indicates a 0 bit value, and a count of less than 21 indicates a 1 bit value.

BASIC Example: Not applicable

Special Comments: None

The bit-reading routine has two major parts: one part counts pulses that go low-high-low, and the other part counts pulses that go high-low-high. The DATAR routine uses only the first part. In this part, there are two loops. The first loop waits for a high on the SID line, and the second loop measures how long the SID line stays high.

The first loop of the bit-reading routine looks for a BREAK by calling the BREAK check routine at 729Fh = 29,343d and monitors the SID line by executing the RIM instruction. The RIM instruction leaves the value of the SID signal line in bit 7 of the accumulator. As soon as the SID line goes high, the routine leaves its first loop and starts the second loop.

The second loop of the bit-reading routine counts the number of times that it loops while the SID line is still high. Each execution of the loop takes about 29 CPU clock cycles; thus, a count of about 17 corresponds to a 1 bit, and a count of about 35 corresponds to a 0 bit. If the count gets as high as 256, it starts all over again with the first loop. If not, it returns with the count, calling a routine at 7676h = 30,326d to flip the sound bit to make a click for you to hear (see Chapter 8). If the SOUND OFF command is currently in force, it skips this bit flip. Location FF44h = 65,348d is used to control the SOUND ON/OFF feature during cassette operation.

After the bit-reading routine, the DATAR routine calls a routine at 7023h = 28,707d to check the count and pack the bits, one at a time, into the D register (see box). This routine checks the count in the C register, using the CPI instruction to compare the count against the value of 21. This places the correct bit value into the carry. It then rotates this bit value from the carry into the D register via the accumulator.

Routine: Pack Cassette Data Bit

Purpose: To pack serial bits from the cassette into the D register

Entry Point: 7023h = 28,707d

Input: Upon entry, the C register contains the count from the bit-reading routine, and the D register contains the partially packed data byte.

Output: When the routine returns, the bit is placed into bit 0 of the data byte, and the previous contents are shifted left by one position.

BASIC Example: Not applicable

Special Comments: None

The CASIN routine at 14B0h = 5296d is the next higher level routine for reading data bytes from the cassette recorder (see box). As it reads data bytes from the cassette, it computes the checksum byte.

Routine: CASIN

Purpose: To read a data byte from the tape cassette player and update the checksum

Entry Point: 14B0h = 5296d

Input: Upon entry, the C register contains the current checksum.

Output: When the routine returns, the data byte is in the A register, and the updated checksum is in the C register.

BASIC Example: Not applicable

Special Comments: None

Upon entry, the CASIN routine expects the current value of the checksum byte in the C register. It returns with the updated checksum in the C register and the data byte in the A register.

The CASIN routine saves the DE, HL, and BC register pairs on the stack. It calls the DATAR routine to read the data byte. If a BREAK is detected, DATAR returns with the carry set, and CASIN jumps to declare an IO error. If there was no BREAK, CASIN continues, adding the value of the data byte to the current checksum and placing the result back into C.

Synchronizing Header

The routine to read the special synchronizing header is called SYNCR and is located at 6F85h = 28,549d (see box). It is designed to wait for this special header signal.

Routine: SYNCR

Purpose: To read the synchronizing header from the tape cassette player

Entry Point: 6F85h = 28,549d

Input: From the cassette player

Output: None

BASIC Example:

```
CALL 28549
```

Special Comments: None

The SYNCR routine is more complicated and tricky than the corresponding SYNCW routine, which was used to generate the header. The SYNCR routine consists of three loops. The first two lock into the alternating pattern of 0 and 1 bits in the body of the header, and the last loop checks for the last byte of the header, which is 7Fh = 127d.

The actions of turning on the cassette motor and detecting the special header are combined in one routine, located at 1499h = 5273d (see box). This routine calls the CTON routine, waits for almost a second to let the tape get up to speed, and then jumps to the SYNCR routine to look for the header.

Routine: Start Cassette Read

Purpose: To turn on the tape cassette motor and wait for the end of the synchronizing header from the cassette player

Entry Point: 1499h = 5273d

Input: From the cassette player

Output: None

BASIC Example:

```
CALL 5273
```

Special Comments: None

Summary

In this chapter we have examined the operation of the cassette tape interface on the Model 100. We have seen how the cassette motor is controlled through bit 3 of port E8h = 232d and how the data lines to and from the cassette are connected to the serial data lines of the 8085 CPU. We have discussed how to control the motor circuit for other purposes than simply controlling a cassette player/recorder. We have also studied the low-level ROM routines for reading from and writing to the cassette player/recorder.

