the maximum is +32767. If more precision is required, the program will have to handle longer strings of eight bits in a *multiple-precision scheme.*

Let's see how the assembler handles representation of signed numbers. The program that follows shows a data table of various types of signed numbers, eight bits (DEFB) and 16 bits (DEFW). Note how the assembler automatically computes the proper two's complement form. Might we even suggest the odious task of looking at the arguments, converting a few numbers yourself, and then checking them against the assembled value? Like chicken soup, it won't hurt!

```
                00100 ;TABLE OF CONSTANTS
                00110 ;
4000            00120       ORG    4000H
4000 00         00130       DEFB   0
4001 01         00140       DEFB   1
4002 FF         00150       DEFB   0FFH
4003 FE         00160       DEFB   0FEH
4004 7F         00170       DEFB   127
4005 0000       00180       DEFW   0
4007 FFFF       00190       DEFW   -1
4009 0100       00200       DEFW   1
400B FF7F       00210       DEFW   +32767
400D 0080       00220       DEFW   -32768
0000            00230       END
00000 TOTAL ERRORS
```

Note that the 16-bit values are in standard Z-80 representation, reversed so that the most significant byte is last and the least significant byte is first.

Adding and Subtracting 8-Bit Numbers

There are several actions that occur when two 8-bit signed numbers are added in the Z-80. First, the instruction adds the two operands and puts the result in the A register (initially, as you will recall, one of the operands was in A). In the course of adding the numbers, the carry flag, half carry flag, overflow flag, zero flag, and sign flag are all affected according to the results of the add.

The zero flag is set if the result is zero. The two instructions

```
LD    A,23     ;LOAD 23 INTO A
ADD   A,-23    ;ADD -23
```

would result in an A register result of zero and the zero flag set to a one. The carry flag is set if there is a carry out of bit position 7 after the add, and the half carry is set if there is a carry out of bit position 3. These carries are equivalent to decimal carries during an addition of two decimal numbers. The carry out of bit position 3 is the "half-carry" and is used for decimal addition of binary-coded-decimal operands discussed later on in this chapter. The "carry" out of the high-order bit position occurs whenever a carry is generated for the add, as in the add of 23 and −23.

```
        00010111     23
carry   11101001    -23  (try the two's complement)
------------------------
  1     00000000      0  (zero result)
```

The carry flag can be used for adds of multiple bytes, for adds of bcd operands, or for certain types of compares.

The sign flag is really the duplication of the sign bit in the result after the add. If the result of the add is positive, the sign flag is reset (0), while if the result is negative, the sign flag is set (1). The sign flag can then be used for conditional jumps such as jump if result positive (JP P,aaaa) or jump if result negative (JP M,aaaa).

The overflow flag is used during adds and subtracts to detect *overflow* conditions. Overflow occurs when the result of the add is too large to fit into an 8-bit signed representation. Suppose that we are adding +127 and +50. We know that the maximum positive number that can fit in 8 bits is +127. What would the result be if we actually performed the add?

```
01111111  (+127)
00110010  (+ 50)
--------  -----
10110001  (- 79)   result — wrong!
```

As the reader can see from the example, the result of −79 is incorrect. If we had no way to detect the overflow, we might go merrily on our way printing a paycheck for an employee of $1,045,067.66, or an equally catastrophic action. Fortunately, the Z-80 *does* set overflow when the result is greater than +127 or less than −128.

When a subtract instead of an add is used, all of the above actions apply. The Z-80 performs the subtract just as you

would on paper, and then sets the flags according to the results of the subtract. There are really no fundamental differences between an add and subtract, as the reader can see if he considers adding +23 and −15 and then compares it to subtracting +15 from +23.

To illustrate the settings of the flag bits after an add or subtract, let's use T-BUG to execute some examples of arithmetic operations. Load T-BUG and key in the following program. Run the following examples by using T-BUG to change the operands in 4B00H and 4B01H, breakpoint at location 4A14H and then use the M command to look at the flags and results in locations 4B02H through 4B05H as shown in Table 7-1. In addition to the examples below the reader is urged to try his own values. The flags will have to be "decoded" from an 8-bit value to determine the state of the flags (it *is* some work, but you'll be a better programmer for it). The bit positions of the flag register are shown in Figure 7-5, and in Table 7-1.

```
                00100 ;PROGRAM TO ILLUSTRATE ARITHMETIC
                00110 ;
4A00            00120          ORG     4A00H
4A00 3A014B     00130          LD      A,(4B01H)       ;GET SOURCE
4A03 47         00140          LD,     B,A             ;FOR OPERATION
4A04 3A004B     00150          LD      A,(4B00H)       ;GET DESTINATION
4A07 80         00160          ADD     A,B             ;ADD
4A08 F5         00170          PUSH    AF              ;TRANSFER FLAGS
4A09 E1         00180          POP     HL              ;GET RESULT FLAGS
4A0A 22024B     00190          LD      (4B02H),HL      ;STORE
4A0D 90         00200          SUB     B               ;RESTORE
4A0E 90         00210          SUB     B               ;SUBTRACT
4A0F F5         00220          PUSH    AF              ;TRANSFER FLAGS
4A10 E1         00230          POP     HL              ;GET RESULT, FLAGS
4A11 22044B     00240          LD      (4B04H),HL      ;STORE
4A14 C3144A     00245 LOOP     JP      LOOP            ;LOOP HERE FOR BP
0000            00250          END
00000 TOTAL ERRORS
LOOP    4A14
```

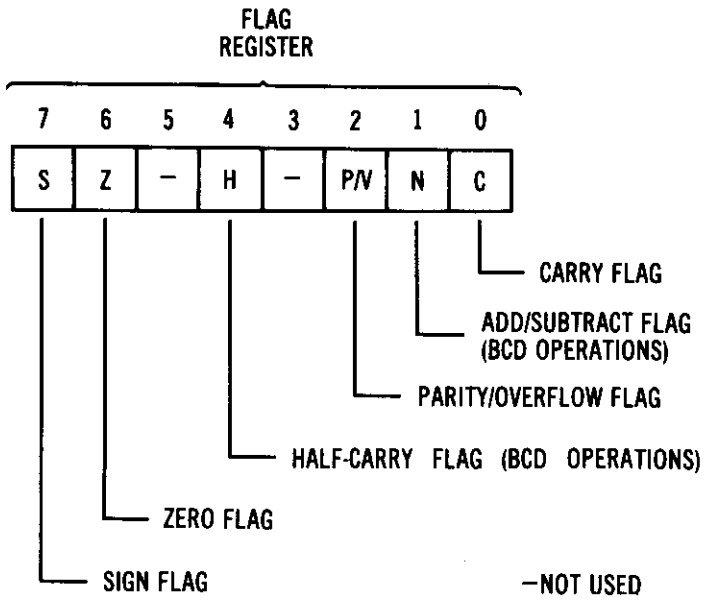


Fig. 7-5. Flag register bit positions.

Table 7-1. Examples of Add and Subtract Flag Bit

Location	Contents	Test Cases			
		1	2	3	4
4B00H	Dest Op	+33(21H)	−5(FBH)	−30(E2H)	120(78H)
4B01H	Source Op	+64(40H)	−30(E2H)	−5(FBH)	100(64H)
4B02H	Add Flags	00100000	10001001	10001001	10001100
4B03H	Add Result	+97(61H)	−35(DDH)	−35(DDH)	−24(DCH)
4B04H	Sub Flags	10100011	00001010	10110011	00000010
4B05H	Sub Result	−31(E1H)	+25(19H)	−25(E7H)	+20(14H)
FLAGS S Z - H - P/V N C 7 6 5 4 3 2 1 0					

Adding and Subtracting 16-Bit Numbers

The Z-80 allows two 16-bit operands to be added, as we found in a previous chapter. One of the operands must be in the HL, IX, or IY registers, analogous to the A register in 16-bit arithmetic; the second operand must be in one of the other register pairs. When an add or subtract is performed 16 bits at a time, the flags are affected in various ways, depending upon which of the 16-bit arithmetic instructions is being used. When an add is done to the IX register, for example, the zero and sign flags are not affected, but when an "ADC" is done with the HL register, the sign and zero flags *are* affected. When in doubt about flag action, consult the

individual flag action listed under the instruction in question in the Editor/Assembler manual.

The advantage of the 16-bit adds, of course, is that much larger numbers can be handled, at the expense of addressing versatility. Since the HL, IX, and IY registers are generally used as memory printer registers, the 16-bit adds and subtracts using these registers can be used to advantage to calculate memory addresses. As an example of this memory address computation capability, let's use the following program. This program uses 16-bit adds and subtracts to calculate memory addresses for movement of a dot across the video screen.

```
                00100 ;ROUTINE TO MOVE A DOT
                00110 ;
4000            00120           ORG     4000H           ;START
4000 21203C     00130           LD      HL,3C00H+32     ;START POSITION
4003 114000     00150           LD      DE,64           ;INCREMENT
4006 3E0F       00160           LD      A,15            ;NUMBER OF LINES
4008 010000     00170           LD      BC,0            ;DELAY COUNT
400B 36BF       00180 LOOP1     LD      (HL),0BFH       ;ALL ON
400D 05         00200 LOOP2     DEC     B               ;DELAY CNT-1
400E 20FD       00210           JR      NZ,LOOP2        ;GO IF NOT DONE
4010 0D         00220           DEC     C               ;DELAY COUNT
4011 20FA       00230           JR      NZ,LOOP2        ;GO IF NOT DONE
4013 3680       00240           LD      (HL),80H        ;ALL OFF
4015 19         00250           ADD     HL,DE           ;NEXT ROW
4016 3D         00260           DEC     A               ;#LINES-1
4017 20F2       00270           JR      NZ,LOOP1        ;CONTINUE
4019 18FE       00280 LOOP3     JR      LOOP3           ;LOOP HERE
0000            00290           END
00000 TOTAL ERRORS
LOOP3   4019
LOOP2   400D
LOOP1   400B
```

The program starts by loading HL with the first position of the dot, the screen memory plus one-half line. DE is loaded with 64, representing the number that must be added to move

the dot to the middle of the next line. A is loaded with 15, the number of lines that the dot will move. BC is loaded with a delay count of 0, representing a delay of 65536 counts when BC is decremented in the loop. The action of the loop from LOOP1 through 4A17H is this: The dot is initially set on by outputting the graphic character 0BFH. This character sets every one of the six *pixels* in the character position. Now the program delays about ½ second by means of a 4 instruction delay loop. BC has zero at the end of the loop. After the delay the pixels are turned off by outputting the graphics character 80H. Then the next address is computed by adding the 64 in DE to HL, the address pointer. The contents of A are decremented by one. If 15 lines have not been reached, the program loops back to LOOP1.

There are several interesting things in the above program. Because the assembly-language code is extremely fast, we had to delay each time a dot (actually six dots) was moved to a new position. The delay count in BC was initialized to 0, and decremented by decrementing B back to 0 again (256 loops) as an *inner loop* and by decrementing C from 0 back to 0 as an *outer loop*. The reader should realize that at 4A13H, the count in BC is 0, in preparation for the next delay loop. Another point is that there is no way to decrement BC and test for zero, as the flags are not affected by a DEC BC. Hence two decrements are used, each one checking one of the two registers for zero—a DEC B or DEC C *does* set the flags after the decrement.

To illustrate the 16-bit subtract, we'll rewrite the program above to make a single pixel move from the bottom of the screen to the top of the screen. This program will be identical to the one above except that the starting position will be 3C00H+992, the 32nd character position in line 16, the increment in DE will be −64, and the graphics codes will specify all on or all off for a single pixel (we'll be looking at the graphics codes in more detail in a later chapter).

```
                00100 ;ROUTINE TO MOVE A DOT (BACKWARDS)
                00110 ;
4A00            00120       ORG     4A00H           ;START
4A00 21E03F     00130       LD      HL,3C00H+992    ;START POSITION
4A03 114000     00140       LD      DE,64           ;INCREMENT
4A06 3E0F       00150       LD      A,15            ;NUMBER OF LINES
```

```
4000 010000    00160        LD     BC,0          ;DELAY COUNT
4003 3681      00170 LOOP1  LD     (HL),81H      ;ONE PIXEL ON
4005 05        00180 LOOP2  DEC    B             ;DELAY COUNT - 1
4006 20FD      00190        JR     NZ,LOOP2      ;GO IF NOT DONE
4008 0D        00200        DEC    C             ;DELAY COUNT
4009 20FA      00210        JR     NZ,LOOP2      ;GO IF NOT DONE
400B 3680      00220        LD     (HL),80H      ;ALL OFF
400D B7        00230        OR     A             ;RESET CARRY
400E ED52      00240        SBC    HL,DE         ;NEXT ROW
4010 3D        00250        DEC    A             ;#LINES-1
4011 20F0      00260        JR     NZ,LOOP1      ;CONTINUE
4013 18FE      00270 LOOP3  JR     LOOP3         ;LOOP HERE
0000           00280        END
00000 TOTAL ERRORS
LOOP3   4013
LOOP2   4005
LOOP1   4003
```

The subtract was performed by the SBC instruction which subtracted the increment value of 64 from the current video memory position in HL. Note that before the subtract, an OR A was done. *The only reason for performing the* OR *A was to reset the carry flag.* The two questions that may immediately come to the readers mind are why use an OR A to reset the carry, and *why* reset the carry? The OR A is used because it is a short (one byte) and fast instruction. We could have reset the carry flag by an SCF followed by a CCF (set carry, complement carry), but the OR A does not affect the contents of the A register (try ORing any value with itself) and it is efficient.

Why do we want to reset the carry before the SBC? Well the SBC is actually a Subtract with Carry type instruction that not only subtracts a second operand from the contents of HL, but also subtracts the current state of the carry. That means that one more count might be subtracted from HL if the carry is set before the subtract. Since the carry is set and reset with many instructions, we have no way of knowing whether the carry will be set or reset before the SBC, and therefore *must* clear the carry to avoid subtracting a possible one from the result.

A Precision Instrument

The reason that the carry enters into some adds and subtracts on the TRS-80 is that the Z-80, like other microprocessors, is able to handle *multiple-precision* adds and subtracts. Remember that the maximum value that can be held in 8 bits is 255 and that the maximum value that can be held in 16 bits is 65535. What happens if we want more *precision* and want to hold larger numbers for adds and subtracts? How could we add 32-bit (four byte) numbers, for example, allowing us to work with values up to 4 billion or so (2^{32})?

Larger numbers are held in multiple-precision representation, which is simply a method for representing the numbers in as many bytes as required. If we know, for example, that a billion or so is the largest number we'll be working with, we can conveniently work with four-byte numbers in the Z-80. Suppose that we wanted to add two four-byte operands of +344,050 and +500,000, as shown in Figure 7-6. The numbers

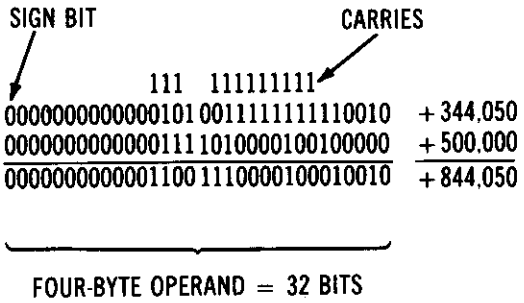


Fig. 7-6. Multiple-precision adds by manual methods.

are *signed* 32-bit operands, with the most significant bit representing the sign of plus (0) or minus (1) just as in the case of 8- or 16-bit operands. To add them with pencil and paper, we simply add the ones and zeros, and any carry from the lower bit positions as shown in the figure.

To add the numbers in the Z-80, we have a bit of a problem (32 bits of a problem, to be precise). We can add up to 16-bit operands, but how can we add 32 bits at a time? The answer is that the adds must be either four 8-bit adds or two 16-bit adds. Each of the adds must *add in* any carry from the last byte or two bytes, just as we do on pencil and paper operations. The following program adds two four-byte operands, representing the above values. The operands are in memory locations 4B00H-4B03H and 4B10H-4B13H and the result is stored in location 4B00H-4B03H. Key in the program using

T-BUG (or assemble and load), execute at 4A00H after break-pointing, and then check the result at 4B00H-4B03H. It should correspond with the result shown in Figure 7-7.

```
                00100 ;FOUR BYTE ADD ROUTINE
                00110 ;
4000            00120       ORG     4000H
4000 DD21004B   00130 START LD      IX,4B00H    ;DESTINATION
4004 FD21104B   00140       LD      IY,4B10H    ;SOURCE
4008 DD7E03     00150       LD      A,(IX+3)    ;GET BYTE 0
400B FD8603     00160       ADD     A,(IY+3)    ;ADD SOURCE
400E DD7703     00170       LD      (IX+3),A    ;STORE RESULT
4011 DD7E02     00180       LD      A,(IX+2)    ;GET BYTE 1
4014 FD8E01     00190       ADC     A,(IY+1)    ;ADD SOURCE
4017 DD7702     00200       LD      (IX+2),A    ;STORE RESULT
401A DD7E01     00210       LD      A,(IX+1)    ;GET BYTE 2
401D FD8E02     00220       ADC     A,(IY+2)    ;ADD SOURCE
4020 DD7701     00230       LD      (IX+1),A    ;STORE RESULT
4023 DD7E00     00240       LD      A,(IX)      ;GET BYTE 3
4026 FD8E00     00250       ADC     A,(IY)      ;ADD SOURCE
4029 DD7700     00260       LD      (IX),A      ;STORE RESULT
402C C32C4A     00270 LOOP  JP      LOOP        ;LOOP HERE FOR BP
4B00            00280       ORG     4B00H       ;DESTINATION AREA
4B00 00         00290       DEFB    0           ;+344,050
4B01 05         00300       DEFB    5
4B02 3F         00310       DEFB    3FH
4B03 F2         00320       DEFB    0F2H
4B10            00330       ORG     4B10H       ;SOURCE AREA
4B10 00         00340       DEFB    0
4B11 07         00350       DEFB    7
4B12 A1         00360       DEFB    0A1H
4B13 20         00370       DEFB    20H
0000            00380       END
00000 TOTAL ERRORS
LOOP    402C
START   4000
```

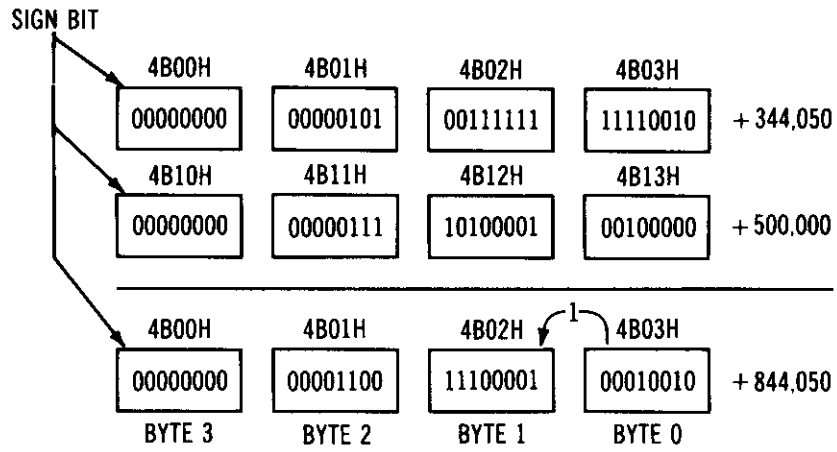


Fig. 7-7. Multiple-precision adds by machine.

The program uses indexed addressing with both IX and IY. The IX register points to the *destination* operand in 4B00H through 4B03H. Note that the most significant byte is at 4B00H and the last significant byte is at 4B03H. The IY register points to the *source* operand at 4B10H. Although the four adds could have been done in a loop, the in-line code in the program clearly shows the steps that must be taken for the adds. The first add adds (IX+3) and (IY+3), the least significant byte, and stores the result in 4B03H. After the ADD, the carry flag is set or reset dependent upon the carry from bit position 7, *which is not a sign bit, but just another bit position*. The next add (ADC) adds not only the two bytes from (IX+2) and (IY+2), but the *carry* from the previous add, which is undisturbed, as loads do not affect the carry or other flags. The next add adds in the carry from the second byte, and the last add adds in the carry from the third byte. All adds, except the first, added in a possible carry from the lower order byte. In the first add there was no preceding carry to be added in.

The program shows the general approach to add any number of bytes. There is no limit on the maximum number of bytes that could be used, but working with 32-byte operands might get somewhat tedious after a while. Floating-point format allows a more compact representation of large numbers, at the sacrifice of the number of significant digits, and is widely used in cases where very large, very small, or mixed numbers must be used.

Subtraction of multiple-precision numbers is handled in similar fashion. The first subtract would be an SUB without the carry, but the remaining three would be SBCs, which use the borrow from the preceding lower-order byte. A portion of this code is shown below.

```
LD    A,(IX+2)  ;SECOND BYTE
SBC   A,(IY+2)  ;SUBTRACT SOURCE
LD    (IX+2),A  ;STORE RESULT
```

There is no reason that 16-bit adds and subtracts couldn't be used, as long as the total number of bytes was a multiple of two. In the general case, 8-bit adds and subtracts are somewhat easier to work with, as they allow for an odd number of bytes and permit a direct add or subtract of the source operand (through HL, IX, or IY). The two programs shown below are general-purpose subroutines for multiple-precision adds and subtracts. They will handle any number of bytes required. Upon entry, IX and IY point to the first (most significant) bytes of the destination and source, respectively. The B register contains the number of bytes in the operands (both operands must have the same number of bytes). The subroutines add or subtract the source operand from the destination operand and put the result in the destination operand memory locations. Upon return from the subroutine IX and IY are unchanged and the contents of B are zero.

```
                00100 ;SUBROUTINE TO DO MULTIPLE-PRECISION ADDS
                00110 ;   ENTRY:(IX)=POINTS TO MS BYTE OF DESTINATION
                00120 ;         (IY)=POINTS TO MS BYTE OF SOURCE
                00130 ;         (B)=# OF BYTES IN OPERANDS
                00140 ;         CALL  MULADD
                00150 ;         (RETURN)
                00160 ;   EXIT:(IX)=UNCHANGED
                00170 ;         (IY)=UNCHANGED
                00180 ;         (A)=DESTROYED
                00190 ;         (B)=0
                00200 ;
4000            00210         ORG   4000H         ;CHANGE ON REASSEMBLY
4000 D5         00220 MULADD  PUSH  DE            ;SAVE DE
4001 58         00230         LD    E,B           ;#BYTES TO E
4002 1600       00240         LD    D,0           ;DE NOW HAS #
4004 1B         00250         DEC   DE            ;DE NOW HAS #-1
4005 DD19       00260         ADD   IX,DE         ;POINT TO LS BYTE
4007 FD19       00270         ADD   IY,DE         ;POINT TO LS BYTE
```

```
4009 D1       00280          POP    DE             ;RESTORE ORIGINAL
400A AF       00290          XOR    A              ;RESET CARRY
400B DD7E00   00300 LOOP     LD     A,(IX)         ;GET DESTINATION
400E FD8E00   00310          ADC    A,(IY)         ;ADD SOURCE
4011 DD7700   00320          LD     (IX),A         ;STORE RESULT
4014 1001     00330          DJNZ   LOOP1          ;GO IF NOT DONE
4016 C9       00340          RET                   ;RETURN
4017 DD2B     00350 LOOP1    DEC    IX             ;PNT TO NEXT HIGHER
4019 FD2B     00360          DEC    IY             ;PNT TO NEXT HIGHER
401B C30B4A   00370          JP     LOOP           ;CONTINUE
0000          00380          END
00000 TOTAL ERRORS
LOOP1   4017
LOOP    400B
MULSUB  4000

                00100 ;SUBROUTINE TO DO MULTIPLE-PRECISION SUBTRACTS
                00110 ;    ENTRY:(IX)=POINTS TO MS BYTE OF DESTINATION
                00120 ;          (IY)=POINTS TO MS BYTE OF SOURCE
                00130 ;          (B)=# OF BYTES IN OPERANDS
                00140 ;          CALL  MULSUB
                00150 ;          (RETURN)
                00160 ;    EXIT:(IX)=UNCHANGED
                00170 ;         (IY)=UNCHANGED
                00180 ;         (A)=DESTROYED
                00190 ;         (B)=0
                00200 ;
4000            00210          ORG    4000H          ;CHANGE ON REASSEMBLY
4000 D5       00220 MULSUB   PUSH   DE             ;SAVE DE
4001 58       00230          LD     E,B            ;#BYTES TO E
4002 1600     00240          LD     D,0            ;DE NOW HAS #
4004 1B       00250          DEC    DE             ;DE NOW HAS #-1
4005 DD19     00260          ADD    IX,DE          ;POINT TO LS BYTE
4007 FD19     00270          ADD    IY,DE          ;POINT TO LS BYTE
4009 D1       00280          POP    DE             ;RESTORE ORIGINAL
```

```
4000 AF       00290        XOR    A          ;RESET CARRY
4001 DD7E00   00300 LOOP   LD     A,(IX)     ;GET DESTINATION
4004 FD9E00   00310        SBC    A,(IY)     ;SUBTRACT SOURCE
4007 DD7700   00320        LD     (IX),A     ;STORE RESULT
400A 1001     00330        DJNZ   LOOP1      ;GO IF NOT DONE
400C C9       00340        RET               ;RETURN
400D DD2B     00350 LOOP1  DEC    IX         ;PNT TO NEXT HIGHER
400F FD2B     00360        DEC    IY         ;PNT TO NEXT HIGHER
4011 C30B40   00370        JP     LOOP       ;CONTINUE
0000          00380        END
00000 TOTAL ERRORS
LOOP1   4017
LOOP    400B
MULSUB  4000
```

Decimal Arithmetic

Up to this point we've been doing arithmetic operations with absolute and two's complement numbers. As we mentioned earlier in the chapter, there is a third type of arithmetic that is possible in the Z-80 and many other microprocessors, *binary-coded-decimal* (*bcd*) arithmetic. The bcd representation is a more direct translation from decimal than binary. To convert a decimal number into bcd, change each decimal digit into its 4-bit binary equivalent. Some exam-

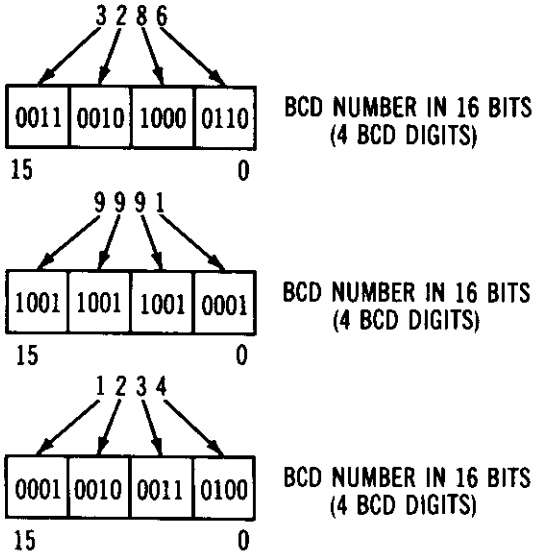


Fig. 7-8. The bcd representation.

ples of this are shown in Figure 7-8. After the conversion we're left with a *binarylike* number whose length equals four times the number of decimal digits, or to put it another way, two bcd digits in each 8-bit segment as shown in the figure.

The bcd representation is used for a variety of purposes. Much instrumentation uses bcd, especially instrumentation that displays digits in digital readout form, such as digital voltmeters and digital frequency counters. We could, of course, convert from bcd to binary, perform arithmetic operations in binary, and reconvert to bcd, but it is convenient to be able to directly add or subtract bcd values in the Z-80.

Adding or subtracting bcd is *not* the same as adding or subtracting binary numbers. Since the binary groups of 1010 through 1111 are not permitted in bcd (there is no bcd equivalent), operations in binary produce erroneous results, as

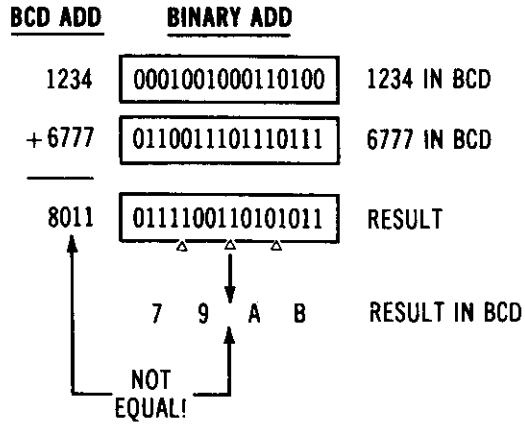


Fig. 7-9. A bcd add with erroneous result.

shown in Figure 7-9, where the bcd add of 1234 and 6777 produces 8011H and the binary add of the two numbers produces 79ABH. It turns out that to convert a binary result of the add of two bcd operands into bcd, it is only necessary to

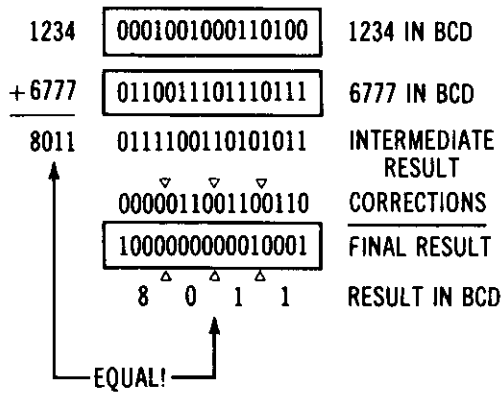


Fig. 7-10. Bcd corrections.

look at each of the groups of four bits to see whether or not a *correction* is required. If a 4-bit group in the result contains 1010, 1011, 1100, 1101, 1110, or 1111, or if a carry *from* the group resulted, then 0110 is added to the group to adjust the binary result to a bcd result. As every byte holds two bcd digits, two such checks are necessary for each binary byte. The process is shown in Figure 7-10, where corrections are made to the operands shown in Figure 7-9.

Bcd subtractions require the same adjustment, but in this case six is *subtracted* if necessary from a bcd digit in the result. It's relatively easy to implement a program to look at each bcd digit and test to see if an add or subtract adjustment is necessary, but the Z-80 does it all in one instruction, the DAA, or *D*ecimal *A*djust *A*ccumulator instruction. When bcd operands are being added or subtracted, the DAA is executed directly after the add or subtract to automatically (aren't computers wonderful) adjust the binary result to a bcd result. To see how this works, we'll write a program to count in bcd for 00 to 99 and compare the results with values stored from 00H through 63H in binary. The following program stores the bcd values from 00 through 99 into a buffer starting at 4B00H and stores a corresponding count from 0 through 99 in binary into a second buffer at 4C00H. Enter the program by assembling and loading or by using T-BUG to enter in machine language, breakpoint at END, and then compare the results in the two buffers. By "dumping" the bcd buffer using the **M** command in T-BUG (with carriage return), you will see a sequence of bcd numbers from 00, 01, 02, 03, 04, 05, 06, 07, 08, 09, 10, 11, up to 99.

```
                00100 ;PROGRAM TO DEMONSTRATE BCD
                00110 ;
4A00            00120         ORG     4A00H
4A00 110000     00130 START   LD      DE,0            ;D=BCD,E=BINARY
4A03 DD21004B   00140         LD      IX,4B00H        ;BCD BUFFER
4A07 FD21004C   00150         LD      IY,4C00H        ;BINARY BUFFER
4A0B 0664       00160         LD      B,100           ;COUNT
4A0D DD7200     00170 LOOP    LD      (IX),D          ;STORE BCD
4A10 FD7300     00180         LD      (IY),E          ;STORE BINARY
4A13 DD23       00190         INC     IX              ;BUMP BCD POINTER
4A15 FD23       00200         INC     IY              ;BUMP BINARY POINTER
4A17 7A         00210         LD      A,D             ;GET BCD
```

```
401B C601       00220          ADD     A,1          ;ADD 1
401A 27         00230          DAA                  ;DECIMAL ADJUST
401B 57         00240          LD      D,A          ;SAVE BCD VALUE
401C 7B         00250          LD      A,E          ;GET BINARY
401D C601       00260          ADD     A,1          ;ADD 1
401F 5F         00270          LD      E,A          ;SAVE BINARY VALUE
4020 10EB       00280          DJNZ    LOOP         ;GO IF NOT 100
4022 18FE       00290 LOOP1    JR      LOOP1        ;LOOP HERE IF DONE
0000            00300          END
00000 TOTAL ERRORS
LOOP1   4022
LOOP    4000
START   4000
```

Compare Operations

As we described in an earlier chapter, compares are essentially subtracts, where the result of the subtraction is only used to set the cpu flags and is not put into the destination register. Unlike subtracts, compares only operate with 8-bit operands, and one of the operands must be in the A register. Compares and subtracts may be used to test two operands for the same states as BASIC comparisons—tests for an operand greater than another, greater or equal, equal, not equal, less than or equal, or less than. Some of these tests are directly handled by the zero and sign flags, while others must use the carry flag.

The test for equality or non-equality is simple and uses the zero flag. In the following code a branch is made to NOTEQ if the contents of the A register are not equal to the contents of the B register and to EQUAL if the two registers are the same.

```
TEST  CP   B          ;TEST BY A-B
      JP   Z,EQUAL    ;GO IF A=B
      JP   NOTEQ      ;MUST BE A NE B HERE
```

When the two numbers to be compared are absolute (unsigned) numbers, the carry flag will be set after the compare if the contents of A are less than the second operand. If A holds 128 and the C register holds 130, for example, the branch to LESSTH will be taken in the code below.

```
TEST    CP   C          ;TEST A-C
        JP   C,LESSTH   ;GO IF A LESS THAN C
        JP   Z,EQUAL    ;GO IF A=C
GTHAN   ...             ;A GREATER THAN C HERE
```

When the two numbers to be compared are signed numbers, then the carry flag logic gets rather confusing. For this reason we present a general-purpose subroutine that compares two signed numbers and jumps to one of three locations based on a comparison of the operands. By making the branch locations identical, any combination of equality conditions may be constructed. If a branch is to be made on greater or equal, for example, the greater than branch will be to GTEQU and the equal branch will also be to GTEQU, with the less than branch to some other location.

```
                00100 ;SUBROUTINE TO COMPARE TWO 8-BIT SIGNED OPERANDS
                00110 ;    ENTRY: (A)=OPERAND 1
                00120 ;           (B)=OPERAND 2
                00130 ;           CALL   CMPARE        ;CALL SR
                00140 ;           (RTN FOR A LT B)     ;PUT JP  LESST HERE
                00150 ;           (RTN FOR A=B)        ;PUT JP  EQUAL HERE
                00160 ;           (RTN FOR A GT B)     ;PUT JP  GREATR HERE
                00170 ;    EXIT:  (A)=UNCHANGED
                00180 ;           (B)=UNCHANGED
                00190 ;           (HL)=DESTROYED
                00200 ;
4A0E            00210          ORG    4A00H          ;CHANGE ON REASSEMBLY
4A00 E1         00220 CMPARE   POP    HL             ;GET RTN ADDRESS
4A01 D5         00230          PUSH   DE             ;SAVE DE
4A02 110300     00240          LD     DE,3           ;ADDRESS INCREMENT
4A05 B8         00250          CP     B              ;COMPARE A:B
4A06 280A       00260          JR     Z,EQUAL        ;GO IF EQUAL
4A08 F5         00270          PUSH   AF             ;SAVE FLAGS
4A09 A8         00280          XOR    B              ;TEST SIGN BITS
4A0A 17         00290          RLA                   ;XOR TO C
4A0B DA154A     00300          JP     C,DIFFER       ;GO IF DIFFERENT SIGNS
4A0E F1         00310          POP    AF             ;RESTORE FLAGS
4A0F D802       00320          JR     C,LESST        ;GO IF A LT B
4A11 19         00330 GREATR   ADD    HL,DE          ;BUMP RTN BY 3
```

```
4A12 19      00340 EQUAL  ADD   HL,DE        ;BUMP RTN BY 3
4A13 D1      00350 LESST  POP   DE           ;RESTORE DE
4A14 E9      00360        JP    (HL)         ;RTN TO 0,3,6
4A15 F1      00370 DIFFER POP   AF           ;RESTORE FLAGS
4A16 DA114A  00380        JP    C,GREATR     ;GO IF A GT B
4A19 C3134A  00390        JP    LESST        ;A LT B
0000         00400        END
00000 TOTAL ERRORS
GREATR  4A11
LESST   4A13
DIFFER  4A15
EQUAL   4A12
CMPARE  4000
```

The block compare is used in string searches and will be discussed in Chapter 9 when we look at strings and tables.

CHAPTER 8

Logical Operations, Bit Operations, and Shifts

The operations in this chapter differ from the arithmetic operations in the last chapter in that the operations here are all concerned with subdivisions of bytes, either *fields* of a byte or down to the individual bit level. The logical instructions are used to retrieve or store information in segments less than a byte in length, the bit instructions manipulate individual bits in memory or register bytes, and the shifts align fields or manipulate individual bits.

AND, ORs, and Exclusive ORs

The AND instruction is used primarily to *mask out* unwanted data in bytes. Suppose, for example, that in each byte of data in a table in memory we had an ASCII character representing the digits of 0 through 9. Now it turns out that the ASCII representation of those digits follows a rather logical order as the reader can see from Table 8-1. The ASCII representation of 0 is 30H, 1 is 31H, and so on up to 39H for 9. To convert one ASCII digit of 30H through 39H into a binary value equivalent to the ASCII character, it is only necessary to get rid of the *bias* of 30H. This could be done by subtraction, but an equivalent alternative would be to mask out the "3" portion of the ASCII by an AND.

```
LD    A,ASCII  ;GET ASCII VALUE
AND   0FH      ;GET LAST FOUR BITS
```

When the ASCII values are masked by the immediate value 0FH (00001111), only the last four bits fall through, and since the least significant four bits are 0 through 9 in this case, the result is the equivalent binary value.

Table 8-1. ASCII Representation of Decimal and Hexadecimal

	Digit	ASCII Code
Decimal	0	30H
	1	31H
	2	32H
	3	33H
	4	34H
	5	35H
	6	36H
	7	37H
	8	38H
	9	39H
Hexadecimal	A	41H
	B	42H
	C	43H
	D	44H
	E	45H
	F	46H

Conversely, a binary value of 0 through 9 could be converted into an equivalent ASCII value for output by setting the "3" bits. Although an add could be used, the ASCII values could also be generated by an OR instruction.

```
LD   A,(BINARY)  ;GET BINARY VALUE
OR   30H         ;CONVERT TO ASCII
```

In both of the preceding cases we have assumed that only valid ASCII characters of 0 through 9 are involved, and that the binary values will be 0 through 9. As a simple illustration of this conversion, let's write out the screen line number 0 through 9, for the first ten lines of the screen. The following program does this by counting for 0 through 9 and ORing in the "3" value to make an ASCII digit out of the count.

```
                00100 ;WRITE OUT LINES 0-9 IN ASCII
                00110 ;
4000            00120       ORG     4000H
4000 21203C     00130       LD      HL,3000H+32     ;MIDDLE OF 1ST LINE
```

```
4003 0600      00140        LD      B,0          ;INITIALIZE COUNT
4005 0E39      00150        LD      C,39H        ;LAST ASCII
4007 114000    00170        LD      DE,64        ;LINE INCREMENT
400A 78        00180 LOOP   LD      A,B          ;GET CURRENT COUNT
400B F630      00190        OR      30H          ;CONVERT TO ASCII
400D 77        00200        LD      (HL),A       ;STORE ON SCREEN
400E 19        00210        ADD     HL,DE        ;BUMP LINE PNTR
400F 04        00220        INC     B            ;BUMP COUNT
4010 B9        00230        CP      C            ;TEST FOR END
4011 C20A40    00240        JP      NZ,LOOP      ;GO IF NOT DONE
4014 C31440    00250 LOOP1  JP      LOOP1        ;LOOP HERE AT END
0000           00260        END
00000 TOTAL ERRORS
LOOP1   4014
LOOP    400A
```

The exclusive OR does not find as much use as the AND and OR instructions. Recall that the exclusive OR generates a one bit in the result if there is a single one bit but not two one bits in the bit positions of the two operands. The most common use of the exclusive OR in the TRS-80 is to zero the accumulator by the efficient instruction.

```
XOR  A  ;ZERO A REGISTER AND CARRY
```

Another use of the exclusive OR is to *toggle* a counter from 0 to 1 and back again as in

```
        LD    A,1       ;SET TOGGLE TO ONE
LOOP    XOR   1         ;TOGGLE
        JP    Z,ZERO    ;GO IF ZERO
        JP    ONE       ;ONE ACTION
```

One of the more common operations in the Z-80 and other computers is to set or reset a bit in a memory byte or register byte. To set a bit in memory in many computers, the following three instructions must be executed

```
LD   A,(HL)   ;LOAD THE MEMORY BYTE
OR   A,4      ;SET BIT 2
LD   (HL),A   ;STORE BYTE WITH BIT SET
```

Similarly, resetting any of the eight bits of a memory byte calls for

```
LD    A,(HL)    ;LOAD THE MEMORY BYTE
AND   A,0FBH    ;RESET BIT 2
LD    (HL),A    ;STORE BYTE WITH BIT RESET
```

Lastly, testing a bit of a memory location requires a load and test, usually an AND

```
LD    A,(HL)    ;LOAD THE MEMORY BYTE
AND   A,4       ;TEST BIT 2
JP    Z,ZERO    ;GO IF BIT 2 = 0
JP    ONE       ;BIT 2 = 1
```

Bit Instructions

In the Z-80 only one instruction is required to set, reset, or test any one bit of a memory or cpu register bit. The instruction SET 2,(HL) takes the place of the three instructions for setting a bit, RES 2,(HL) causes a reset of bit 2, and BIT 2,(HL) sets the zero flag to the condition of the bit. Since these sets, resets, and tests are continually being done in assembly language programming, the bit instructions are quite powerful.

Shiftless Computers

It is possible to perform the actions of aligning data, dividing and multiplying by powers of two, and bit testing without shift instructions, but the Z-80 shifts are much more efficient than other shiftless instruction sets, and make these common operations much easier to perform.

Often shifts are used to align data, that is, to move fields within bytes to a desired location. The Z-80 shift instructions for data alignment are the *Rotate* instructions. Rotates are either 8-bit rotates or 9-bit rotates. The 8-bit rotates move the 8 bits within a register or memory location out one end and in the other, as shown in Figure 8-1. The 9-bit rotates rotate the carry along with the 8 register or memory data bits. Both types of rotates have their uses.