A

BASIC Function Addresses

Address 40h = 64d

Address	Function
3407h = 13,319d	SGN
3654h = 13,908d	INT
33F2h = 13,298d	ABS
2B4Ch = 11,084d	FRE
1100h = 4,352d	INP
10C8h = 4,296d	LPOS
10CEh = 4,302d	POS
305Ah = 12,378d	SQR
313Eh = 12,606d	RND
2FCFh = 12,239d	LOG
30A4h = 12,452d	EXP
2EEFh = 12,015d	COS
2F09h = 12,041d	SIN
2F58h = 12,120d	TAN
2F71h = 12,145d	ATN
1284h = 4,740d	PEEK
1889h = 6,281d	EOF
506Dh = 20,589d	LOC
506Bh = 20,587d	LOF
3501h = 13,569d	CINT
352Ah = 13,610d	CSNG
35BAh = 13,754d	CDBL
3645h = 13,893d	FIX
2943h = 10,563d	LEN
273Ah = 10,042d	STR$
2A07h = 10,759d	VAL
294Fh = 10,575d	ASC
295Fh = 10,591d	CHR$
298Eh = 10,638d	SPACE$
29ABh = 10,667d	LEFT$
29DCh = 10,716d	RIGHT$
29E6h = 10,726d	MID$

B

BASIC Keywords

Address 80h = 128d

Keyword	Token Value
END	80h = 128d
FOR	81h = 129d
NEXT	82h = 130d
DATA	83h = 131d
INPUT	84h = 132d
DIM	85h = 133d
READ	86h = 134d
LET	87h = 135d
GOTO	88h = 136d
RUN	89h = 137d
IF	8Ah = 138d
RESTORE	8Bh = 139d
GOSUB	8Ch = 140d
RETURN	8Dh = 141d
REM	8Eh = 142d
STOP	8Fh = 143d
WIDTH	90h = 144d
ELSE	91h = 145d
LINE	92h = 146d
EDIT	93h = 147d
ERROR	94h = 148d
RESUME	95h = 149d
OUT	96h = 150d
ON	97h = 151d
DSKO$	98h = 152d
OPEN	99h = 153d
CLOSE	9Ah = 154d
LOAD	9Bh = 155d
MERGE	9Ch = 156d
FILES	9Dh = 157d
SAVE	9Eh = 158d
LFILES	9Fh = 159d

LPRINT	A0h = 160d
DEF	A1h = 161d
POKE	A2h = 162d
PRINT	A3h = 163d
CONT	A4h = 164d
LIST	A5h = 165d
LLIST	A6h = 166d
CLEAR	A7h = 167d
CLOAD	A8h = 168d
CSAVE	A9h = 169d
TIME$	AAh = 170d
DATE$	ABh = 171d
DAY$	ACh = 172d
COM	ADh = 173d
MDM	AEh = 174d
KEY	AFh = 175d
CLS	B0h = 176d
BEEP	B1h = 177d
SOUND	B2h = 178d
LCOPY	B3h = 179d
PSET	B4h = 180d
PRESET	B5h = 181d
MOTOR	B6h = 182d
MAX	B7h = 183d
POWER	B8h = 184d
CALL	B9h = 185d
MENU	BAh = 186d
IPL	BBh = 187d
NAME	BCh = 188d
KILL	BDh = 189d
SCREEN	BEh = 190d
NEW	BFh = 191d
TAB(	C0h = 192d
TO	C1h = 193d
USING	C2h = 194d
VARPTR	C3h = 195d
ERL	C4h = 196d
ERR	C5h = 197d
STRING$	C6h = 198d
INSTR	C7h = 199d
DSKI$	C8h = 200d
INKEY$	C9h = 201d

CSRLIN	CAh = 202d
OFF	CBh = 203d
HIMEM	CCh = 204d
THEN	CDh = 205d
NOT	CEh = 206d
STEP	CFh = 207d
+	D0h = 208d
-	D1h = 209d
*	D2h = 210d
/	D3h = 211d
^	D4h = 212d
AND	D5h = 213d
OR	D6h = 214d
XOR	D7h = 215d
EQV	D8h = 216d
IMP	D9h = 217d
MOD	DAh = 218d
\	DBh = 219d
>	DCh = 220d
=	DDh = 221d
<	DEh = 222d
SGN	DFh = 223d
INT	E0h = 224d
ABS	E1h = 225d
FRE	E2h = 226d
INP	E3h = 227d
LPOS	E4h = 228d
POS	E5h = 229d
SQR	E6h = 230d
RND	E7h = 231d
LOG	E8h = 232d
EXP	E9h = 233d
COS	EAh = 234d
SIN	EBh = 235d
TAN	ECh = 236d
ATN	EDh = 237d
PEEK	EEh = 238d
EOF	EFh = 239d
LOC	F0h = 240d
LOF	F1h = 241d
CINT	F2h = 242d
CSNG	F3h = 243d