Rotates

As an example of use of rotate, let's write a routine that will output the contents of a block of memory locations in binary. Each memory location has eight bits, of course, and we must convert each bit to an ASCII one or zero for display. The following code outputs locations 0 through 0FH to the screen in binary ASCII.

```
                00100 ;ROUTINE TO DUMP IN BINARY
                00110 ;
4A00            00120           ORG     4A00H
4A00 DD212D3C   00130 START   LD      IX,3C00H+32     ;MIDDLE OF LINE 0
4A04 FD21004B   00140           LD      IY,4B00H        ;START OF DUMP LOC
4A08 113800     00150           LD      DE,56           ;LINE INCREMENT
4A0B 0610       00160           LD      B,16            ;LINE COUNT
4A0D D9         00170 LOOP1     EXX                     ;SWITCH REGISTERS
4A0E 0608       00180           LD      B,8             ;BIT COUNT
4A10 3E30       00190 LOOP2     LD      A,30H           ;ASCII 0
4A12 FDCB0006   00200           RLC     (IY)            ;ROTATE LEFT
4A16 3001       00210           JR      NC,LOOP3        ;GO IF 0
4A18 3C         00220           INC     A               ;CHANGE 0 TO 1
4A19 DD7700     00230 LOOP3     LD      (IX),A          ;STORE 0 OR 1
4A1C DD23       00235           INC     IX              ;NEXT CHARACTER POSTN
4A1E 10F0       00240           DJNZ    LOOP2           ;GO IF NOT 8 BITS
4A20 D9         00250           EXX                     ;SWITCH BACK
4A21 FD23       00260           INC     IY              ;BUMP LOCATION PNTR
4A23 DD19       00270           ADD     IX,DE           ;POINT TO NEXT LINE
4A25 10E6       00280           DJNZ    LOOP1           ;GO IF NOT 16 LOOPS
4A27 18FE       00290 LOOP4     JR      LOOP4           ;LOOP HERE ON DONE
0000            00300           END
00000 TOTAL ERRORS
LOOP4   4A27
LOOP3   4A19
LOOP2   4A10
LOOP1   4A0D
START   4A00
```

This is our most complicated program thus far, and it bears some detailed study. The IX register is used to point to the current screen line, starting at the middle of the first line. The IY register is used to point to the location to be *dumped*, in this case starting at 4B00. DE holds the line increment to be added to IX to point to the next display line. Since we're going to be writing out 8 ASCII bytes on each line, the increment on

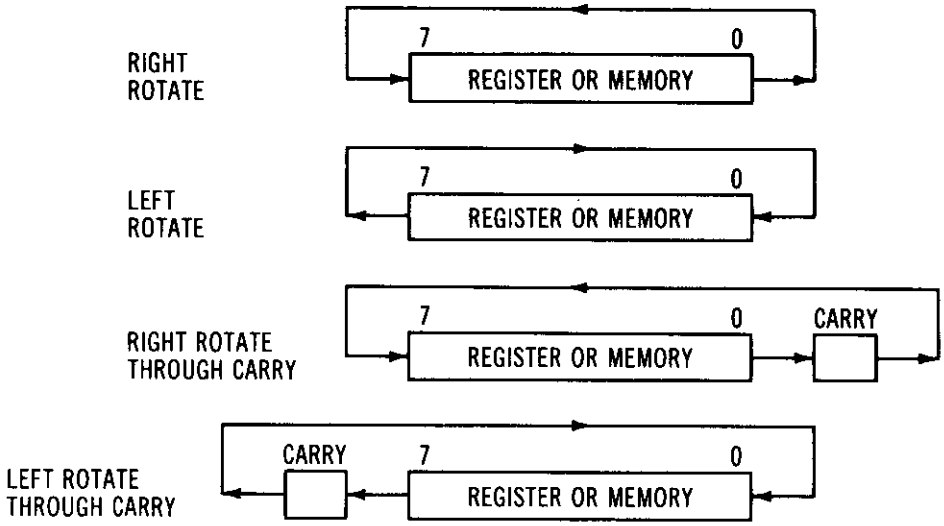


Fig. 8-1. Rotate operation.

this is (64-8) or 56. B is initialized with the number of locations to be dumped or 16.

The main loop in the code starts at LOOP1. The first instruction swaps the inactive and active set of cpu registers B through L. This is done to enable us to use more cpu registers since just about every one is in use. Now we can use B of the second set to hold a bit count for the inner loop of 4A10 through 4A1F that looks at the 8 bits and outputs them in ASCII to the screen. The inner loop rotates the location pointed to by IY. As each rotate is done, the leftmost bit is rotated both to the carry and around to the right-hand side of the memory location. The carry is tested to store either a 30H, for an ASCII zero, or 31H, for an ASCII one. Eight rotates are done, and at the end the memory location in the 4B00H area has been rotated completely around.

For each store of an ASCII one or zero, IX is incremented to point to the next character position on the line. When the count in B is decremented down to zero, an EXX switches back the cpu registers, restoring the original count in B for number of lines. IY is incremented to point to the next memory location in the 4B00H area, and IX is incremented by 56 to point to the next line for display. If 16 locations have not been dumped, the next location is stored as eight ASCII characters.

A program that has several nested loops such as this can be confusing to a programmer seeing it for the first time. It can also be confusing to the programmer who wrote it when he picks it up several months later! One convenient way to get a clear picture of what is going on in a program such as this is to "play computer." On a sheet of paper, make columns rep-

resenting the registers that are in use in the program. Then step through the program one instruction at a time, filling in the proper values in the registers. It isn't necessary to loop all the way through some of the loops (65536 loops makes for a lot of writing), but it does make many programs very clear. See Figure 8-2.

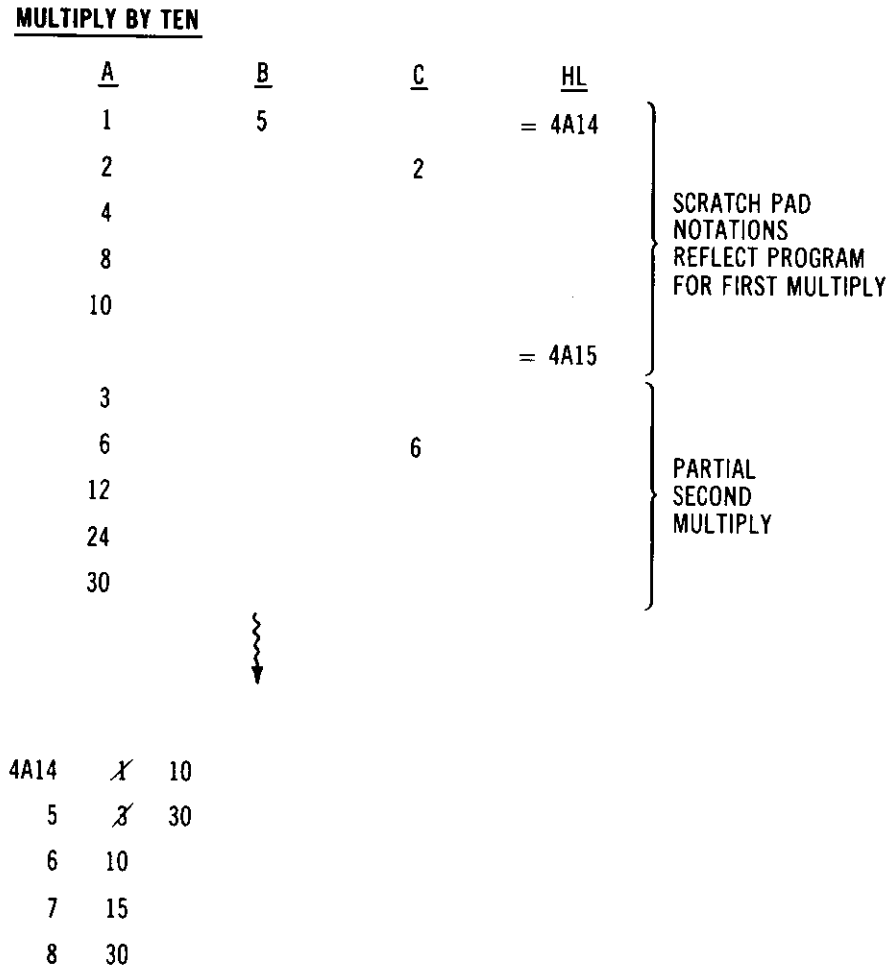
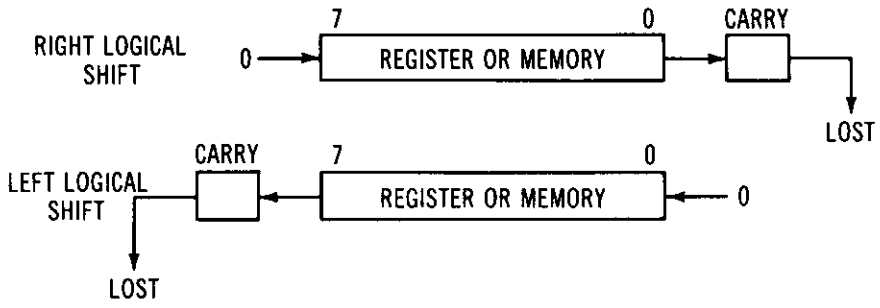


Fig. 8-2. Playing computer.

Some Shifting Is Very Logical

Logical shifts differ from rotates in that data shifted off the end of the register or memory location is lost. Zeros are used to shift into the byte from the other end, as shown in Figure 8-3. Logical shifts are used to align data as in the rotate case, and to divide or multiply by two. If an 8-bit value is shifted left one bit, the effect is to multiply the original value by two while a shift right of one bit position divides the original value by two and discards the remainder.



	SAMPLE RIGHT LOGICAL SHIFT		SAMPLE LEFT LOGICAL SHIFT	
ORIGINAL NUMBER	01010000	(80_{10})	00001011	(11_{10})
1 SHIFT	00101000	(40_{10})	00010110	(22_{10})
2 SHIFTS	00010100	(20_{10})	00101100	(44_{10})
3 SHIFTS	00001010	(10_{10})	01011000	(88_{10})
4 SHIFTS	00000101	(5_{10})	10110000	(-80_{10})!
5 SHIFTS	00000010	(2_{10})	01100000	(96_{10})!
6 SHIFTS	00000001	(1_{10})	11000000	(-64_{10})!
7 SHIFTS	00000000	(0)	10000000	(-128_{10})!

Fig. 8-3. Logical shift operation.

All rotates, logical shifts, and arithmetic shifts in the Z-80 operate only one bit at a time, so that a shift of four bit positions requires four separate shifts. To show how shifts may be used to multiply, consider the following code. Multiplication by ten is a common problem in many programs. For example, keyboard values may be input in ASCII and represent a string of decimal digits, such as 567.89, that must be converted to binary values for arithmetic manipulations within the program. MULTEN takes an 8-bit value from memory, multiplies it by ten, and stores it back into the memory location.

```
                00100 ;MULTIPLY BY TEN ROUTINE
                00110 ;
4000            00120          ORG     4000H
4000 21154A     00130 MULTEN   LD      HL,DATA         ;TABLE OF DATA
4003 0605       00140          LD      B,5             ;FOR FIVE VALUES
4005 7E         00150 LOOP     LD      A,(HL)          ;GET VALUE
4006 CB27       00160          SLA     A               ;VALUE*2
4008 4F         00170          LD      C,A             ;SAVE VALUE*2
```

```
4A09 CB27     00180        SLA   A          ;VALUE*4
4A0B CB27     00190        SLA   A          ;VALUE*8
4A0D 81       00200        ADD   A,C        ;VALUE*10
4A0E 77       00210        LD    (HL),A     ;RESTORE
4A0F 23       00220        INC   HL         ;POINT TO NEXT VALUE
4A10 10F3     00230        DJNZ  LOOP       ;CONTINUE
4A12 C3124A   00240 LOOP1  JP    LOOP1      ;LOOP HERE IF DONE
4A15 01       00250 DATA   DEFB  1
4A16 03       00260        DEFB  3
4A17 0A       00270        DEFB  10
4A18 0F       00280        DEFB  15
4A19 1E       00290        DEFB  30
0000          00300        END
00000 TOTAL ERRORS
LOOP1  4A12
LOOP   4A05
DATA   4A15
MULTEN 4A00
```

After the value is loaded into the A register it is shifted left by the SLA A to multiply the value by two. This value is then saved in the C register. Now the A register is shifted left two more times to multiply the original value by four and eight. Now the value in the C register, which represents the original value times two, is added to the value times eight to give a result of the value times ten. Execute the program with a breakpoint at 4A12 and then look at the table locations to see the results. Note that the multiply was an unsigned (absolute) multiply, and that in one case (30), the result was too large for the 8-bit memory location. In this case only the lower-order eight bits of the result are in the memory location!

Arithmetic Shifts

The TRS-80 has one shift that is an arithmetic-type shift (even though the mnemonic for the SLA is *S*hift *L*eft *A*rithmetic it is really a logical shift). The SRA (*S*hift *R*ight *A*rithmetic) always retains the *sign* of the operand to be shifted as shown in Figure 8-4. The bit in bit 7 is shifted right to bit 6,

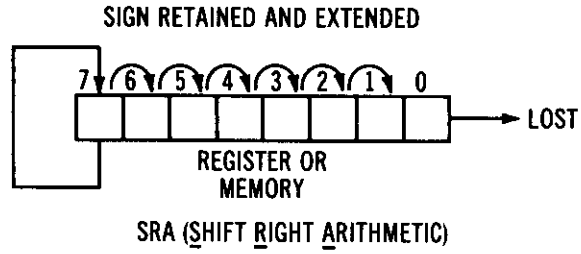


Fig. 8-4. Arithmetic shift operation.

but also goes back into bit 7 as the sign. The process is called *sign extension* as the sign is extended to the right. The SRA may be used to divide a signed 8-bit operand by two. The operation for a value of −37 is shown in Figure 8-5.

Software Multiply and Divide

What! No multiply and divide instructions in the Z-80! That's right, and no current 8-bit microprocessor has them either. Before you pull out that weathered four-function calculator, let's see how multiply and divide can be implemented in software.

There are a number of approaches in writing a multiply routine for any computer. The easiest is repetitive addition. Mulplying 63 by 15 is really only adding 63 to itself 14 times

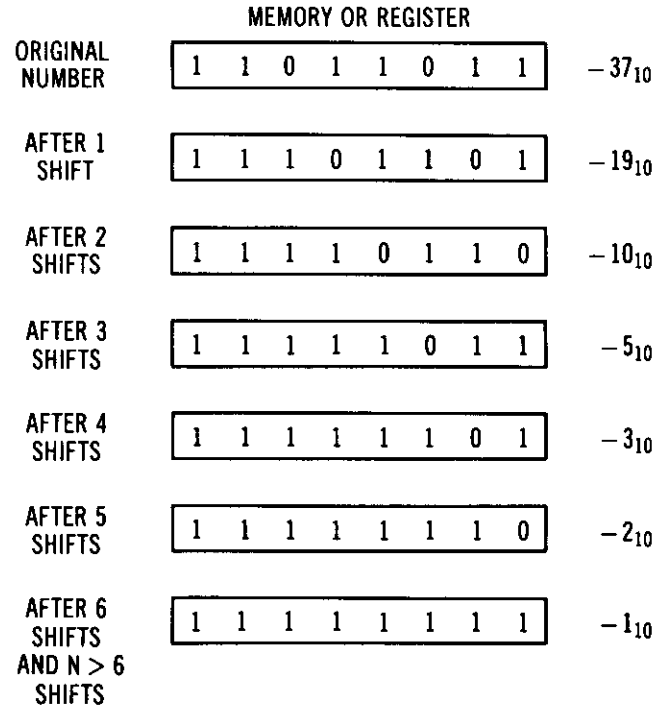


Fig. 8-5. Arithmetic shift example.

(or adding 15 63 times), and that's very easy to implement in the Z-80. The following routine uses this approach to multiply the 16-bit absolute value in DE by an 8-bit *multiplier* in B, which is also unsigned or absolute. The product is in HL at completion (use the R command to see the product).

```
                00100 ;REPETITIVE ADDITION MULTIPLY
                00110 ;
4000            00120           ORG     4A00H
4000 ED5B114A   00130 START     LD      DE,(ARG1)       ;LOAD MULTIPLICAND
4004 3A134A     00140           LD      A,(ARG2)        ;LOAD MULTIPLIER
4007 47         00150           LD      B,A             ;TRANSFER TO B
4008 210000     00160           LD      HL,0            ;CLEAR PARTIAL PRODUCT
400B 19         00170 LOOP      ADD     HL,DE           ;ADD MULTIPLICAND
400C 10FD       00180           DJNZ    LOOP            ;GO IF NOT DONE
400E C30E4A     00190 LOOP1     JP      LOOP1           ;LOOP HERE ON DONE
4011 E803       00200 ARG1      DEFW    1000            ;PUT MULTIPLICAND HERE
4013 1400       00210 ARG2      DEFB    20              ;PUT MULTIPLIER HERE
0000            00220           END
00000 TOTAL ERRORS
LOOP1   400E
LOOP    400B
ARG2    4013
ARG1    4011
START   4000
```

As short and sweet as this routine is, it does have a serious disadvantage. It is horrendously slow, compared to other ways in which the multiply could be implemented. Use this approach only when the multiplier is small. It is efficient when multipliers of ten or less will be used.

The usual way of implementing a software multiply is to use the same approach as the pencil and paper method for decimal numbers. In this approach a shifted multiplicand multiplied by the digit in the multiplier is added to other partial products to get the final product as shown in Figure 8-6. Binary multiplication using this technique is fairly simple as the value to be added can *only be* the multiplicand or zero, depending upon the value of the multiplier bit. The following

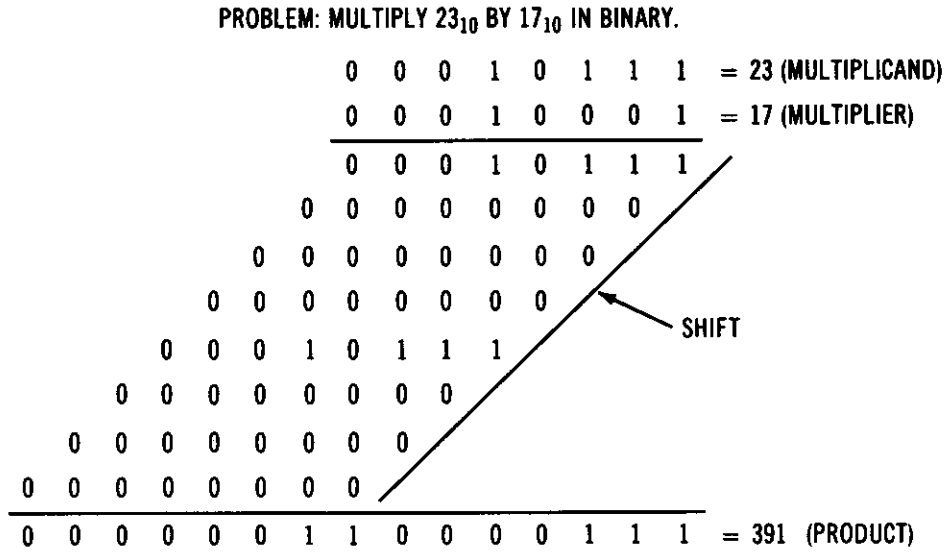


Fig. 8-6. Multiplication methods.

routine is one of the "standard" routines that might be helpful in the user's programs. It multiplies an *unsigned* 16-bit value in DE by an *unsigned* 8-bit value in the B register and returns the product in HL. The B register contents is zero upon return.

```
                00100 ;SUBROUTINE TO MULTIPLY 16 BY 8
                00110 ;   ENTRY:(DE)=MULTIPLICAND,UNSIGNED
                00120 ;         (B)=MULTIPLIER,UNSIGNED
                00130 ;         CALL  MUL16
                00140 ;   EXIT: (HL)=PRODUCT
                00150 ;         (DE)=DESTROYED
                00160 ;         (B)=0
                00170 ;
4000            00180         ORG   4000H       ;CHANGE ON REASSEMBLY
4000 210000     00190 MUL16   LD    HL,0        ;CLEAR PARTIAL PRODUCT
4003 CB38       00200 LOOP    SRL   B           ;SHIFT OUT M'IER BIT
4005 3001       00210         JR    NC,CONT     ;GO IF NO CARRY (1 BIT)
4007 19         00220         ADD   HL,DE       ;ADD MULTIPLICAND
4008 C8         00230 CONT    RET   Z           ;GO IF M'IER
4009 EB         00240         EX    DE,HL       ;MULTIPLICAND TO HL
400A 29         00250         ADD   HL,HL       ;SHIFT MULTIPLICAND
400B EB         00260         EX    DE,HL       ;SWAP BACK
```

```
400C C3034A   00270        JP     LOOP          ;CONTINUE
0000          00280        END
00000 TOTAL ERRORS
CONT   4A03
LOOP   4A03
MLL16  4A00
```

Note that in the above routine the ADD HL,DE *does not* affect the zero flag, allowing it to be used for a check on the shifted result in B *after* the add.

Divide routines implemented in software are not nearly so neat. Experienced programmers have been known to wail and gnash their teeth while trying to implement an efficient divide routine on certain computers. Successive subtraction may be used, but it is as slow as a multiply routine using this approach, and should be used only with operations resulting in small quotients. The following code divides the contents of the HL register, an *unsigned* 16-bit number by the contents of DE, an unsigned 16-bit *divisor*. Both numbers must be less than 32,768. The quotient is in B at the end, and any remainder is in HL. If the quotient is larger than 255 overflow will result.

```
                00100 ;DIVIDE BY SUCCESSIVE SUBTRACTION
                00110 ;
4A00            00120        ORG    4A00H
4A00 2A1A4A     00130 START  LD     HL,(ARG2)     ;GET DIVISOR
4A03 E5         00140        PUSH   HL            ;TRANSFER TO DE
4A04 D1         00150        POP    DE
4A05 2A184A     00160        LD     HL,(ARG1)     ;DIVIDEND
4A08 0600       00170        LD     B,0           ;CLEAR QUOTIENT
4A0A B7         00180 LOOP   OR     A             ;CLEAR CARRY FOR SUBTR
4A0B ED52       00190        SBC    HL,DE         ;DIVIDEND-DIVISOR
4A0D FA144A     00200        JP     M,DONE        ;GO IF DONE
4A10 04         00210        INC    B             ;BUMP QUOTIENT
4A11 C30A4A     00220        JP     LOOP          ;CONTINUE
4A14 19         00230 DONE   ADD    HL,DE         ;FIND TRUE REMAINDER
4A15 C3154A     00240 LOOP1  JP     LOOP1         ;LOOP HERE ON DONE
4A18 204E       00250 ARG1   DEFW   20000         ;ARG1/ARG2
4A1A C800       00260 ARG2   DEFW   200
```

```
0000        00270       END
00000 TOTAL ERRORS
LOOP1   4015
DONE    4014
LOOP    400A
ARG1    4018
ARG2    401A
START   4000
```

In the routine the divisor is repeatedly subtracted from the dividend until the dividend goes negative. When this occurs, the *residue* is changed to a true remainder by adding back the divisor. Each time the subtraction can be successfully made the contents of B are incremented by one to show the quotient. This method exactly emulates what can be done with pencil and paper.

A more general-purpose divide for an unsigned 16-bit dividend and unsigned 8-bit divisor is shown in the following "standard" subroutine. Here the division is a *restoring* type similar to a paper and pencil approach. Instead of asking itself "Does the divisor go into the next group of digits," however, the computer in this case blindly goes ahead and attempts the divide. If the divisor doesn't go, then the previous residue is *restored* by adding back the shifted dividend, similar to what was done in the successive subtraction case.

```
                00100 ;SUBROUTINE TO DIVIDE 16 BY 8
                00110 ;   ENTRY:(HL)=DIVIDEND 16 BITS
                00120 ;         (D)=DIVISOR 8 BITS
                00125 ;         CALL  DIV16
                00130 ;   EXIT: (IX)=QUOTIENT 16 BITS
                00140 ;         (H)=REMAINDER 8 BITS
                00150 ;         (L)=DESTROYED
                00160 ;         (D)=UNCHANGED
                00170 ;         (E)=0
                00180 ;         (A)=DESTROYED
                00190 ;
4000            00200       ORG     4000H       ;CHANGE ON REASSEMBLY
4000 7D         00210 DIV16 LD      A,L         ;LS BYTE DIVDND
4001 6C         00220       LD      L,H         ;MS BYTE DIVDND
```

```
4002 2600      00230        LD      H,0          ;CLEAR FOR SUBT
4004 1E00      00240        LD      E,0          ;SETUP FOR SUBTRACT
4006 0610      00250        LD      B,16         ;16 ITERATIONS
4008 DD210000  00260        LD      IX,0         ;INITIALIZE QUOTIENT
400C 29        00270 LOOP   ADD     HL,HL        ;SHIFT DIVD LEFT
400D 17        00280        RLA                  ;SHIFT 8 LS BITS
400E D21240    00290        JP      NC,LOOP1     ;GO IF 0 BIT
4011 2C        00300        INC     L            ;SHIFT TO HL
4012 DD29      00310 LOOP1  ADD     IX,IX        ;SHIFT QUOTIENT LEFT
4014 DD23      00320        INC     IX           ;0 BIT=1
4016 B7        00330        OR      A            ;CLEAR CARRY FOR SUB
4017 ED52      00340        SBC     HL,DE        ;TRY SUBTRACT
4019 D21F40    00350        JP      NC,CONT      ;GO IF IT WENT
401C 19        00360        ADD     HL,DE        ;RESTORE
401D DD2B      00370        DEC     IX           ;SET 0 BIT=0
401F 10EB      00380 CONT   DJNZ    LOOP         ;GO IF NOT 16
4021 C9        00390        RET                  ;RETURN
0000           00400        END
00000 TOTAL ERRORS
CONT    401F
LOOP1   4012
LOOP    400C
DIV16   4000
```

The register setup before and after the divide is shown in Figure 8-7. The divisor in D is repetitively subtracted from the residue of the dividend in HL. The residue is shifted over one bit position for every iteration just the way it is done by the paper and pencil method. If the subtract for any *iteration* is successful, a one bit is left in the quotient; if the subtract is not successful a zero bit is put in the quotient. The quotient is shifted left one bit for every iteration as less and less significant subtracts are made. After 16 bits the IX register holds the possible 16-bit quotient, the H register holds an 8-bit remainder, D holds the original divisor, and E is zeroed. One interesting point is that both the HL and IX registers are effectively shifted left one bit position in a logical shift by adding HL or IX to themselves. It may benefit the reader to actually play computer on this routine and step through the 16 iterations of the divide while using actual numeric values.

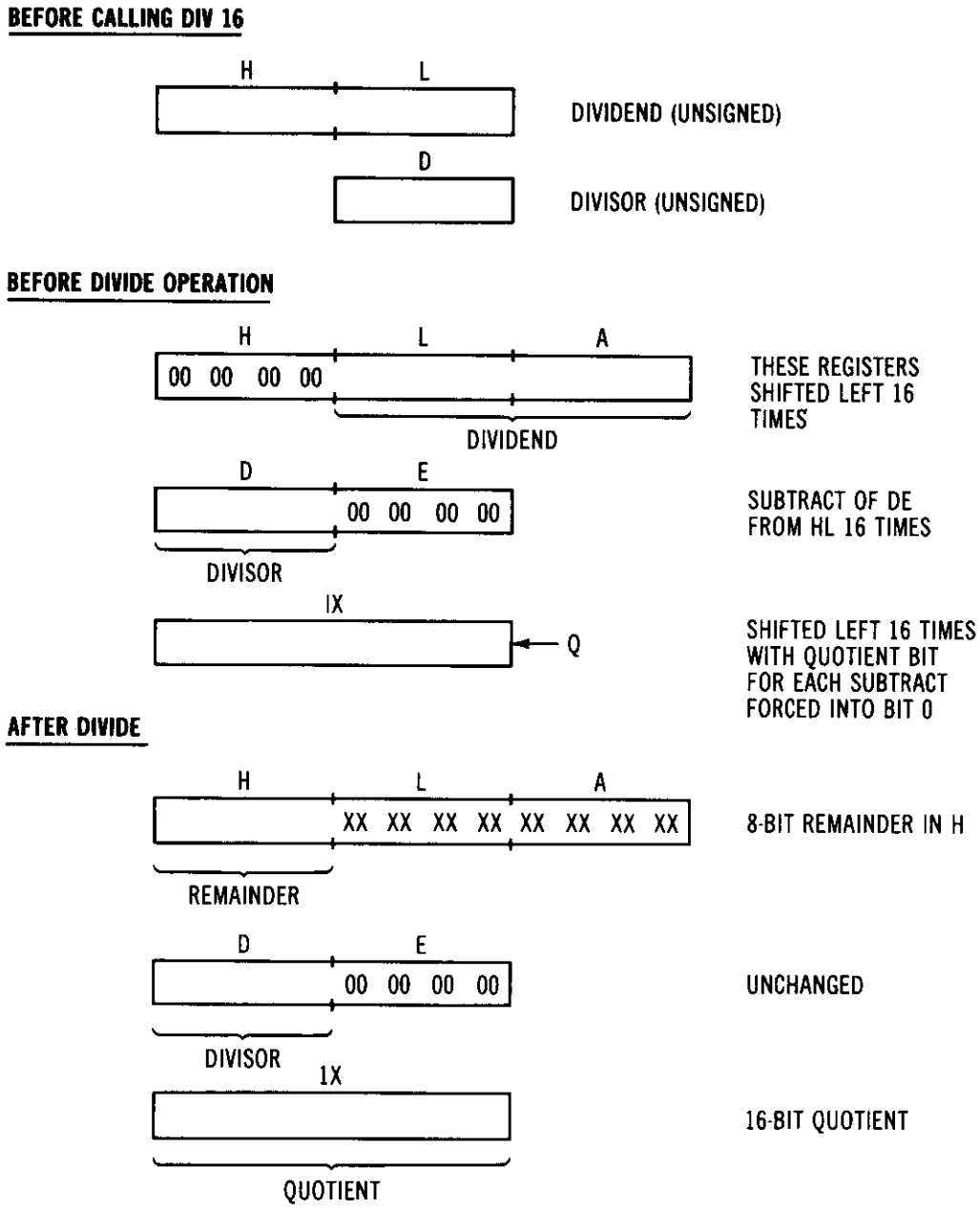


Fig. 8-7. Divide register setup.

At the end you may vow never to do it again, but it will give some insight into this type of operation.

The preceding multiply and divide routines are *unsigned* multiplies and divides. It is possible to implement *signed* multiplies and divides, but they are not as neatly packaged as the unsigned. The unsigned routines may be used to implement a signed multiply or divide if the operands are changed to their absolute values and the results changed again to their proper signs. However, watch for overflow conditions when this approach is used, such as multiplying −128 by −128!

Input and Output Conversions

The techniques of shifting, multiplications, and divides that we covered in this chapter are very useful in conversion between internal data representation and ASCII. Most programs require some type of input of ASCII data from keyboard, usually in decimal, and that string of decimal digits must be converted into an eight, sixteen, or larger number of bits so that the program can process the data. Sometimes, as in the case of T-BUG, there must be a way of converting from a *hexadecimal* ASCII input to eight or sixteen bits, and infrequently, a way of converting ASCII *binary* into internal data values. Similarly, once the data has been processed, it must be displayed in a more convenient form, which usually means ASCII decimal, but which may also be hexadecimal (in the T-BUG case) or binary.

We have already covered one conversion in an earlier program in this chapter, the conversion of eight bits into equivalent ASCII ones or zeros for display. The conversion of

```
00100 ;SUBROUTINE TO CONVERT FROM HEX TO ASCII
00110 ;
00120 ;   ENTRY:(A)=8-BIT VALUE TO BE CONVERTED
00130 ;         CALL   HEXCV
00140 ;         (RETURN)
00150 ;   EXIT: (HL)=TWO ASCII VALUES, HIGH AND LOW
00160 ;         (A)=DESTROYED
00170 ;         (C)=DESTROYED
00180 ;
4A00          00190         ORG    4A00H          ;CHANGE ON REASSEMBLY
4A00 4F       00200 HEXCV   LD     C,A            ;SAVE TWO HEX DIGITS
4A01 CB3F     00210         SRL    A              ;ALIGN HIGH DIGIT
4A03 CB3F     00220         SRL    A
4A05 CB3F     00230         SRL    A
4A07 CB3F     00240         SRL    A
4A09 CD154A   00250         CALL   TEST           ;CONVERT TO ASCII
4A0C 67       00260         LD     H,A            ;SAVE FOR RTN
4A0D 79       00270         LD     A,C            ;RESTORE ORIGINAL
4A0E E60F     00280         AND    0FH            ;GET LOW DIGIT
```

```
4012 CD154A   00290        CALL   TEST          ;CONVERT TO ASCII
4013 6F       00300        LD     L,A           ;SAVE FOR RTN
4014 C9       00310        RET
4015 C630     00320 TEST   ADD    A,30H         ;CONVERSION FACTOR
4017 FE3A     00330        CP     3AH           ;TEST FOR 0-9
4019 FA1E4A   00340        JP     M,TEST1       ;GO IF 0-9
401C C607     00350        ADD    A,7           ;CORRECT FOR A-F
401E C9       00360 TEST1  RET                  ;RETURN
0000          00370        END
00000 TOTAL ERRORS
TEST1  401E
TEST   4015
HEXCV  4000
```

hexadecimal values to ASCII digits of 0 through 9 and A through F is a similar problem. Let's write a program to convert any number in the A register into two hexadecimal ASCII digits. We will also write a simple *driver* to use the program and display some data.

Program HEXCV is the general-purpose routine to perform the conversion. The first four bits, representing the first hexadecimal digit are shifted 4 bit positions right in the A register. They are now aligned in the A register, and the register holds a value of from 0000 through 1111 (the upper four bits are zero), representing hexadecimal 0 through F. The ASCII equivalents for 0 through F are shown in Table 8-1. Unfortunately, there is a "gap" between the digits 0 through 9 and the characters A through F. If there were no gap, 30H could be added to the four bits to compute the ASCII value for the character. Since there is a gap, however, there must be a test for the hexadecimal letter digits, and this is done in the compare. If the conversion resulted in a result greater than 39H, then the ASCII character must be a letter, and 7 is added to obtain the letter value. For the second (least significant) hexadecimal digit, the A register is restored, the upper four bits are *masked out* (the lower four are already aligned) and the same conversion is made. Upon completion, HL holds the two ASCII characters representing the hexadecimal digits.

A simple *driver* to test this routine could be constructed from something similar to the following code.

```
        LD      B,8         ;8 LINES
        LD      IX,3C00H    ;FIRST LINE
        LD      DE,xxxx     ;LOCATIONS TO DISPLAY
LOOP    LD      A,(DE)      ;GET LOCATION
        CALL    HEXCV       ;CONVERT
        LD      (IX),H      ;STORE 1ST CHARACTER
        LD      (IX+1),L    ;STORE 2ND CHARACTER
        INC     IX          ;BUMP POINTER
        INC     IX
        INC     DE          ;BUMP LOCATION POINTER
        DJNZ    LOOP        ;CONTINUE IF NOT 8
```

A driver in this case means a routine to test and exercise the HEXCV routine. This driver displays 8 locations on line 1 of the video display in a string of 16 hexadecimal digits. The reader can undoubtedly see how different formats could be constructed to display hexadecimal data in a more convenient format by using HEXCV and other types of drivers.

Converting input data from ASCII to binary or hexadecimal is about as easy as the output conversion. For binary, the ASCII character representing a binary one or zero is converted to a true binary one or zero by subtracting 30H. This bit is then aligned and merged with other bits representing the 8- or 16-bit input value. Hexadecimal ASCII characters are adjusted by subtraction of 30H. If the result is greater than 9, a second subtract of 7 is performed to convert the letter digit to A through F in hexadecimal. The 4-bit result is merged with a second result or three other results to produce an 8-bit or 16-bit value.

Conversion of decimal data is the most difficult of the three types of cenversions. It is not simply a case of shifting bits as is the case in binary and hexadecimal.

For a conversion of decimal input data, each ASCII character represents a decimal digit from 0 through 9 (30H through 39H). The ASCII character is changed to four bits of bcd by subtracting 30H. Now this result must be multiplied by the power of ten it represents. For example, if the ASCII string was 123, the one would be converted to a bcd 1 and multiplied by 100, the 2 would be converted and multiplied by 10, and the 3 would only be converted. In practice decimal input conversion routines work with five digits as 65,535 can be held in 16 bits, and use a combination of conversion of each of the digits and multiplication of the result by ten for five iterations to convert the data.

For output conversions, an 8- or 16-bit value is converted by division by ten, the resulting remainders adjusted to an ASCII character by addition of 30H, and the result

stored in an intermediate buffer before output. Another approach is to use successive subtractions of the powers of ten, starting with 10000 (for a 16-bit value) to convert the number into decimal values which can then be converted by addition of 30H to ASCII outputs.

CHAPTER 9

Strings and Tables

This chapter discusses two important aspects of assembly-language programs, strings and tables. Strings are generally strings of text characters, just as in BASIC programs. Many assembly-language programs are concerned with separating segments of the string into various fields representing subdivisions of the string data such as names, addresses, mnemonics, and so forth. The Z-80 has a powerful block search capability to help in handling strings. Tables are generally one-dimensional arrays that represent such diverse things as addresses for jumps, sine values, and withholding tax percentages. The Z-80 has many features that permit the assembly-language programmer to work with tables, such as indexing.

Assembler-Generated Strings

We have seen in an earlier chapter how the assembler automatically generates a text string when the DEFM pseudo-op is used. Generally, this pseudo-op is used to produce messages which are output to the display or printer. The code below, for example, outputs a message to the middle of the screen, after the message has been converted from a symbolic source line into ASCII by the assembler.

```
                00100 ;ROUTINE TO OUTPUT MESSAGE
                00110 ;
4000            00120       ORG     4000H
```

```
4A00 210E4A   00130 START  LD     HL,MESS        ;LOAD ADDRESS OF MESS
4A03 11203E   00140        LD     DE,3C00H+544   ;MIDDLE LINE+32
4A06 011100   00150        LD     BC,MESSL       ;LENGTH OF MESS
4A09 EDB0     00160        LDIR                  ;OUTPUT TO SCREEN
4A0B C30B4A   00170 LOOP   JP     LOOP           ;LOOP HERE ON DONE
4A0E 41       00180 MESS   DEFM   'ANOTHER FINE MESS'
4A0F 4E       4A10 4F      4A11 54      4A12 48      4A13 45      4A14 52
   4A15 20      4A16 46      4A17 49      4A18 4E      4A19 45      4A1A 2
0      4A1B 4D      4A1C 45      4A1D 53      4A1E 53      0011         00
185 MESSL  EQU    $-MESS
0000         00190        END
00000 TOTAL ERRORS
LOOP   4A0B
MESSL  0011
MESS   4A0E
START  4A00
```

The program uses the block move LDIR after setting up the register pairs for the parameters of the move. Note that the length of the message has been *generated by the assembler* by equating an assembly variable MESSL to the next assembler location minus the start of the message. When the BC register pair is loaded with MESSL, the assembler loads the immediate field of the load instruction with the length of 11H.

Generalized String Output

In the case above, the message could be moved to the output device in a block, as the output device was really a memory area. If your system has a printer that operates through a parallel or serial port on the TRS-80, the way that an output string is sent to the printer is somewhat different. Let's suppose that subroutine OUTPUT actually communicates with the printer (we'll talk about that communication in the next chapter). The subroutine below CALLs OUTPUT with the next ASCII character to be transmitted to the printer. The problem here is to determine when to stop. Initially, the MESSGE subroutine is called with HL holding the start of the message area. However, we need not only the start of the message area, but the end of the message area, the number of

bytes in the message, or some other means to signal the MESSGE subroutine that the message has come to an end. MESSGE here uses a *terminator* approach to detect the end of the message. The next character is sent to the OUTPUT subroutine as long as a *null* (all zeros) character is not detected. If a null is detected, MESSGE knows that the message area has come to an end and returns to the calling subroutine. A length could have been specified to MESSGE, but the terminator approach is used quite frequently.

```
                00100 ;MESSAGE OUTPUT ROUTINE
                00110 ;
4000            00120         ORG    4000H
4000 7E         00130 START   LD     A,(HL)      ;LOAD NEXT CHARACTER
4001 B7         00140         OR     A           ;TEST FOR NULL
4002 C8         00150         RET    Z           ;RETURN ON ZERO
4003 CD0050     00160         CALL   OUTPUT      ;OUTPUT TO PRINTER
4006 23         00170         INC    HL          ;POINT TO NEXT CHAR
4007 18F7       00180         JR     START       ;CONTINUE
5000            00190 OUTPUT  EQU    5000H       ;PRINTER OUT ROUTINE
0000            00200         END
00000 TOTAL ERRORS
OUTPUT  5000
START   4000
```

In many cases, the message to be output to the screen or I/O device must first be assembled during program execution. In these cases, a *message buffer* area is allocated, and the component parts of the message are moved into the area, and the message is then printed. The approach is valuable for printing variable data that cannot be defined beforehand, and for saving memory when a large number of messages must be printed. In the code below, a message buffer for a mailing list has been defined. The *fields* of the buffer are defined by symbolic names and the execution time assembly can be done by transferring ASCII data to the proper fields.

```
                00100 ;MAILING LIST PRINT LINE
                00110 ;
4000            00115         ORG    4000H
```

```
4000            00117 LABEL   EQU   $           ;START OF LABEL BUFFER
4000            00120 NAME    EQU   $           ;20 CHAR NAME HERE
0014            00130         DEFS  20          ;RESERVE 20
4014            00140 STREET  EQU   $           ;22 CHAR STREET HERE
0016            00150         DEFS  22          ;RESERVE 22
402A            00160 CITY    EQU   $           ;15 CHAR CITY HERE
000F            00170         DEFS  15          ;RESERVE 15
4039            00180 STATE   EQU   $           ;2 CHAR STATE HERE
0002            00190         DEFS  2           ;RESERVE 2
403B            00200 ZIP     EQU   $           ;5 CHAR ZIP HERE
0005            00210         DEFS  5           ;RESERVE 5
4040 00         00220         DEFB  0           ;NULL TERMINATOR
0000            00230         END
00000 TOTAL ERRORS
ZIP     403B
STATE   4039
CITY    402A
STREET  4014
NAME    4000
LABEL   4000
```

String Input

When strings are input from either the TRS-80 keyboard or from another type of I/O device, an *input buffer* is allocated to hold the string of characters in much the same way as the output message buffer is defined at assembly time. The problem with input of strings is not how to detect the end of the string, but to limit the number of input characters so that the space allocated for the input buffer is not exceeded. In the code below, the subroutine INPUT is called to input one character from an external keyboard. INPUT handles all of the communication between the TRS-80 in regard to *status* and transmission of the character. The input text string in ASCII is stored into INMESS, starting at 4B00H. The INPTMS routine is exited when either a carriage return (0DH) or 64 characters has been input. Terminating the routine at 64 characters guarantees that the message buffer will not overflow, possibly overwriting program code adjacent to it.

```
                00100 ;MESSAGE INPUT ROUTINE
                00110 ;
4000            00120           ORG     4000H
4000 21104A     00130 INPTMS    LD      HL,INMESS       ;START OF INPUT BUFFER
4003 0640       00140           LD      B,64            ;MAXIMUM # OF CHARACTERS
4005 CD004B     00150 LOOP      CALL    INPUT           ;GET ONE CHARACTER
4008 FE0D       00160           CP      0DH             ;TEST FOR CARRIAGE RTN
400A C8         00170           RET     Z               ;RETURN IF CR
400B 77         00180           LD      (HL),A          ;STORE IN BUFFER
400C 23         00190           INC     HL              ;BUMP POINTER
400D 10F6       00200           DJNZ    LOOP            ;CONTINUE IF NOT 64
400F C9         00210           RET                     ;64 CHARACTERS
0040            00212 INMESS    DEFS    64
4B00            00213 INPUT     EQU     4B00H           ;TERMINAL INPUT
0000            00220           END
00000 TOTAL ERRORS
INPUT   4B00
LOOP    4005
INMESS  4010
INPTMS  4000
```

Once the string has been stored in the input buffer, of course, it must be separated into fields representing different types of data, as in the case of the mailing list line defined earlier. Conversion from ASCII data into decimal, hexadecimal, and other number representations must be performed. We've covered some of the conversion techniques for numbers earlier, but let us look at processing of the text strings that will be in the input message and may be carried through the entire processing of the program without being reformatted. The block move instructions allow shuffling of the strings from one place in memory to another, but the block search instructions perform an equally important task, comparison of one text string to another.