CDBL	F4h = 244d
FIX	F5h = 245d
LEN	F6h = 246d
STR$	F7h = 247d
VAL	F8h = 248d
ASC	F9h = 249d
CHR$	FAh = 250d
SPACE$	FBh = 251d
LEFT$	FCh = 252d
RIGHT$	FDh = 253d
MID$	FEh = 254d

C

BASIC Command Addresses

Address 262h = 610d

Address	Command
409Fh = 16,543d	END
0726h = 1,830d	FOR
4174h = 16,756d	NEXT
099Eh = 2,462d	DATA
0CA3h = 3,235d	INPUT
478Bh = 18,315d	DIM
0CD9h = 3,289d	READ
09C3h = 2,499d	LET
0936h = 2,358d	GOTO
090Fh = 2,319d	RUN
0B1Ah = 2,842d	IF
407Fh = 16,511d	RESTORE
091Eh = 2,334d	GOSUB
0966h = 2,406d	RETURN
09A0h = 2,464d	REM
409Ah = 16,538d	STOP
1DC3h = 7,619d	WIDTH
09A0h = 2,464d	ELSE
0C45h = 3,141d	LINE
5E51h = 24,145d	EDIT
0B0Fh = 2,831d	ERROR
0AB0h = 2,736d	RESUME
110Ch = 4,364d	OUT
0A2Fh = 2,607d	ON
5071h = 20,593d	DSKO$
4CCBh = 19,659d	OPEN
4E28h = 20,008d	CLOSE

4D70h = 19,824d	LOAD
4D71h = 19,825d	MERGE
1F3Ah = 7,994d	FILES
4DCFh = 19,919d	SAVE
506Fh = 20,591d	LFILES
0B4Eh = 2,894d	LPRINT
0872h = 2,162d	DEF
128Bh = 4,747d	POKE
0B56h = 2,902d	PRINT
40DAh = 16,602d	CONT
1140h = 4,416d	LIST
113Bh = 4,411d	LLIST
40F9h = 16,633d	CLEAR
2377h = 9,079d	CLOAD
2280h = 8,832d	CSAVE
19ABh = 6,571d	TIME$
19BDh = 6,589d	DATE$
19F1h = 6,641d	DAY$
1A9Eh = 6,814d	COM
1A9Eh = 6,814d	MDM
1BB8h = 7,096d	KEY
4231h = 16,945d	CLS
4229h = 16,937d	BEEP
1DC5h = 7,621d	SOUND
1E5Eh = 7,774d	LCOPY
1C57h = 7,255d	PSET
1C66h = 7,270d	PRESET
1DECh = 7,660d	MOTOR
7F0Bh = 32,523d	MAX
1419h = 5,145d	POWER
1DFAh = 7,674d	CALL
5797h = 22,423d	MENU
1A78h = 6,776d	IPL
2037h = 8,247d	NAME
1F91h = 8,081d	KILL
1E22h = 7,714d	SCREEN
20FEh = 8,446d	NEW

D

Operator Priorities for Binary Operations

Address 2E2h = 738d

Priority Number	Operation
79h = 121d	+
79h = 121d	−
7Ch = 124d	*
7Ch = 124d	/
7Fh = 127d	^
50h = 80d	AND
46h = 70d	OR
3Ch = 60d	XOR
32h = 50d	EQV
28h = 40d	IMP
7Ah = 122d	MOD
7Bh = 123d	\

E

Some Numerical Conversion Routines

Address 2EEh = 750d

Address	Operation
35BAh = 13,754d	CDBL (Convert to Double Precision)
0000h = 0d	none
3501h = 13,569d	CINT (Convert to Integer)
35D9h = 13,785d	check for integer type
352Ah = 13,610d	CSNG (Convert to Single Precision)

F

Binary Operations for Double Precision

Address 2F8h = 760d

Address	Operation
2B78h = 11,128d	+
2B69h = 11,113d	–
2CFFh = 11,518d	*
2DC7h = 11,719d	/
3D8Eh = 15,758d	^
34FAh = 13,562d	comparisons