Block Compares

The block compare instructions, CPD, CPI, CPIR, and CPDR, search a block of memory (string) for a given char-

acter. If the character is found, the *location* of the character is returned. Since the search can be done in one instruction for the CPIR and CPDR, the search process is much faster on the Z-80 than on equivalent microprocessors. Let us see how the block compares operate. Suppose that we have just input a line of mailing list information using the INPTMS routine. The information input was in the format

JOHN J. PROGRAMMER/32768 OVERFLOW ST./COMPUTERTON/CA/92677

Here the fields of the mailing list information were separated by special characters called *delimiters,* which could have been any character normally not used in the text. To use the CPIR to search the input line for the next delimiter, the HL register pair is set up with the start of the message area, the BC register pair is set up with the number of bytes to be searched, and the A register is loaded with the character for which the search is to be done. The code below shows the initialization and the CPIR.

```
LD      HL,INMESS   ;INPUT MESSAGE START
LD      BC,64       ;64 CHARACTERS TO BE SCANNED
LD      A,'/'       ;SEARCH FOR SLASH
CPIR                ;PERFORM SEARCH
```

At the end of the search, the Z flag will be set if the character has been found, or reset if the character was not found in the entire block of memory. If the character *was* found, the HL register points to the location of the character *plus one,* and the HL register must, therefore, be decremented to point to the actual character. An actual example of this search would be the code below. Assemble and load using T-BUG, or key in using T-BUG, execute the program, and then display the registers using the R command. The Z flag should be set, and the HL register pair should contain 4A11H, the location of the slash plus one.

```
                00100 ;ROUTINE TO SEARCH FOR SLASH
                00110 ;
4000            00120       ORG     4000H
4000 21004A     00130 START LD      HL,INMESS     ;START OF MESSAGE AREA
4003 010600     00140       LD      BC,6          ;# OF CHARACTERS TO SCAN
4006 3E2F       00150       LD      A,'/'         ;SEARCH CHARACTER
4008 EDB1       00160       CPIR                  ;SEARCH
400A C30A40     00170 LOOP  JP      LOOP          ;LOOP HERE ON DONE
400D 31         00180 INMESS DEFM   '123/ME'      ;MESSAGE
```

```
4A0E 32      4A0F 33      4A10 2F      4A11 4D      4A12 45      0000
   00190        END
00000 TOTAL ERRORS
LOOP   4A0A
INMESS 4A00
START  4A00
```

The CPID works similarly to the CPIR, except that the CPID searches the string from end to beginning. In this case the HL register pair points to the character found minus one byte for the location. The HL register pair must be set up to the end of the string area in the CPID case.

```
LD      HL,INMESS+63   ;INPUT MESSAGE END
LD      BC,64          ;64 CHARACTERS TO BE SCANNED
LD      A,'/'          ;SEARCH FOR SLASH
CPDR                   ;SEARCH FOR SDRAWKCAB
JP      Z,FOUND        ;GO IF FOUND
...                    :NOT FOUND HERE
```

The CPI and CPD instructions require the same setup as the CPIR and CPID, respectively. They operate in similar fashion to the block move instructions in that only one iteration is done at a time. The instruction then pauses so that additional operations can be performed. Suppose, for example, we wished to search for two characters in the search. The following code would do that by a CPI-type search. After each iteration the Z flag would be set if the search character was found, and the P/V flag would be set if the byte count in BC was counted down to zero and the search was over. In this case, if the Z flag is set the first character was found and a check is made for the second character, as the HL register pair now points to a location one past the found character. If the second character does not match, then the search is continued until the end. Upon completion the HL register pair should point to 4A1EH in this case.

```
                00100 ;ROUTINE TO SEARCH FOR '/*'
                00110 ;
4A00            00120        ORG    4A00H
4A00 211B4A     00130 START  LD     HL,INMESS      ;START OF MESSAGE
4A03 010600     00140        LD     BC,6           ;# OF BYTES TO SEARCH
4A06 3E2F       00150 LOOPA  LD     A,'/'          ;SLASH FOR FIRST CHAR
4A08 EDA1       00160 LOOP   CPI                   ;SEARCH ONE BYTE
```

```
4A0A 2806     00170        JR     Z,MAYBE        ;GO IF FIRST FOUND
4A0C EA0B4A   00190        JP     PE,LOOP        ;GO IF NOT DONE
4A0F C30F4A   00190 LOOP1  JP     LOOP1          ;LOOP HERE NOT FOUND
4A12 3E2A     00200 MAYBE  LD     A,'*'          ;SECOND CHAR
4A14 BE       00210        CP     (HL)           ;COMPARE
4A15 C2064A   00220        JP     NZ,LOOPA       ;NO MATCH
4A18 C3184A   00230 LOOP2  JP     LOOP2          ;LOOP HERE IF FOUND
4A1B 23       00240 INMESS DEFM   '#$/*()'       ;MESSAGE
4A1C 24       4A1D 2F      4A1E 2A      4A1F 28      4A20 29      0000
   00250        END
00000 TOTAL ERRORS
LOOP2  4A18
LOOP1  4A0F
MAYBE  4A12
LOOP   4A0B
LOOPA  4A06
INMESS 4A1B
START  4A00
```

Searches for greater than one character may be done in this manner by searching for the first character using the search character in A for the CPI or CPD, and then searching the remainder of the string one byte at a time if there is a match on the first byte.

Table Searches

Tables are used extensively in all types of assembly-language programs. One of the simplest table types is a table of unordered or random data. The table is searched for a specific piece of data and the position in the table, or its *index*, is then used to access other information or simply as data itself.

Suppose, for example, that we have a table consisting of one-letter commands for T-BUG as shown in Figure 9-1. (In fact, this table is a kind of text string, as it is made up of ASCII characters.) We would like to see if we can find a given one letter command that has been input from the TRS-80 keyboard, match it up with a table entry, find the index, and then use that index to get the address of the routine to process that command in T-BUG.

The first thing that we must do is a table search, which in this case is exactly the same as the string search we performed under the string operations.

```
START   LD    HL,TABLE    ;TABLE START
        LD    BC,9        ;# OF BYTES
        LD    A,(INPUT)   ;GET INPUT CHARACTER
        CPIR              ;SEARCH
```

In the above code A was loaded with the input character from the keyboard, a one-letter ASCII command. At the end of the LDIR search Z will be set if the character was found and HL will then point to the character in the table plus one location. If the table is set up as in Figure 9-1, then HL will contain

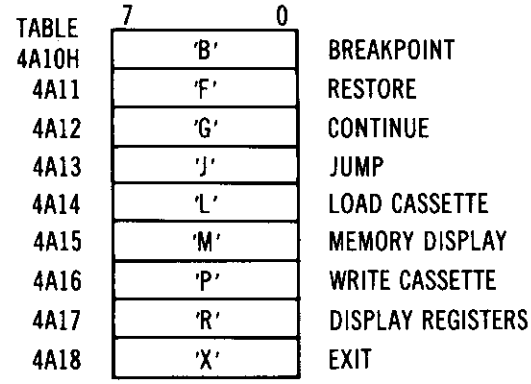


Fig. 9-1. Sample table of T-BUG commands.

location 4A11H through 4A19H if the character was found and location 4A19H (with zero reset) if the character was not found. We can find the *index* of the command in the table by subtracting the value of table from the value in HL if the character was found.

```
JP    NZ,NFND   ;GO IF CHARACTER NOT FOUND
LD    BC,TABLE  ;START OF TABLE
OR    A         ;CLEAR CARRY FOR SUBTRACT
SBC   HL,BC     ;FIND INDEX
```

At the end of the code above, L will contain the index of 1 through 9. If "INPUT" was a G, for example, L will contain a 3, indicating that G was the third entry in the table, counting from the *zeroth* entry. Now that we have the index, what do we do with it? Well, we can now use that index to *index into* another table of jumps corresponding to the routines that process each of the T-BUG commands. The relationships of the two tables are shown in Figure 9-2.

In the case of the first command table, the *entries* of the table were one byte long, each byte being an ASCII character representing the command. In the address table, however, each

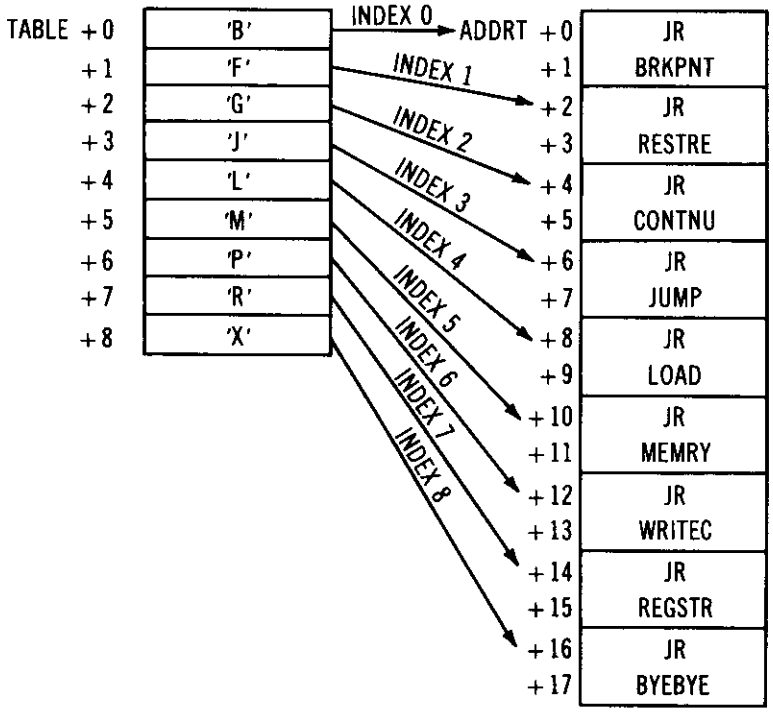


Fig. 9-2. Indexing into tables.

entry is two bytes long, since a relative jump must be represented. We now need to change that index from the first table into a *displacement* value that will pick up the right address table entry, the displacement being the number of physical bytes from the beginning of the address table. (The displacement for the first table was one times the index, but the displacement in the second table is two times the index.) The following code accomplishes this after first decrementing the index to adjust for the way the CPIR leaves the HL register.

```
DEC   HL          ;FIND TRUE INDEX
SLA   L           ;INDEX TIMES TWO
EX    DE,HL       ;SWAP DE AND HL
LD    HL,ADDRT    ;JUMP TABLE LOCATION
ADD   HL,DE       ;HL NOW HAS LOCATION OF JUMP
JP    (HL)        ;JUMP OUT TO JUMP
```

In the short tables here this code is not the most efficient (it took about 14 instructions to get to the routine), but the reader can see that this is a good approach for very long tables that are used in this fashion.

To recap the table structure, once again, a general table (see Figure 9-3) has a number of *entries*, each a certain *entry length*, and each having a displacement from the start of the table of entry length times # of entry.

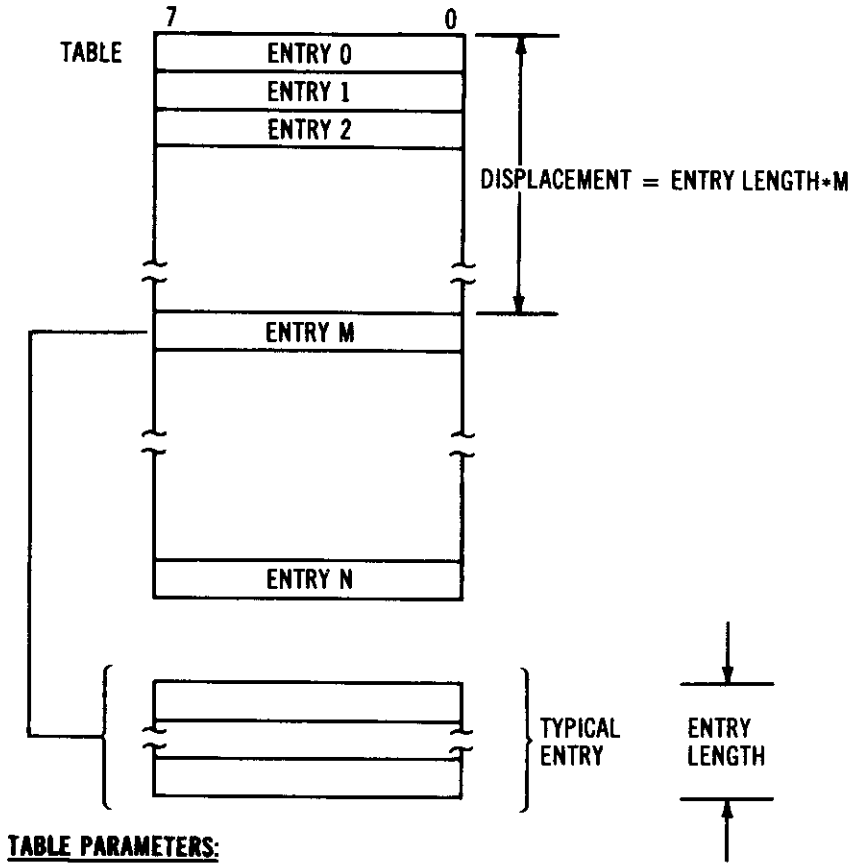


Fig. 9-3. General table structure.

Another method of using tables is to include the data associated with the *search key* in the entry itself, rather than in a separate table. Figure 9-4 shows this type of table. Each entry consists of a disc file name of 1 to 8 characters, a track number, and a sector number. The track and sector number always occupy the ninth and tenth bytes of each entry.

This table could be used to locate a specific *file* on disc by first searching the entire table for the correct file name, and then picking up the location of the file by the associated track and sector number when the file is found.

Unordered Tables

Tables in which the key entries are in random fashion are said to be unordered. When tables of this type are searched

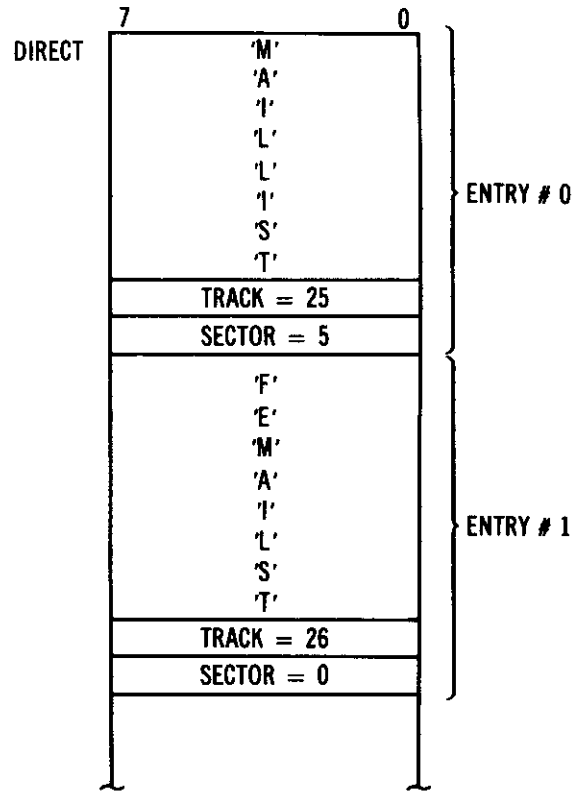


Fig. 9-4. Sample table of disc files.

for a specific entry, the minimum search occurs when the first entry is the desired entry and the maximum search occurs when the last entry is the one sought. The average number of entries that must be searched in this type of table is one-half the number of entries in the table. This type of table is fine for a small number of entries, but when the table must be continually searched and it holds a large number of entries, then a table with ordered entries could be used to a greater advantage.

The following program is another "standard" subroutine that the reader might find useful. It searches an unordered table from beginning to end for an 8-bit search key. Before the subroutine is called, A must be loaded with the search key, HL must be loaded with the start of the table, DE must be loaded with the length of each entry, and C must be loaded with the number of entries. If the entry is found, HL points to the entry upon return and the Z flag is set. If the entry is not found, the Z flag is not set upon return. The key in each entry is assumed to be the first byte.

```
                00100 ;SUBROUTINE FOR TABLE SEARCH
                00110 ;   ENTRY:(A)=KEY
                00120 ;         (HL)=TABLE START
                00130 ;         (DE)=LENGTH OF EACH ENTRY IN BYTES
                00140 ;         (C)=# OF ENTRIES IN TABLE
                00150 ;         CALL  SEARCH
                00160 ;   EXIT: Z FLAG SET IF FOUND, NOT SET IF NOT FOUND
                00170 ;         (HL)=LOCATION OF MATCH IF FOUND
                00180 ;         (BC)=CURRENT # LEFT
                00190 ;         (DE)=UNCHANGED
                00200 ;
4A00            00210         ORG     4A00H         ;CHANGE ON REASSEMBLY
4A00 0600       00220 SEARCH  LD      B,0           ;BC NOW HAS #
4A02 EDA1       00230 LOOP    CPI                   ;COMPARE A WITH (HL)
4A04 CA0E4A     00240         JP      Z,FOUND       ;GO IF FOUND
4A07 E20F4A     00250         JP      PO,NFND       ;AT END AND NOT FND
4A0A 19         00260         ADD     HL,DE         ;CURRENT+LENGTH+1
4A0B 2B         00270         DEC     HL            ;CURRENT +LENGTH
4A0C 18F4       00280         JR      LOOP          ;TRY AGAIN
4A0E 2B         00290 FOUND   DEC     HL            ;ADJUST TO FOUND LOC
4A0F C9         00300 NFND    RET                   ;RETURN
0000            00310         END
00000 TOTAL ERRORS
NFND    4A0F
FOUND   4A0E
LOOP    4A02
SEARCH  4A00
```

Ordered Tables

Tables may be ordered in many different ways. The order may be *ascending* as in the sequence 1,3,5,6,7,10, . . . or descending as in the sequence 101,99,97,5,1,0. The keys used for ordering may be one byte or larger straight numeric values, or ASCII text strings. Tables that are ordered invariably require that new data must be merged into the existing order, existing entries deleted or modified, or that the entries

should be resorted. There have been literally thousands of books and articles written about the problems and approaches of sorting (ordering data), searching (finding data), and merging (merging in new data), and we may not cover all of it in this chapter. We will present *one* of the approaches to ordering data in a *list* of items, the *bubble sort*. Becoming familiar with the 16,387 other methods will be left up to the reader as an exercise.

The bubble sort orders data by comparing each entry in a list with the next entry of the list. If the next entry is a lower value, then the two entries are swapped. The next entry is then compared, and so on, until the end of the list is reached. If there has been *at least* one set of items swapped during the search of the list, then another pass is made, starting from the beginning. Passes continue until there have been no swaps made during the last pass, signifying that the list has been ordered. The code for the sort takes advantage of the indexing capability to swap the items, and is shown below.

```
                00100 ;BUBBLE SORT
                00110 ;
4000            00120           ORG     4000H
4000 DD212B40   00130 LOOP      LD      IX,TABLE        ;TABLE START
4004 060F       00150           LD      B,15            ;NO OF LINES
4006 0E00       00160           LD      C,0             ;CHANGE FLAG
4008 DD7E00     00170 LOOP1     LD      A,(IX)          ;GET ENTRY
400B DD6E01     00180           CP      (IX+1)          ;TEST NEXT
400E CA1F40     00185           JP      Z,NOSWAP        ;GO IF EQUAL
4011 DA1F40     00190           JP      C,NOSWAP        ;GO IF NEXT LARGER
4014 DD4E01     00200           LD      C,(IX+1)        ;GET NEXT TO C
4017 DD7701     00210           LD      (IX+1),A        ;STORE CURRENT
401A DD7100     00220           LD      (IX),C          ;STORE NEXT
401D 0E01       00270           LD      C,1             ;SET CHANGE FLAG
401F DD23       00280 NOSWAP    INC     IX              ;POINT TO NEXT
4021 10E5       00290           DJNZ    LOOP1           ;DECREMENT LN CNT
4023 CB41       00300           BIT     0,C             ;TEST CHANGE
4025 C20440     00310           JP      NZ,LOOP         ;GO IF CHANGE
4028 C32840     00320 LOOP2     JP      LOOP2           ;DONE HERE
402B            00325 TABLE     EQU     $               ;PUT 16 ITEMS HERE
```

```
0000          00C30         END
00000 TOTAL ERRORS
LOOP2   4A28
NOSWAP  4A1F
LOOP1   4A08
TABLE   4A2B
LOOP    4A00
```

Assemble and load the program using T-BUG, or key in the program using T-BUG. TABLE can be filled with any number of data items that the reader desires, in any order. When a breakpoint at LOOP2 is reached, the table will have been reordered so that it is in ascending order, and the bubbles will have done an effective job in cleaning some of that RAM memory area. The reader may wish to breakpoint at the JP NZ,LOOP before LOOP2 to investigate the intermediate sorting after each pass. Use an "F" command and a "G" after looking at the table data, if breakpointing.

For another display of the bubble sort, use the program below. First use the M command in T-BUG to fill screen memory locations 3C20, 3C60, 3CA0, 3CE0, 3D20, 3D60 . . . 3FE0 with alphabetic or other characters in random order. A suggested sequence is shown in Table 9-1. You will see the characters appear in the middle of the screen as you fill them in. Now run the program, and you will see a literal graphic display of the bubble sort implementation.

```
                00100 ;BUBBLE SORT TO DISPLAY
                00110 ;
4A00            00120         ORG     4A00H
4A00 DD21203C   00130 LOOP    LD      IX,3C00H+32     ;FIRST LINE, MIDDLE
4A04 114000     00140         LD      DE,64           ;LINE INCREMENT
4A07 060F       00150         LD      B,15            ;NO OF LINES
4A09 0E00       00160         LD      C,0             ;CHANGE FLAG
4A0B DD7E00     00170 LOOP1   LD      A,(IX)          ;GET ENTRY
4A0E DDBE40     00180         CP      (IX+64)         ;TEST NEXT
4A11 CA2B4A     00185         JP      Z,NOSWAP        ;GO IF EQUAL
4A14 D22B4A     00190         JP      C,NOSWAP        ;GO IF NEXT LARGER
```

```
4A17 DD4E40   00200        LD     C,(IX+64)     ;GET NEXT TO C
4A1A DD7740   00210        LD     (IX+64),A     ;STORE CURRENT
4A1D DD7100   00220        LD     (IX),C        ;STORE NEXT
4A20 210000   00230        LD     HL,0          ;DELAY
4A23 23       00240 LOOPD  INC    HL
4A24 CB7C     00250        BIT    7,H           ;TEST FOR COUNTDOWN
4A26 CA234A   00260        JP     Z,LOOPD       ;GO FOR DELAY
4A29 0E01     00270        LD     C,1           ;SET CHANGE FLAG
4A2B DD19     00280 NOSWAP ADD    IX,DE         ;POINT TO NEXT LN
4A2D 10DC     00290        DJNZ   LOOP1         ;DECREMENT LN CNT
4A2F CB41     00300        BIT    0,C           ;TEST CHANGE
4A31 C2004A   00310        JP     NZ,LOOP       ;GO IF CHANGE
4A34 C3344A   00320 LOOP2  JP     LOOP2         ;DONE HERE
0000          00330        END
00000 TOTAL ERRORS
LOOP2   4A34
LOOPD   4A23
NOSWAP  4A2B
LOOP1   4A0B
LOOP    4A00
```

Table 9-1. Bubble Sort Sample Data

Display Memory Location	Contents
3C20H	46H
3C60	45
3CA0	44
3CE0	43
3D20	42
3D60	41
3DA0	39
3DE0	38
3E20	37
3E60	36
3EA0	35
3EE0	34
3F20	33
3F60	32
3FA0	31
3FE0	30

CHAPTER 10

I/O Operations

In this chapter we will rush in where many programmers fear to tread and describe some simple I/O operations in the TRS-80. I/O programming is intimately tied to the hardware configuration of a system, and for that reason some people are somewhat afraid of it, but we hope that the reader will find at the end of the chapter that it is really not that difficult. To lay the groundwork to discuss I/O programming we will review the *memory and I/O* mapping of the TRS-80. Then we will discuss the keyboard, display, cassette, and real-world applications, such as controlling the lawn sprinklers or your electric toothbrush.

Memory Versus I/O

In the first part of the book we talked somewhat about the architecture of the TRS-80. We mentioned that the TRS-80 has 64K or 65,536 bytes of memory available to it and explained how the memory was broken down into ROM, dedicated I/O addresses, and RAM as shown in Figure 10-1. The area that we will be considering in this chapter will be the central area of the figure, the dedicated I/O addresses, together with 256 I/O *ports*.

Let us expand that dedicated I/O address area and see what I/O devices are involved. Figure 10-2 shows that most of the area is devoted to display memory. Anytime that locations 3C00H through 3FFFH are addressed we are communicating with display memory, and that memory looks very similar to

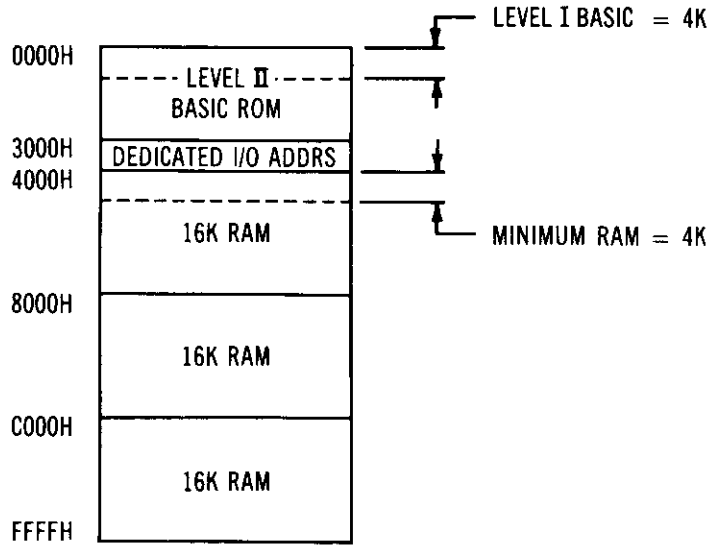


Fig. 10-1. Memory mapping with I/O addresses.

other RAM. We have been using display memory for many of the programs in previous chapters, and the reader should be very familiar with display memory at this point.

The section of dedicated memory from 3800H through 3BFFH is devoted to *keyboard addressing*. In this area memory does not exist, as it does for the display. When a location in this area is addressed, the keys of the TRS-80 keyboard are actually addressed. Addressing location 3801H addresses

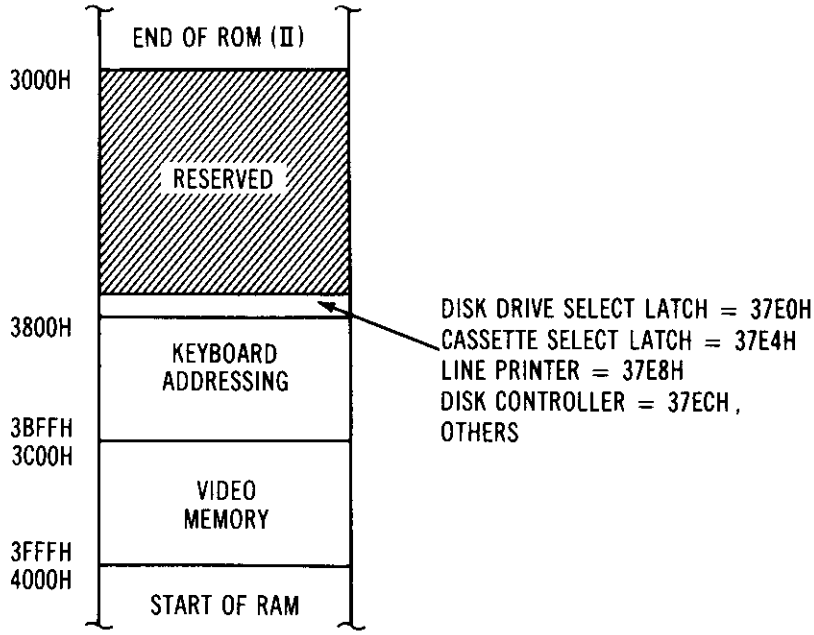
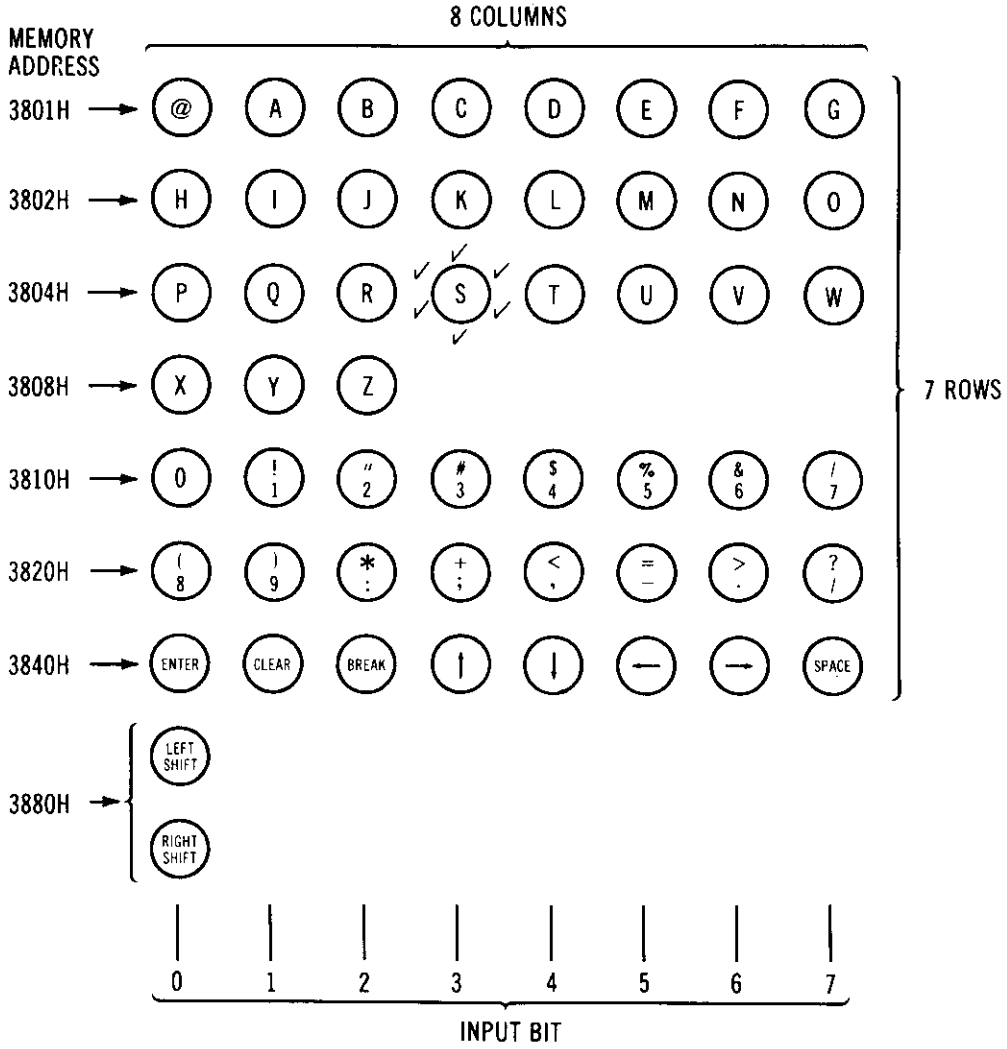


Fig. 10-2. Dedicated memory addresses.

the first row of keys, from "@" to "G", addressing location 3802H addresses the second row of keys from "H" to "O," and so forth, as shown in Figure 10-3. It turns out that there are eight addresses that address the keyboard, and they are 3801H, 3802H, 3804H, 3808H, 3810H, 3820H, 3840H, and 3880H. Every time a load is performed with one of these addresses 8 bits from the columns are loaded into the cpu register, as shown in Figure 10-3. These bits represent keys being pressed (1 bit) or not pressed (0 bit). We will discuss keyboard I/O a little later.



EXAMPLE: IF "S" IS PRESSED INPUT BYTE WILL BE 08H FOR ADDRESSING LOCATION 3804H. ALL OTHER INPUTS WILL YIELD 00H FOR INPUT BYTE.

Fig. 10-3. Keyboard addressing.

The remaining area of the dedicated memory addresses are used for such things as the line printer, floppy disc controller, and cassette select. Most of this area is reserved for future use (3000H through 37DDH). Addressing locations in the addresses above 37DDH enable communications with appropriate I/O devices. Loading a register from "memory" location 37E8H, for example, actually loads the register with eight bits of *status* for the system line printer, if one is attached. The status is a byte that is transmitted by the line printer that indicates whether the line printer is *ready* for the next character, whether it is *on-line*, and whether it has enough paper. Storing a register to location 37E8H actually transmits a byte of data, assumed to be an ASCII character, to the line printer for printing, in exactly the way a character is sent to a normal memory location to be stored.

For all intents and purposes, then, there is no practical difference in addressing a memory location in RAM or display memory and addressing an I/O device, as long as the I/O device is connected in such a manner as to look for that address and respond in the same manner that a memory location would respond.

Along with the memory addressing area devoted to system I/O devices, the TRS-80 has 256 other addresses that are devoted to I/O. These are the addresses used when an I/O instruction is executed. They differ from a memory address in that a signal goes out to all parts of the system that essentially says "here is an *I/O* address of 00000000 through 11111111." That signal is *not* present when a memory address is used (instead another signal goes out that says "here is a memory address of 16 bits").

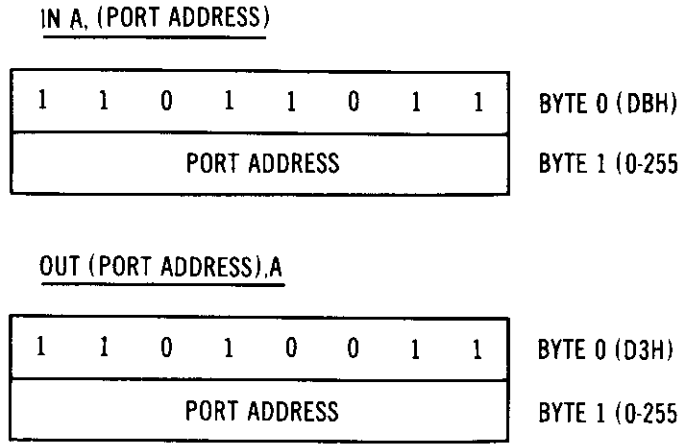


Fig. 10-4. I/O instruction format.

The general form of the I/O instruction is shown in Figure 10-4. There are several other formats, but we will be using these two in the rest of this chapter. The second byte of the instruction is the *port address* of 0 through 255. When an OUT instruction is executed, 8 bits of data from the A register are sent out to the system along with a signal that says "here is an I/O address" and the actual 8 bits of the port address itself. In a large system there could be many devices attached to the system *bus* (collection of data, address, and control signals), and they would all be continually looking for the I/O signal, *their* unique address (one of the 256), and the data to be received (or sent). See Figure 10-5.

In most configurations of the TRS-80, the only device that is attached in this port fashion is the cassette recorder. Logic on the cpu board is continually looking for port address FFH and the I/O signal indicating that an I/O instruction is being executed. If the instruction is an input (IN), the cassette

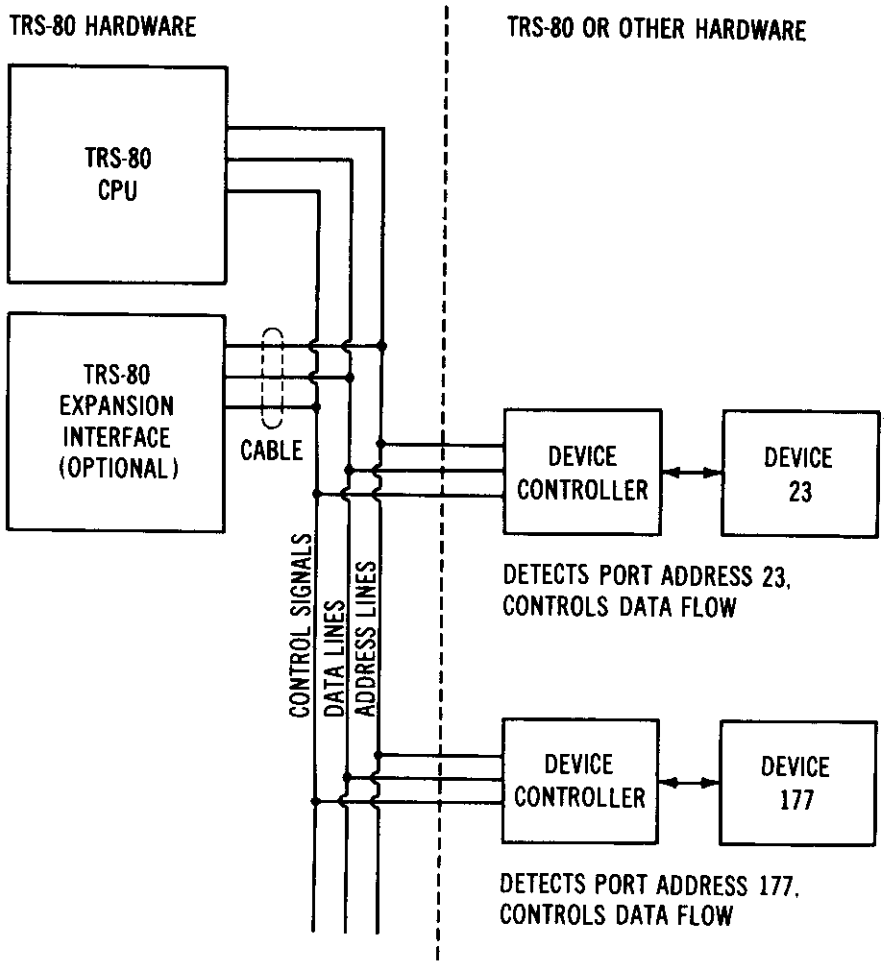


Fig. 10-5. I/O ports and port addressing.

logic will send a byte of data to the A register. Seven of those bits will be zeros, with only the most significant bit being active. If the instruction is an output (OUT), the contents of the A register will be sent to the cassette logic. Only the four least significant bits will cause actions in the logic.

The TRS-80 is expandable so that additional ports can be used by external devices, as long as the port addresses do not conflict with FFH or other port addresses used by TRS-80 devices. Since there are 256 total port addresses, however, there is a great deal of room for expansion, and conceivably the TRS-80 could be used to control dozens of functions such as home heating and lighting, burglar alarms, and others limited only by the user's imagination (and bank account).

Keyboard Decoding

Refer back to Figure 10-3. The keyboard is set up in eight rows and eight columns as shown in the figure. If a key is pressed, then the corresponding bit for that column becomes a one, and if the associated row address is read by a load instruction, then the column *byte* that is loaded will contain a one bit for the column of the key. As the program knows which row of the eight is being addressed when the one bit appears, it knows the key junction from the row and column. This type of I/O operation is called *matrix* decoding as the keyboard forms an eight-by-eight matrix.

The following program continually *scans* the keyboard and waits for a one bit to appear for the first and second rows (characters @, A through 0). When a one does appear, the row and column is computed to give an index of 0 through 15. This index is then used to *look up* the corresponding character in a sixteen-byte *look-up table.* The character is then printed on the screen.

```
                00100 ;KEYBOARD SCAN ROUTINE FOR FIRST TWO ROWS
                00110 ;
4000            00120       ORG     4A00H
4000 0E00       00130 KEYSCN LD     C,0         ;FOR FIRST ROW
4002 3A0138     00140       LD      A,(3801H)   ;1ST ROW ADDRESS
4005 B7         00150       OR      A           ;TEST ZERO OR NON-ZERO
4006 C2124A     00160       JP      NZ,KEY10    ;GO IF KEY PRESSED
4009 3A0238     00170       LD      A,(3802H)   ;2ND ROW ADDRESS
400C B7         00180       OR      A           ;TEST ZERO OR NON-ZERO
```

```
4A00 C2004A   00190        JP     Z,KEYSCN        ;GO IF NO KEY
4A10 0E08     00200        LD     C,8             ;FOR 2ND ROW
4A12 06FF     00210 KEY10  LD     B,0FFH          ;INDEX
4A14 04       00220 KEY20  INC    B               ;DECREMENT INDEX
4A15 CB3F     00230        SRL    A               ;SHIFT 'TIL ZERO
4A17 C2144A   00240        JP     NZ,KEY20        ;GO IF NOT ZERO
4A1A 78       00250        LD     A,B             ;GET # 0-7
4A1B 81       00260        ADD    A,C             ;ADD ROW #
4A1C 4F       00270        LD     C,A             ;INDEX 0-15 TO C
4A1D 0600     00280        LD     B,0             ;ZERO B FOR ADDR
4A1F 21344A   00290        LD     HL,TABLE        ;TABLE OF CHARACTERS
4A22 09       00300        ADD    HL,BC           ;COMPUTE DISPLACEMENT
4A23 7E       00310        LD     A,(HL)          ;GET CHARACTER
4A24 32203E   00320        LD     (3C00H+512+32),A  ;DISPLAY
4A27 0E09     00330        LD     C,10
4A29 0600     00340 LOOP   LD     B,0             ;DELAY ABOUT 17 MILLISEC
4A2B 10FE     00350 LOOP1  DJNZ   LOOP1
4A2D 0D       00360        DEC    C
4A2E C2294A   00370        JP     NZ,LOOP
4A31 C3004A   00380        JP     KEYSCN          ;GO FOR NEXT KEY
4A34 40       00390 TABLE  DEFM   '@ABCDEFGHIJKLMNO'
4A35 41       4A36 42      4A37 43      4A38 44      4A39 45      4A3A 46
   4A3B 47      4A3C 48      4A3D 49      4A3E 4A      4A3F 4B      4A40 4
C      4A41 4D      4A42 4E      4A43 4F      0000         00400        EN
D
00000 TOTAL ERRORS
LOOP1   4A2B
LOOP    4A29
TABLE   4A34
KEY20   4A14
KEY10   4A12
KEYSCN  4A00
```

The A register is loaded with the contents of row 1 by addressing 3801H. If this is zero, the next row, 3802H, is addressed. If either row has at least one bit, the rest of the

program is executed, otherwise the program loops back to KEYSCN to scan the rows again. If a one bit has been detected, the C register holds either 0 for row one or 8 for row 2. The A register holds the column bit corresponding to the key column. As this is a power of two (80H, 40H, 20H, 10H, 8H, 4H, 2H, or 1H) it must be converted to a number representing the column of 0 through 7. This is done by shifting A until it becomes zero, and keeping a count of the number of shifts. 80H will require 7 shifts, for example, before A becomes 0. At the end of the shifting B holds the column number. This is added to the row number of 0 or 8 to produce an index of 0 through 15. This index is then added to the address of TABLE to point to the corresponding character in the table. This character is picked up and displayed on the center of the screen. LOOP is a *timing loop* to *debounce* the key so that the program does not loop back to the *same* key depression and output a spurious character (the same character twice or a number of times).

Although this program works only with the first two rows of keys, the reader can see how it can be expanded to work with *all* keys on the keyboard, and he will find a similar program in Level I or II BASIC.