G

Binary Operations for Single Precision

Address 304h = 772d

Address	Operation
37F4h = 14,324d	+
37FDh = 14,333d	–
3803h = 14,339d	*
380Eh = 14,350d	/
3D7Fh = 15,743d	^
3498h = 13,464d	comparisons

H

Binary Operations for Integers

Address 310h = 784d

Address	Operation
3704h = 14,084d	+
36F8h = 14,072d	–
3725h = 14,117d	*
0F0Dh = 3,853d	/
3DF7h = 15,863d	^
34C2h = 13,506d	comparisons

I

Error Codes

Address 31Ch = 796d

Symbol	Code	Explanation
NF	1	NEXT WITHOUT FOR
SN	2	SYNTAX ERROR
RG	3	RETURN WITHOUT GOSUB
OD	4	OUT OF DATA
FC	5	ILLEGAL FUNCTION CALL
OV	6	OVERFLOW
OM	7	OUT OF MEMORY
UL	8	UNDEFINED LINE
BS	9	BAD SUBSCRIPT
DD	10	DOUBLE DIMENSIONED ARRAY
/0	11	DIVISION BY ZERO
ID	12	ILLEGAL DIRECT
TM	13	TYPE MISMATCH
OS	14	OUT OF STRING SPACE
LS	15	STRING TOO LONG
ST	16	STRING FORMULA TOO COMPLEX
CN	17	CAN'T CONTINUE
IO	18	ERROR
NR	19	NO RESUME
RW	20	RESUME WITHOUT ERROR
UE	21	UNDEFINED ERROR
MO	22	MISSING OPERAND
IE	50	UNDEFINED ERROR
BN	51	BAD FILE NUMBER
FF	52	FILE NOT FOUND
AO	53	ALREADY OPEN
EF	54	INPUT PAST END OF FILE
NM	55	BAD FILE NAME
DS	56	DIRECT STATEMENT IN FILE
FL	57	UNDEFINED ERROR
CF	58	FILE NOT OPEN

J

BASIC Error Routines

Address	Code	Explanation
446h = 1,094d	02h = 2d	SYNTAX ERROR
449h = 1,097d	0Bh = 11d	DIVISION BY 0
44Ch = 1,100d	01h = 1d	NEXT WITHOUT FOR
44Fh = 1,103d	0Ah = 10d	DOUBLE DIM ARRAY
452h = 1,106d	14h = 20d	RESUME WITHOUT ERROR
455h = 1,109d	06h = 6d	OVERFLOW
458h = 1,112d	16h = 22d	MISSING OPERAND
45Bh = 1,115d	0Dh = 13d	TYPE MISMATCH
504Eh = 20,558d	37h = 55d	BAD FILE NAME
5051h = 20,561d	35h = 53d	ALREADY OPEN
5054h = 20,564d	38h = 56d	DIRECT STATEMENT IN FILE
5057h = 20,567d	34h = 52d	FILE NOT FOUND
505Ah = 20,570d	3Ah = 58d	FILE NOT OPEN
505Dh = 20,573d	33h = 51d	BAD FILE NUMBER
5060h = 20,576d	32h = 50d	UNDEFINED ERROR
5063h = 20,579d	36h = 54d	INPUT PAST END OF FILE
5066h = 20,582d	39h = 57d	UNDEFINED ERROR

K

Control Characters for the Model 100

Address 438Ah = 17,290d

ASCII Code	Address of Routine	Function
07h = 7d	7662h = 30,306d	BELL
08h = 8d	4461h = 17,505d	BACKSPACE
09h = 9d	4480h = 17,536d	TAB
0Ah = 10d	4494h = 17,556d	LF
0Bh = 11d	44A8h = 17,576d	HOME
0Ch = 12d	4548h = 17,736d	FF
0Dh = 13d	44AAh = 17,578d	CR
1Bh = 27d	43B2h = 17,330d	ESC