Display Programming

We have used programs that output both ASCII and graphics characters to the screen, but have not discussed the graphics capabilities of the TRS-80 in any detail. The display memory is similar to normal RAM memory, except that each address of the 1024 bytes of display memory is made up of seven instead of eight bits, as shown in Figure 10-6. As the reader knows from his BASIC experiences, the display can display upper case alphanumeric and special characters *or* graphics characters, intermixed in any combination. The most significant bit of the 7-bit display memory is used to mark a graphics character. If this bit is a zero, then the remaining six bits define an alphanumeric or special character. If the most significant bit is a one, then the other six bits define a graphics character. The ASCII codes for alphanumeric and special characters are defined in the Editor/Assembler manual or the TRS-80 BASIC manual.

The graphics codes define a six-element graphics character that occupies one character position on the screen. As there are 1024 character positions (64 characters per line and 16 lines), there are 6144 graphics elements on the screen, ar-

ranged in a 128 by 48 matrix. The question arises of how one sets or resets a single element. There is no corresponding assembly-language SET or RESET command as there is in BASIC.

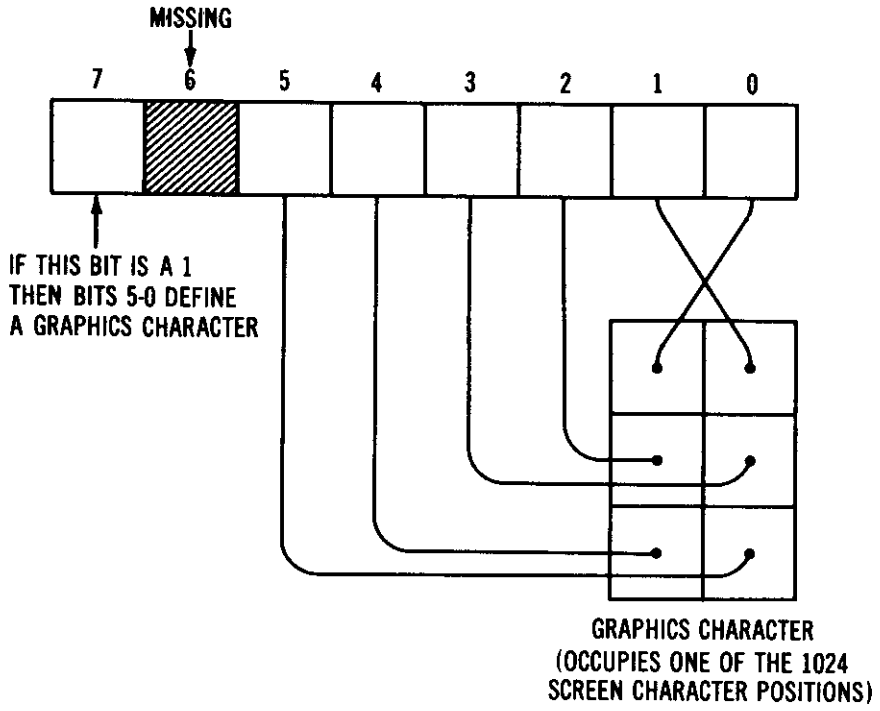


Fig. 10-6. Display memory format.

The following code attempts to solve the problem of converting an x,y coordinate into the proper bit position in the graphics memory cell. There are three entry points in the routine. The first entry point sets the *pixel* (element) corresponding to the given x,y (horizontal, vertical) position. The second entry point resets the pixel corresponding to the given x,y position, and the third entry point tests the current on/off status of the pixel, returning the zero/non-zero status in the zero flag. The three entry points of SET, RESET, and TEST all converge to a common location at TEST10. The store at TEST10 stores the second byte or a SET, RES, or BIT instruction at location INST+1. The first byte of all three instructions are the same, a CBH. The second byte is complete except for a three-bit field defining the bit to be set, reset, or tested. This will be calculated in the main body of the routine, along with the location in screen memory to be used, which will be put into HL. All three instructions use HL as a register pointer. See Figure 10-7.

The main body of the code converts an x,y location into a screen memory location and bit position. The bit position is merged into INST+1 to set the proper field. The memory location is retained in HL for the instruction. The actual *algorithm* works like this: The y position of 0-47 is converted to a line number by dividing by 3 to give 0 through 15. The remainder is saved. The x position is divided by 2 to give the character position along the line. We now have a line number of 0 through 15 and a character position of 0 through 63. If the line number is multiplied by 64 and the character position added to it, we will have the *byte displacement* from the start of screen memory, as shown in Figure 10-8. The actual location can then be found by adding 3C00H, the start of display memory.

The only remaining task is to find the bit position of the pixel to be set, reset, or tested. This is given by the remainder of the Y/3 operation times 2 plus the remainder of the X/2 operation. This value is stored in the bit position field of the instruction at INST+1. As a last step, bit 7 is set to ensure that all character positions processed will be graphics characters.

The code for this problem is somewhat complex and it may help the reader to "play computer" by actually using some values of x and y and working through the routine to find

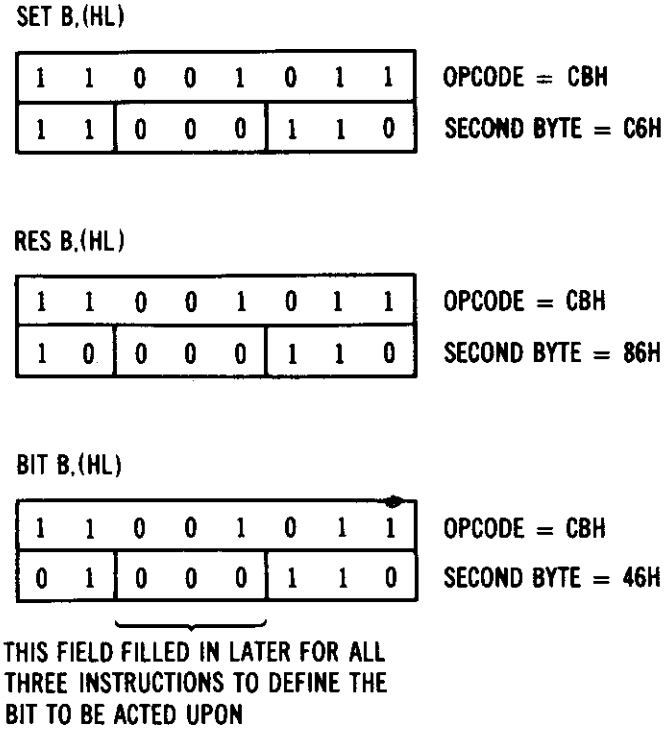
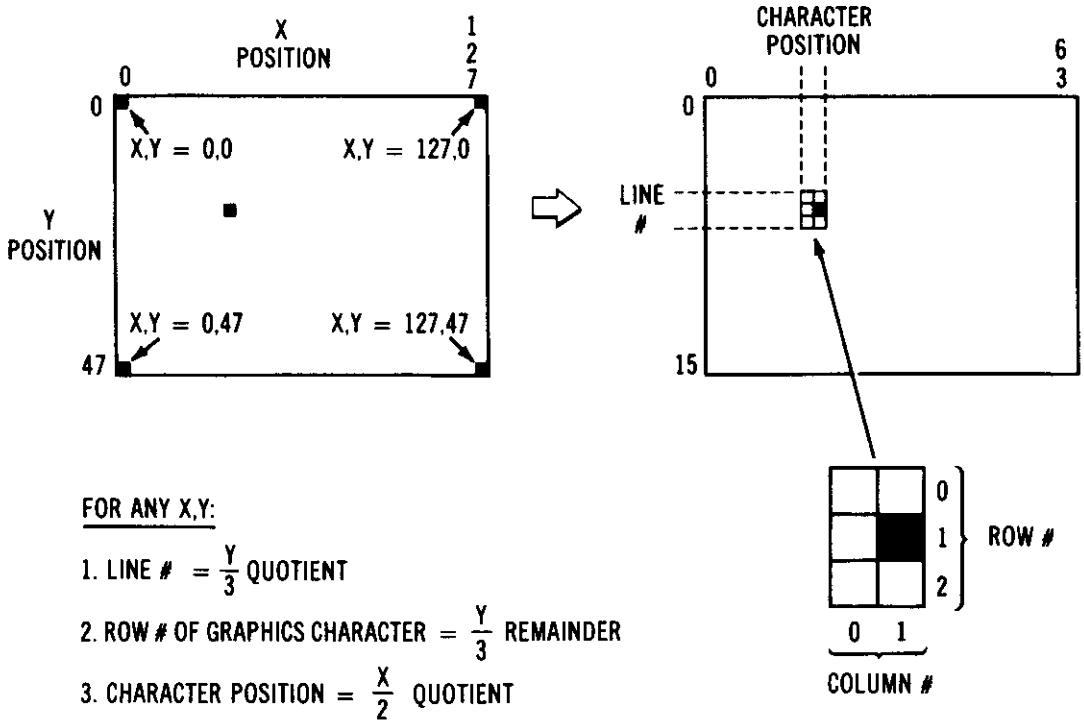


Fig. 10-7. Modifying instructions.



FOR ANY X,Y:

1. LINE # = $\frac{Y}{3}$ QUOTIENT
2. ROW # OF GRAPHICS CHARACTER = $\frac{Y}{3}$ REMAINDER
3. CHARACTER POSITION = $\frac{X}{2}$ QUOTIENT
4. COLUMN # OF GRAPHICS CHARACTER = $\frac{X}{2}$ REMAINDER
5. BYTE DISPLACEMENT FROM START OF SCREEN MEMORY = (LINE #) *64 + CHARACTER POSITION
6. ACTUAL LOCATION IN MEMORY = (LINE #) *64 + CHARACTER POSITION + 3C00H
7. BIT POSITION WITHIN GRAPHICS CHARACTER = (ROW #) *2 + COLUMN NUMBER

Fig. 10-8. Screen coordinate algorithm.

out how it works. An interesting point is that the instruction at INST has been treated as another piece of data to be processed and modified. It is not a good practice to do this in some types of programming (for example, where interrupts are involved), but it is perfectly permissible in many *stand-alone* programs of this type.

```
00100 ;SUBROUTINE TO CONVERT SCREEN COORDINATES
00110 ;   ENTRY:(DE)=Y,X COORDINATES OF POINT
00120 ;              X=0 TO 127;Y=0 TO 47
00130 ;         CALL   SET       ;SETS POINT
00140 ;         CALL   RESET     ;RESETS POINT
00150 ;         CALL   TEST      ;TESTS POINT RETURNS Z FLAG
00160 ;   EXIT: (A THROUGH L)=DESTROYED
```

```
                00170 ;         Z FLAG SET IF TEST
                00180 ;
4000            00190           ORG     4000H
4000 3EC6       00200 SET       LD      A,0C6H          ;SET B,(HL) INSTRUCTION
4002 1806       00210           JR      TEST10          ;GO TO STORE
4004 3E86       00220 RESET     LD      A,86H           ;RES B,(HL) INSTRUCTION
4006 1802       00230           JR      TEST10          ;GO TO STORE
4008 3E46       00240 TEST      LD      A,46H           ;BIT B,(HL) INSTRUCTION
400A 32CD4A     00250 TEST10    LD      (INST+1),A      ;STORE 2ND BYTE
400D 7A         00260 ADDRES    LD      A,D             ;GET Y
400E 06FF       00270           LD      B,0FFH          ;-1
4010 04         00280 LOOP      INC     B               ;SUCCESIVE SUBS FOR DIV
4011 D603       00290           SUB     3               ;  BY THREE
4013 F21040     00300           JP      P,LOOP          ;GO IF NOT MINUS
4016 C603       00310           ADD     A,3             ;YQ IN B, YR IN A
4018 CB27       00320           SLA     A               ;YR*2
401A 4F         00330           LD      C,A             ;SAVE YR*2
401B 68         00340           LD      L,B             ;YQ TO L
401C 2600       00350           LD      H,0             ;YQ IN HL
401E 0606       00360           LD      B,6             ;CNT FOR MULTIPLY BY 64
4020 29         00370 LOOP1     ADD     HL,HL           ;YQ*2
4021 10FD       00380           DJNZ    LOOP1           ;GO IF NOT YQ*64
4023 1600       00390           LD      D,0             ;DE NOW HAS X
4025 CB3B       00400           SRL     E               ;XQ
4027 3001       00410           JR      NC,CONT         ;GO IF XR NE 1
4029 0C         00420           INC     C               ;C NOW HAS YR*2+XR
402A 19         00430 CONT      ADD     HL,DE           ;HL NOW HAS YQ*2+XQ
402B 11003C     00440           LD      DE,3C00H        ;START OF DISPLAY
402E 19         00450           ADD     HL,DE           ;HL NOW HAS DISPLACE
402F CB21       00460           SLA     C               ;ALIGN TO FIELD
4031 CB21       00470           SLA     C
4033 CB21       00480           SLA     C
4035 3ACD4A     00490           LD      A,(INST+1)      ;GET INSTRUCTION
4038 81         00500           ADD     A,C             ;SET FIELD
```

```
403F 32204A   00510        LD      (INST+1),A      ;STORE
4042 CB       00520 INST   DEFB    0CBH            ;PERFORM BIT,SET,RES
4043 00       00530        DEFB    0               ;WILL BE FILLED IN
4044 CBFE     00540        SET     7,(HL)          ;FOR GRAPHICS
4046 C9       00550        RET
0000          00560        END
00000 TOTAL ERRORS
CONT    4052
LOOP1   4020
LOOP    4010
ADDRES  4060
INST    4043
TEST    4008
RESET   4004
TEST10  400A
SET     4000
```

Mysteries of the Cassette Revealed

The cassette of a one cassette system is controlled by three bits of a 4-bit *latch* in the cpu. The latch is simply another type of memory, which happens to be four bits wide instead of the usual eight. When the cassette is addressed by performing an OUT instruction to port address 0FFH, the cassette latch is loaded with four bits of data as shown in Figure 10-9.

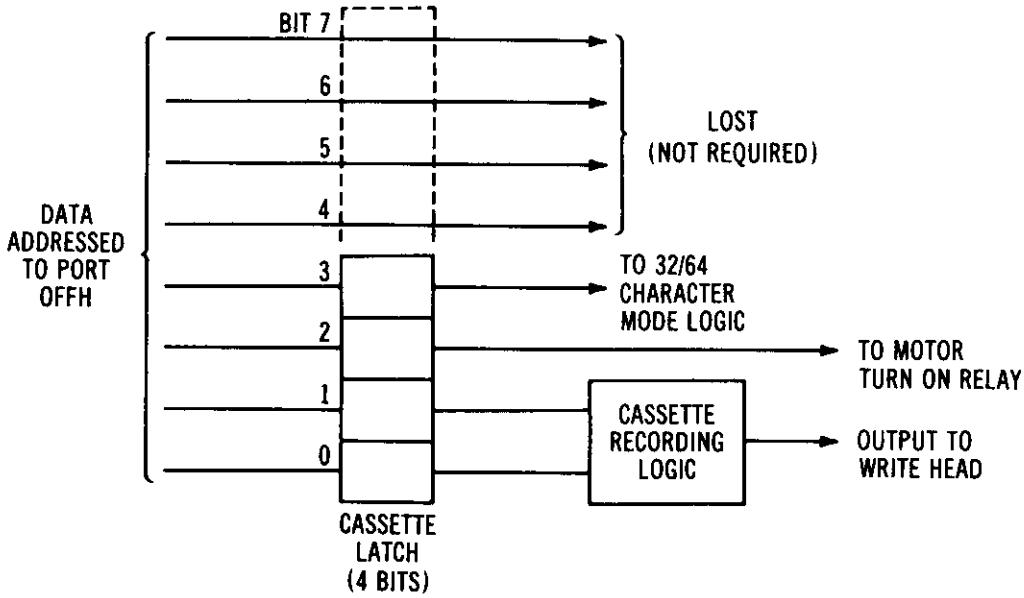


Fig. 10-9. Cassette latch transfers.

The other four bits of data, bits 7 through 4 are discarded into the bit bucket on the floor near the TRS-80.

Bit number 3 of the latch controls the 32- and 64-character mode of the TRS-80. Outputting a one to this bit will set the display into 32 character mode; outputting a zero will reset the display into the normal 64-character mode.

```
LD    A,8       ;BIT 3 IS SET
OUT   0FFH,A    ;SET 32-CHAR MODE
```

Bit number 2 of the cassette latch is the cassette motor on/off bit. Setting this bit by an OUT 0FFH will turn the cassette motor on, and resetting the bit will turn the cassette motor off. This action is produced by a *small* relay in the TRS-80 cpu, and it would be wise to quench all thoughts about controlling that four-ton air conditioner with this one small control device!

```
LD    A,4       ;BIT 2 IS SET
OUT   0FFH,A    ;SET MOTOR ON
```

Bits number 1 and 0 in the cassette latch are used to write data to the cassette tape. As you probably know from reading your TRS-80 Technical Reference Handbook, data on cassette is arranged serially, and everything is represented by a stream of bits. In the implementation on the TRS-80, cassette data is written by setting bit 0 of the cassette latch, then by setting

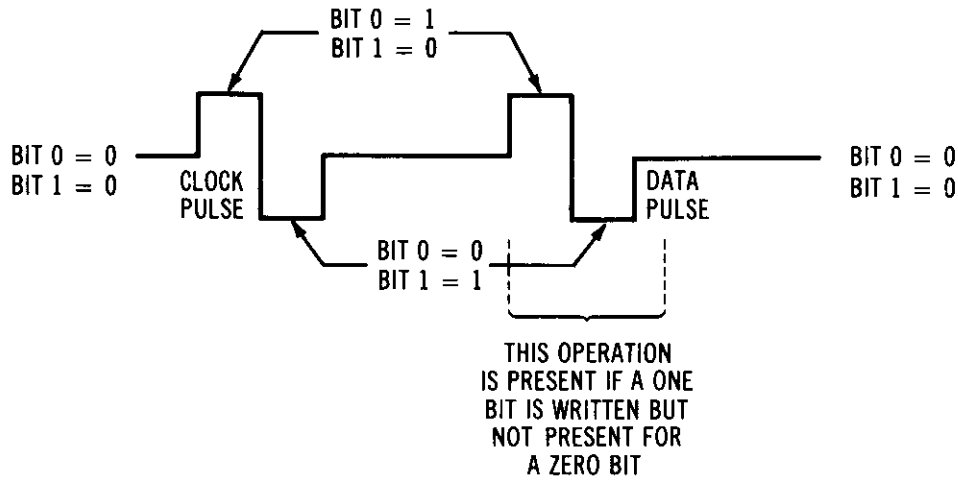


Fig. 10-10. Cassette data waveform.

bit 1 of the cassette latch, and then by resetting both bits. When this is done for both a *clock pulse* and *data pulse* the waveform appears as shown in Figure 10-10.

To illustrate how this works, let us write a program to record some music on cassette. It might be nice to try a little Bach or Beethoven, but perhaps we'll try something a little simpler. First of all, it is necessary to know how to produce *any* tone on the cassette. A simple tone has the appearance of the *sine* wave of Figure 10-11. We can produce a *square wave* on the cassette by turning the cassette output bits on and off rapidly as shown in the figure.

We know how to turn the cassette signal to the recording head on (01) and off (10), but what about the time delay to produce the tone? If we look in the Editor/Assembler Manual we find instruction times under "4MHZ E.T." This is the execution time in microseconds for a Z-80 microprocessor running at a *clock frequency* of 4 megahertz (4 million cycles

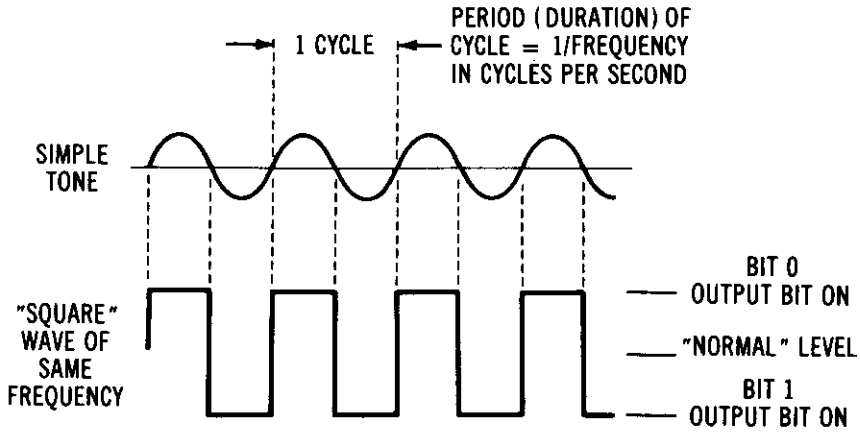


Fig. 10-11. Square wave tones.

per second). The TRS-80 clock frequency is about 1.774 megahertz, so to get the actual *execution times* of TRS-80 instructions we must multiply the 4 MHZ E.T. by 2.26. Let us see how long a simple loop would take. If we have a value of 1 through 255 in the B register, then the simple loop

```
LOOP  DJNZ  LOOP     LOOP HERE FOR 1 TO 255 TIMES
```

would take 3.25 microseconds (4 MHZ E.T.) * 2.255 * count in B, or 7.32 microseconds * count in B. This gives us a range of frequencies from about 535 Hz through 136,612 Hz. (The frequency of the tone can be found by dividing one by the time in microseconds, for example, 500 microseconds would produce a tone of 1/500E-06 or 2000 Hz.)

As the complete cycle would be determined by a timing delay to turn the write head on one direction and off the other, the actual tones that could be produced are 267 Hz through

68,306 Hz. If we stay on the lower end of that range we should be able to get a nice range of notes.