L

Routines for Escape Sequences

Address 43B8h = 17,336d

ASCII code	Character after ESC Symbol	Address of Routine	Function
6Ah = 106d	j	4548h = 17,736d	ERASE SCREEN
45h = 69d	E	4548h = 17,736d	ERASE SCREEN
4Bh = 75d	K	4537h = 17,719d	ERASE TO END OF LINE
4Ah = 74d	J	454Eh = 17,742d	CLEAR TO END OF SCREEN
6Ch = 108d	l	4535h = 17,717d	ERASE LINE
4Ch = 76d	L	44EAh = 17,642d	INSERT BLANK LINE
4Dh = 77d	M	44C4h = 17,604d	DELETE LINE
59h = 89d	Y	43AFh = 17,327d	DIRECT CURSOR ADDRESSING
41h = 65d	A	4469h = 17,513d	CURSOR UP
42h = 66d	B	446Eh = 17,518d	CURSOR DOWN
43h = 67d	C	4453h = 17,491d	CURSOR RIGHT
44h = 68d	D	445Ch = 17,500d	CURSOR LEFT
48h = 72d	H	44A8h = 17,576d	HOME
70h = 112d	p	4431h = 17,457d	SET REVERSE CHAR
71h = 113d	q	4432h = 17,458d	TURN OFF REVERSE CHAR

50h = 80d	P	44AFh = 17,583d	TURN ON CURSOR
51h = 81d	Q	44BAh = 17,594d	TURN OFF CURSOR
54h = 84d	T	4439h = 17,465d	SET SYSTEM LINE
55h = 85d	U	4437h = 17,463d	RESET SYSTEM LINE
56h = 86d	V	443Fh = 17,471d	LOCK DISPLAY
57h = 87d	W	4440h = 17,472d	UNLOCK DISPLAY
58h = 88d	X	444Ah = 17,482d	

Special Screen Routines for the Model 100

Address	Entry Condition	Function
20h = 32d	A has ASCII code	Print a character
4222h = 16,930d	None	Print CR/LF
4229h = 16,937d	None	Beep
422Dh = 16,941d	None	Home cursor
4231h = 16,945d	None	Clear the screen
4235h = 16,949d	None	Lock system line
423Ah = 16,954d	None	Unlock system line
423Fh = 16,959d	None	Disable scrolling
4244h = 16,964d	None	Enable scrolling
4249h = 16,969d	None	Turn on cursor
424Eh = 16,974d	None	Turn off cursor
4253h = 16,979d	None	Delete line at cursor
4258h = 16,984d	None	Insert blank line
425Dh = 16,989d	None	Erase to end of line
4262h = 16,994d	None	Send ESC X
4269h = 17,001d	None	Set reverse character
426Eh = 17,006d	None	Turn off reverse char
4270h = 17,008d	A has ESC code	Send escape sequence
4277h = 17,015d	None	Send cursor to lower left corner of screen
427Ch = 17,020d	H = column (1-40) L = row (1-8)	Set cursor position
428Ah = 17,034d	None	Erase label line
42A5h = 17,061d	HL = address of function table	Set and display function table

N

LCD Data for Character Positions

Address 7551h = 30,033d

Column	Send to ports B9h = 185d and BAh = 186d	Send to port FEh = 254d
UPPER HALF OF DISPLAY:		
1	0001h	00h = 0d
2	0001h	06h = 6d
3	0001h	0Ch = 12d
4	0001h	12h = 18d
5	0001h	18h = 24d
6	0001h	1Eh = 30d
7	0001h	24h = 36d
8	0001h	2Ah = 42d
9	0001h	30h = 48d
10	0002h	04h = 4d
11	0002h	0Ah = 10d
12	0002h	10h = 16d
13	0002h	16h = 22d
14	0002h	1Ch = 28d
15	0002h	22h = 34d
16	0002h	28h = 40d
17	0002h	2Eh = 46d
18	0004h	02h = 2d
19	0004h	08h = 8d
20	0004h	0Eh = 14d
21	0004h	14h = 20d
22	0004h	1Ah = 26d
23	0004h	20h = 32d
24	0004h	26h = 38d
25	0004h	2Ch = 44d
26	0008h	00h = 0d
27	0008h	06h = 6d
28	0008h	0Ch = 12d
29	0008h	12h = 18d
30	0008h	18h = 24d
31	0008h	1Eh = 30d
32	0008h	24h = 36d
33	0008h	2Ah = 42d
34	0008h	30h = 48d
35	0010h	04h = 4d
36	0010h	0Ah = 10d
37	0010h	10h = 16d
38	0010h	16h = 22d
39	0010h	1Ch = 28d
40	0010h	22h = 34d
LOWER HALF OF DISPLAY:		
1	0020h	00h = 0d
2	0020h	06h = 6d
3	0020h	0Ch = 12d
4	0020h	12h = 18d
5	0020h	18h = 24d
6	0020h	1Eh = 30d
7	0020h	24h = 36d
8	0020h	2Ah = 42d
9	0020h	30h = 48d
10	0040h	04h = 4d
11	0040h	0Ah = 10d
12	0040h	10h = 16d
13	0040h	16h = 22d
14	0040h	1Ch = 28d
15	0040h	22h = 34d
16	0040h	28h = 40d
17	0040h	2Eh = 46d
18	0080h	02h = 2d
19	0080h	08h = 8d
20	0080h	0Eh = 14d
21	0080h	14h = 20d
22	0080h	1Ah = 26d
23	0080h	20h = 32d
24	0080h	26h = 38d

25	0080h	2Ch = 44d
26	0100h	00h = 0d
27	0100h	06h = 6d
28	0100h	0Ch = 12d
29	0100h	12h = 18d
30	0100h	18h = 24d
31	0100h	1Eh = 30d
32	0100h	24h = 36d
33	0100h	2Ah = 42d
34	0100h	30h = 48d
35	0200h	04h = 4d
36	0200h	0Ah = 10d
37	0200h	10h = 16d
38	0200h	16h = 22d
39	0200h	1Ch = 28d
40	0200h	22h = 34d

FINISHING PATTERN:

03FFh	01h = 1d

ASCII Tables for Regular Keys

Lowercase
7BF1h = 31,729d

7Ah = 122d z
78h = 120d x
63h = 99d c
76h = 118d v
62h = 98d b
6Eh = 110d n
6Dh = 109d m
6Ch = 108d l
61h = 97d a
73h = 115d s
64h = 100d d
66h = 102d f
67h = 103d g
68h = 104d h
6Ah = 106d j
6Bh = 107d k
71h = 113d q
77h = 119d w
65h = 101d e
72h = 114d r
74h = 116d t
79h = 121d y
75h = 117d u
69h = 105d i
6Fh = 111d o
70h = 112d p
5Bh = 91d [
3Bh = 59d ;
27h = 39d '

Uppercase
7C1Dh = 31,773d

5Ah = 90d (SHIFT) Z
58h = 88d (SHIFT) X
43h = 67d (SHIFT) C
56h = 86d (SHIFT) V
42h = 66d (SHIFT) B
4Eh = 78d (SHIFT) N
4Dh = 77d (SHIFT) M
4Ch = 76d (SHIFT) L
41h = 65d (SHIFT) A
53h = 83d (SHIFT) S
44h = 68d (SHIFT) D
46h = 70d (SHIFT) F
47h = 71d (SHIFT) G
48h = 72d (SHIFT) H
4Ah = 74d (SHIFT) J
4Bh = 75d (SHIFT) K
51h = 81d (SHIFT) Q
57h = 87d (SHIFT) W
45h = 69d (SHIFT) E
52h = 82d (SHIFT) R
54h = 84d (SHIFT) T
59h = 89d (SHIFT) Y
55h = 85d (SHIFT) U
49h = 73d (SHIFT) I
4Fh = 79d (SHIFT) O
50h = 80d (SHIFT) P
5Dh = 93d (SHIFT)]
3Ah = 58d (SHIFT) :
22h = 34d (SHIFT) "

2Ch = 44d ,
2Eh = 46d .
2Fh = 47d /
31h = 49d 1
32h = 50d 2
33h = 51d 3
34h = 52d 4
35h = 53d 5
36h = 54d 6
37h = 55d 7
38h = 56d 8
39h = 57d 9
30h = 48d 0
2Dh = 45d −
3Dh = 61d =

3Ch = 60d SHIFT <
3Eh = 62d SHIFT >
3Fh = 63d SHIFT ?
21h = 33d SHIFT !
40h = 64d SHIFT @
23h = 35d SHIFT #
24h = 36d SHIFT $
25h = 37d SHIFT %
5Eh = 94d SHIFT ^
26h = 38d SHIFT &
2Ah = 42d SHIFT *
28h = 40d SHIFT (
29h = 41d SHIFT)
5Fh = 95d SHIFT _
2Bh = 43d SHIFT +

Unshifted GRPH
7C49h = 31,817d

00h = 0d GRPH z
83h = 131d GRPH x
84h = 132d GRPH c
00h = 0d GRPH v
95h = 149d GRPH b
96h = 150d GRPH n
81h = 129d GRPH m
9Ah = 154d GRPH l
85h = 133d GRPH a
8Bh = 139d GRPH s
00h = 0d GRPH d
82h = 130d GRPH f
00h = 0d GRPH g
86h = 134d GRPH h
00h = 0d GRPH j
9Bh = 155d GRPH k
93h = 147d GRPH q
94h = 148d GRPH w
8Fh = 143d GRPH e
89h = 137d GRPH r
87h = 135d GRPH t
90h = 144d GRPH y
91h = 145d GRPH u

SHIFT GRPH
7C75h = 31,861d

E0h = 224d SHIFT GRPH Z
EFh = 239d SHIFT GRPH X
FFh = 255d SHIFT GRPH C
00h = 0d SHIFT GRPH V
00h = 0d SHIFT GRPH B
00h = 0d SHIFT GRPH N
F6h = 246d SHIFT GRPH M
F9h = 249d SHIFT GRPH L
EBh = 235d SHIFT GRPH A
ECh = 236d SHIFT GRPH S
EDh = 237d SHIFT GRPH D
EEh = 238d SHIFT GRPH F
FDh = 253d SHIFT GRPH G
FBh = 251d SHIFT GRPH H
F4h = 244d SHIFT GRPH J
FAh = 250d SHIFT GRPH K
E7h = 231d SHIFT GRPH Q
E8h = 232d SHIFT GRPH W
E9h = 233d SHIFT GRPH E
EAh = 234d SHIFT GRPH R
FCh = 252d SHIFT GRPH T
FEh = 254d SHIFT GRPH Y
F0h = 240d SHIFT GRPH U

8Eh = 142d GRPH i
98h = 152d GRPH o
80h = 128d GRPH p
60h = 96d GRPH [
92h = 146d GRPH ;
8Ch = 140d GRPH '
99h = 153d GRPH ,
97h = 151d GRPH .
8Ah = 138d GRPH /
88h = 136d GRPH 1
9Ch = 156d GRPH 2
9Dh = 157d GRPH 3
9Eh = 158d GRPH 4
9Fh = 159d GRPH 5
B4h = 180d GRPH 6
B0h = 176d GRPH 7
A3h = 163d GRPH 8
7Bh = 123d GRPH 9
7Dh = 125d GRPH 0
5Ch = 92d GRPH −
8Dh = 141d GRPH =

Unshifted CODE
7CA1h = 31,905d

CEh = 206d CODE z
A1h = 161d CODE x
A2h = 162d CODE c
BDh = 189d CODE v
00h = 0d CODE b
CDh = 205d CODE n
00h = 0d CODE m
CAh = 202d CODE l
B6h = 182d CODE a
A9h = 169d CODE s
BBh = 187d CODE d
00h = 0d CODE f
00h = 0d CODE g
00h = 0d CODE h
CBh = 203d CODE j
C9h = 201d CODE k
C8h = 200d CODE q

F3h = 243d SHIFT GRPH I
F2h = 242d SHIFT GRPH O
F1h = 241d SHIFT GRPH P
7Eh = 126d SHIFT GRPH]
F5h = 245d SHIFT GRPH :
00h = 0d SHIFT GRPH "
F8h = 248d SHIFT GRPH <
F7h = 247d SHIFT GRPH >
00h = 0d SHIFT GRPH ?
E1h = 225d SHIFT GRPH !
E2h = 226d SHIFT GRPH @
E3h = 227d SHIFT GRPH #
E4h = 228d SHIFT GRPH $
E5h = 229d SHIFT GRPH %
E6h = 230d SHIFT GRPH ^
00h = 0d SHIFT GRPH &
00h = 0d SHIFT GRPH *
00h = 0d SHIFT GRPH (
00h = 0d SHIFT GRPH)
7Ch = 124d SHIFT GRPH _
00h = 0d SHIFT GRPH +

SHIFT CODE
7CCDh = 31,949d

00h = 0d SHIFT CODE Z
DFh = 223d SHIFT CODE X
ABh = 171d SHIFT CODE C
DEh = 222d SHIFT CODE V
00h = 0d SHIFT CODE B
00h = 0d SHIFT CODE N
A5h = 165d SHIFT CODE M
DAh = 218d SHIFT CODE L
B1h = 177d SHIFT CODE A
B9h = 185d SHIFT CODE S
D7h = 215d SHIFT CODE D
BFh = 191d SHIFT CODE F
00h = 0d SHIFT CODE G
00h = 0d SHIFT CODE H
DBh = 219d SHIFT CODE J
D9h = 217d SHIFT CODE K
D8h = 216d SHIFT CODE Q

00h = 0d (CODE) w	00h = 0d (SHIFT) (CODE) W
C6h = 198d (CODE) e	D6h = 214d (SHIFT) (CODE) E
00h = 0d (CODE) r	AAh = 170d (SHIFT) (CODE) R
00h = 0d (CODE) t	BAh = 186d (SHIFT) (CODE) T
CCh = 204d (CODE) y	DCh = 220d (SHIFT) (CODE) Y
B8h = 184d (CODE) u	B3h = 179d (SHIFT) (CODE) U
C7h = 199d (CODE) i	D5h = 213d (SHIFT) (CODE) I
B7h = 183d (CODE) o	B2h = 178d (SHIFT) (CODE) O
ACh = 172d (CODE) p	00h = 0d (SHIFT) (CODE) P
B5h = 181d (CODE) [	00h = 0d (SHIFT) (CODE)]
ADh = 173d (CODE) ;	00h = 0d (SHIFT) (CODE) :
A0h = 160d (CODE) '	A4h = 164d (SHIFT) (CODE) "
BCh = 188d (CODE) ,	DDh = 221d (SHIFT) (CODE) <
CFh = 207d (CODE) .	00h = 0d (SHIFT) (CODE) >
AEh = 174d (CODE) /	00h = 0d (SHIFT) (CODE) ?
C0h = 192d (CODE) 1	D0h = 208d (SHIFT) (CODE) !
00h = 0d (CODE) 2	00h = 0d (SHIFT) (CODE) @
C1h = 193d (CODE) 3	D1h = 209d (SHIFT) (CODE) #
00h = 0d (CODE) 4	00h = 0d (SHIFT) (CODE) $
00h = 0d (CODE) 5	00h = 0d (SHIFT) (CODE) %
00h = 0d (CODE) 6	00h = 0d (SHIFT) (CODE) ^
C4h = 196d (CODE) 7	D4h = 212d (SHIFT) (CODE) &
C2h = 194d (CODE) 8	D2h = 210d (SHIFT) (CODE) *
C3h = 195d (CODE) 9	D3h = 211d (SHIFT) (CODE) (
AFh = 175d (CODE) 0	A6h = 166d (SHIFT) (CODE))
C5h = 197d (CODE) –	A7h = 167d (SHIFT) (CODE) _
BEh = 190d (CODE) =	A8h = 168d (SHIFT) (CODE) +

ASCII Table for NUM Key

Memory Address	Regular Key	NUM Pad Key	
7CF9h = 31,993d	6Dh = 109d	30h = 48d	0
7CFBH = 31,995d	6Ah = 106d	31h = 49d	1
7CFDh = 31,997d	6Bh = 107d	32h = 50d	2
7CFFh = 31,999d	6Ch = 108d	33h = 51d	3
7DD1h = 32,001d	75h = 117d	34h = 52d	4
7DO3h = 32,003d	69h = 105d	35h = 53d	5
7DO5h = 32,005d	6Fh = 111d	36h = 54d	6
	Byte	Byte	

ASCII Tables for Special Keys

7DO7h = 32,007d		7D1Bh = 32,027d	
01h = 1d	←	1Dh = 29d	SHIFT ←
06h = 6d	→	1Ch = 28d	SHIFT →
14h = 20d	↑	1Eh = 30d	SHIFT ↑
02h = 2d	↓	1Fh = 31d	SHIFT ↓
20h = 32d	SPACE	20h = 32d	SHIFT SPACE
7Fh = 127d	DEL	08h = 8d	SHIFT BKSP
09h = 9d	TAB	09h = 9d	SHIFT TAB
1Bh = 27d	ESC	1Bh = 27d	SHIFT ESC
8Bh = 139d	PASTE	8Bh = 139d	SHIFT PASTE
88h = 136d	LABEL	88h = 136d	SHIFT LABEL
8Ah = 138d	PRINT	89h = 137d	SHIFT PRINT
0Dh = 13d	ENTER	0Dh = 13d	SHIFT ENTER
80h = 128d	F1	80h = 128d	SHIFT F1
81h = 129d	F2	81h = 129d	SHIFT F2
82h = 130d	F3	82h = 130d	SHIFT F3
83h = 131d	F4	83h = 131d	SHIFT F4
84h = 132d	F5	84h = 132d	SHIFT F5
85h = 133d	F6	85h = 133d	SHIFT F6
86h = 134d	F7	86h = 134d	SHIFT F7
87h = 135d	F8	87h = 135d	SHIFT F8

Index