The routine to play a note with a given value in B follows:

```
PLAYN   LD      C,(DURTN)   ;GET DURATION
CONT    LD      B,(FREQ)    ;GET FREQUENCY
        LD      A,1
        OUT     (0FFH),A    ;TURN ON 1/2 CYCLE
LOOP1   DJNZ    LOOP1       ;DELAY FOR FREQUENCY
        LD      B,(FREQ)    ;GET FREQUENCY
        LD      A,2
        OUT     (0FFH),A    ;TURN ON OTHER 1/2 CYCLE
LOOP2   DJNZ    LOOP2       ;DELAY FOR FREQUENCY
        DEC     C           ;DECREMENT DURATION
        JP      NZ,CONT     ;CONTINUE IF NOT DONE
```

The additional count in C is used to adjust the length of time that the note plays. The value of D is related to the value of the frequency count to make all notes a quarter note duration, or approximately so (what did you expect, the New York Philharmonic?). The entire code required to play the TRS-80 concerto is given below. A table of delay values defines the duration and notes, and is terminated by a zero.

```
                00100 ;OPUS I THE TRS-80 CONCERTO
                00110 ;
4A00            00120       ORG     4A00H
4A00 DD21334A   00130 START LD      IX,TABLE     ;START OF MUSIC TABLE
4A04 DD4E00     00140 CONT1 LD      C,(IX)       ;DURATION
4A07 79         00150       LD      A,C          ;MOVE TO A FOR TEST
4A08 B7         00160       OR      A            ;TEST FOR 0
4A09 CA0D4A     00170 LOOP  JP      Z,LOOP       ;LOOP HERE ON DONE
4A0C DD4601     00180 CONT2 LD      B,(IX+1)     ;GET DELAY COUNT
4A0F 3E01       00190       LD      A,1
4A11 D3FF       00200       OUT     (0FFH),A     ;TURN ON 1/2 CYCLE
4A13 10FE       00210 LOOP1 DJNZ    LOOP1        ;DELAY FOR FREQ
4A15 DD4601     00220       LD      B,(IX+1)     ;GET DELAY AGAIN
4A18 3E02       00230       LD      A,2
4A1A D3FF       00240       OUT     (0FFH),A     ;TURN ON OTHER 1/2 CYCLE
4A1C 10FE       00250 LOOP2 DJNZ    LOOP2        ;DELAY FOR FREQ
4A1E 0D         00260 LOOP3 DEC     C            ;DECREMENT DURATION
4A1F C20C4A     00270       JP      NZ,CONT2     ;GO IF NOT DONE
```

```
4022 DD23      00290         INC     IX              ;POINT TO NEXT NOTE
4024 DD23      00290         INC     IX
4026 01FFFF    00300         LD      BC,-1           ;INCREMENT VALUE
4029 213000    00310         LD      HL,30H          ;INITIAL DELAY VALUE
402C 09        00320 LOOP4   ADD     HL,BC           ;DELAY FOR INTERVAL
402D DA2C4A    00330         JP      C,LOOP4         ;  BETWEEN NOTES
4030 C3044A    00340         JP      CONT1           ;CONTINUE
4033 A090      00350 TABLE   DEFW    90A0H           ;TABLE OF NOTES
4035 3FA2      00360         DEFW    0A23FH          ;EACH ENTRY IS MADE UP
4037 5DAC      00370         DEFW    0AC5DH          ;OF TWO BYTES. FIRST
4039 6090      00380         DEFW    9060H           ;BYTE IS DURATION OF
403B A090      00390         DEFW    9080H           ;NOTE. SECOND BYTE IS
403D 4090      00400         DEFW    9040H           ;FREQUENCY VALUE
403F 7080      00410         DEFW    8070H
4041 F090      00420         DEFW    90F0H
4043 5DA2      00430         DEFW    0A25DH
4045 5BAD      00440         DEFW    0AD5BH
4047 6090      00450         DEFW    9060H
4049 E06B      00460         DEFW    6BE0H
404B 485F      00470         DEFW    5F48H
404D FF54      00480         DEFW    54FFH
404F 00        00490         DEFB    0               ;TERMINATOR
0000           00500         END
00000 TOTAL ERRORS
LOOP4   402C
LOOP3   401E
LOOP2   401C
LOOP1   4013
CONT2   400C
LOOP    4009
CONT1   4004
TABLE   4033
START   4000
```

Arrangement compliments of James Garon

Real-World Interfacing

Is it possible to use the TRS-80 to control real-world events? An emphatic yes! But here's the catch. It does take some *hardware*. In this section, we will discuss how real-world control is done. We will be talking about some simple hardware, but you should find it interesting. (Just think about that TRS-80 controlled robot mowing the grass while you sleep in! But seriously . . .)

First of all, let us talk about what types of control can be provided to the external world with the TRS-80. Things externally are controlled by on/off conditions in a large number of cases. Such things as garage door openers, burglar alarms triggered by a switch being opened, sprinkler valves being turned on by a time switch—these are all events controlled by an on/off state. This class of functions can be controlled by *discrete* inputs and outputs to the TRS-80. One bit of an output or input can control or detect the operation, as only an on or off state is involved.

A second class of things in the external world are those events that are not controlled in binary fashion. The temperature of a room, windspeed, dampness of the soil, and lighting intensity are but a few items that have a range of values and cannot be represented by a single binary one or zero. These physical quantities require many bits to represent them, but they *can* be represented. There are many available devices that convert external world quantities into voltage, current, or resistance *analogs* that are then converted into binary form by an *analog-to-digital* converter. The resulting digital form, whether it is 8 bits or 24 can then be read into a computer such as a TRS-80 and processed.

Discrete Inputs

Suppose that we want to input a set of eight bits into the A register. These bits represent eight different discrete inputs that are either on or off. A good example would be a set of inputs from burglar alarm switches in eight rooms of a house. The bits are either a one (switch closed) or a zero (switch open), and we would like to read these eight inputs once a second or so to find out whether a switch that is normally closed is open, or a switch that is normally open is closed. How do we go about designing interface circuitry to do this, and what programming steps are required?

Earlier we discussed I/O ports. If we set up our burglar alarm inputs for a particular I/O port, then that port must have the following capability:

1. It must be able to recognize its address when it is sent over the system *address lines.*
2. It must be able to tell when an I/O instruction is being executed.

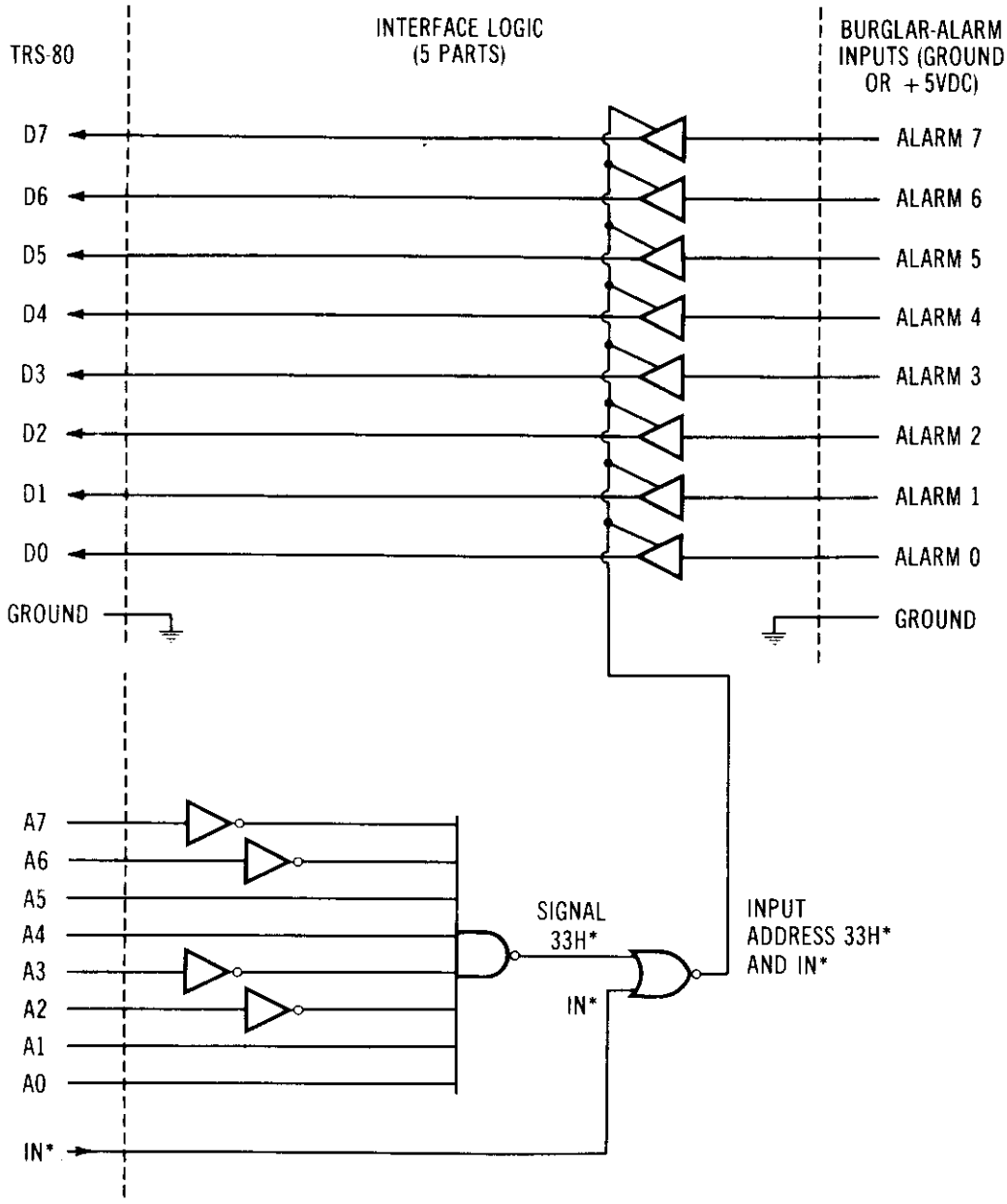


Fig. 10-12. Inputting external data.

3. It must be able to pass the eight bits of data to the cpu over the system *data lines.*

The circuitry for performing these tasks is shown in Figure 10-12. When signal RD* is active (this is the signal on pin 15 of the cpu or interface 40-pin connector), address lines A7 through A0 contain the port address from the IN instruction (address lines A7 through A0 are on various pins of the 40-pin connector). If, for example, we have defined the address of the port as 33H, executing an IN A,(33H) instruction would cause signal RD* to become active and simultaneously put 00110011 on address lines A7 through A0. The circuitry shown in the figure outputs a one for signal INPUT when both RD* and address 33H are present. This will only occur for the IN A,(33H) instruction. Signal INPUT allows the burglar alarm inputs to be *gated* (transmitted) from the eight lines onto the system data bus lines D7 through D0 (on various pins of the connector). During the execution of the IN A,(33H) the cpu will take the contents of the data bus and store it in the A register, completing the execution of the IN instruction. Now the data from the eight inputs can be processed, which might go something like this

```
LOOP  IN    A,(33H)   ;GET INPUTS
      XOR   0B3H      ;TEST 7,5,4,1,0 ON;6,3,2 OFF
      JP    NZ,HELP   ;GO IF BURGLAR
      JP    LOOP      ;TRY AGAIN
```

As the entire input, test, and loop takes under 20 millionths of a second (!) the constraint of one test every second is indeed met. As a matter of fact, there is more than enough time to do all kinds of other processing or control applications and still meet the *poll* of the burglar alarms every second.

This implementation is one of the more simple real-world applications. However, an output of discrete values is not much more complicated. The signal decoded in this case is analogous to the IN* signal, and, strangely enough, is called OUT*. The output operation works as follows: When signal OUT* is active a port address is present on the address bus lines A7 through A0. If the port address matches the built-in address of the hardware, then there is data for the port on data lines D7 through D0. If this is the case, the data lines are written into a memory *latch* similar to that used for the cassette. When the data disappears (it is only present for a few microseconds), the latch will retain the bit configuration and transmit it to the outside world. This circuitry is shown in Figure 10-13.

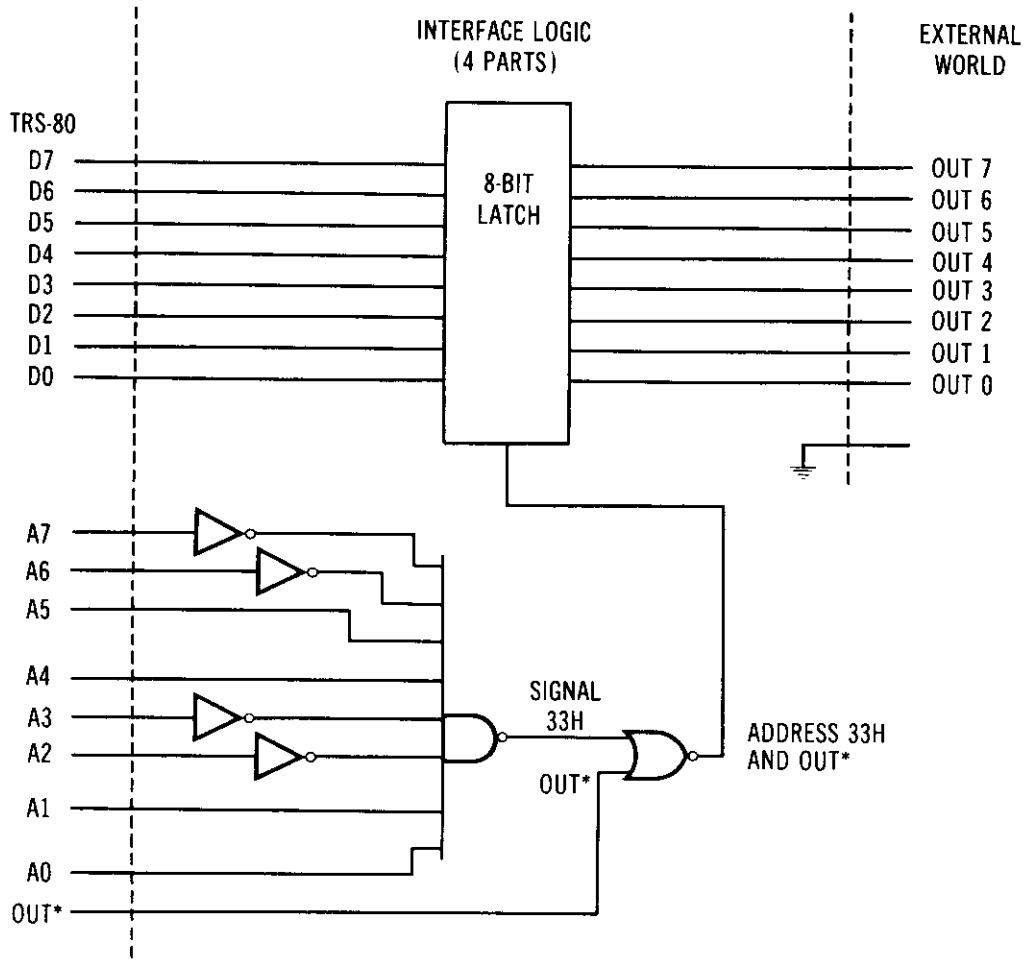


Fig. 10-13. Outputting data to the external world.

Suppose we have a set of lawn sprinkler valves that must be turned on at certain times. Location TIME holds the time in increments of one minute. The time at 12:00 noon is represented by a count of 720 in the two-byte variable. To turn sprinklers 3 and 5 on at noon, the interface in Figure 10-13 is used, along with the code below.

```
LD    HL,TIME     ;GET CURRENT TIME IN MINUTES
LD    BC,720      ;NOON
OR    A           ;RESET CARRY
SBC   HL,BC       ;TEST FOR NOON
JP    NZ,OTHER    ;CONTINUE WITH OTHER PROCESSING
LD    A,28H       ;SPRINKLERS 3 AND 5
OUT   (033H),A    ;TURN THEM ON
```

Another method for implementing discrete input and outputs involves a memory-mapped approach similar to that used for the line printer and other devices. Here IN and OUT instructions are not used, but the external logic is treated as

memory locations and loads and stores are used instead. This is also a valid approach but requires slightly different implementation. The problem of reading in other than discrete inputs or of outputting other than discrete bit patterns is similar to the methods described previously. The major consideration in this case is the conversion between analog values and digital 8-bit values required. The logic required for reading or writing the digital values between the cpu registers and the I/O port, however, is exactly the same as described above.

We hope that these very simple examples will give you some insight into the nature of external-world interfacing. With a little bit of external logic, the TRS-80 can indeed be used to control any number of things around the home or in industry, and with assembly-language programming, this control can be fast indeed.

CHAPTER 11

Common Subroutines

The subroutines presented in this section are the "common" subroutines described elsewhere in the book. Many of them are used continually in larger programs, and they are given here so that the reader may incorporate them into his own programs if he desires. All of them are *subroutines* and must be CALLed from the reader's code. All are assembled at 4A00H, and must be reassembled by incorporating the source code into the reader's source code, or by separate reassembly with a new *ORiGin*. A brief description of each routine is given below, with the assembly following.

FILL Subroutine

The FILL subroutine is used to fill a block of memory with a given 8-bit value. FILL could be used, for example, to zero a buffer, or to fill the video display area with blank characters. On entry, the D register contains the character to be filled, HL points to the start of the fill area, and BC contains the number of bytes to fill, 1 through 65535. An entry with BC=0 is treated as 65536 bytes to fill. On exit HL points to the last byte filled plus one, D is unchanged, and A and BC are zeroed.

```
0100 ;SUBROUTINE TO FILL DATA IN MEMORY
0110 ;   ENTRY:(D)=DATA TO BE FILLED
0120 ;         (HL)=START OF FILL AREA
0130 ;         (BC)=# OF BYTES TO FILL
```

```
                00140 ;         CALL   FILL
                00150 ;   EXIT: (D)=SAME
                00160 ;         (HL)=END OF FILL+1
                00170 ;         (BC)=0
                00180 ;         (A)=0
                00190 ;
4000            00200           ORG    4000H
4000 72         00210 FILL      LD     (HL),D          ;STORE BYTE
4001 23         00220           INC    HL              ;BUMP POINTER
4002 0B         00230           DEC    BC              ;ADJUST COUNT
4003 78         00240           LD     A,B             ;GET MS OF COUNT
4004 B1         00250           OR     C               ;MERGE LS COUNT
4005 20F9       00260           JR     NZ,FILL         ;CONTINUE IF DONE
4007 C9         00270           RET                    ;RETURN IF DONE
0000            00280           END
00000 TOTAL ERRORS
FILL    4000
```

MOVE Subroutine

The MOVE subroutine is used to move one block of memory to another area in memory. The blocks may be *overlapping* without conflict; the program is "smart" enough to calculate the direction of the move based on the type of overlap. On entry HL, DE, and BC are set up as in the block move instructions—HL points to the source block, DE points to the destination block, and BC holds the number of bytes to move. On exit, HL and DE point either to the block areas plus one, or to the block areas minus one, based on the direction of the move. BC contains zero.

```
00100 ;SUBROUTINE TO MOVE MEMORY
00110 ;   ENTRY:(HL)=SOURCE START
00120 ;         (DE)=DESTINATION START
00130 ;         (BC)=# OF BYTES TO MOVE
00140 ;    EXIT:(HL)=SOURCE AREA+1
00150 ;         (DE)=DEST AREA+1
00160 ;         (BC)=0
00170 ;
```

```
4A00            00180          ORG     4A00H           ;CHANGE ON REASSEMBLY
4A00 E5         00190 MOVE     PUSH    HL              ;SAVE SOURCE PNTR
4A01 B7         00200          OR      A               ;CLEAR CARRY
4A02 ED52       00210          SBC     HL,DE           ;SOURCE-DEST PNTRS
4A04 E1         00220          POP     HL              ;RESTORE PNTR
4A05 DA0C4A     00230          JP      C,MOV10         ;GO IF MOVE BACK
4A08 EDB0       00240          LDIR                    ;MOVE FORWARD
4A0A 1900       00250          JR      MOV20           ;GO TO RETURN
4A0C 09         00260 MOV10    ADD     HL,BC           ;POINT TO END+1
4A0D 2B         00270          DEC     HL              ;POINT TO END
4A0E EB         00280          EX      DE,HL           ;SWAP
4A0F 09         00290          ADD     HL,BC           ;POINT TO END+1
4A10 2B         00300          DEC     HL              ;POINT TO END
4A11 EB         00310          EX      DE,HL           ;SWAP BACK
4A12 EDB8       00320          LDDR                    ;MOVE BACK
4A14 C9         00330 MOV20    RET                     ;RETURN
0000            00340          END
00000 TOTAL ERRORS
MOV20   4A14
MOV10   4A0C
MOVE    4A00
```

MULADD Subroutine

MULADD is a subroutine to perform multiple-precision adds. Two multiple-precision operands from one to 256 bytes in length are added to each other, and the result is put into the destination operand. The source operand remains unchanged. The operands are located anywhere in memory desired, with the data arranged most significant byte through least significant byte from low memory through high memory. On entry the IX register points to the first byte (most significant) of the destination operand and IY points to the first byte (most significant) of the source operand. Both operands are treated as the same length. The B register contains the number of bytes in each operand from 1 through 255. An entry of B=0 is treated as a length of 256 bytes. On exit, the destination operand contains the result of the add. IX and IY are un-

changed. The B register is zeroed, and the A register is destroyed.

```
                00100 ;SUBROUTINE TO DO MULTIPLE-PRECISION ADDS
                00110 ;   ENTRY:(IX)=POINTS TO MS BYTE OF DESTINATION
                00120 ;         (IY)=POINTS TO MS BYTE OF SOURCE
                00130 ;         (B)=# OF BYTES IN OPERANDS
                00140 ;         CALL  MULADD
                00150 ;         (RETURN)
                00160 ;   EXIT:(IX)=UNCHANGED
                00170 ;         (IY)=UNCHANGED
                00180 ;         (A)=DESTROYED
                00190 ;         (B)=0
                00200 ;
4A00            00210           ORG     4A00H           ;CHANGE ON REASSEMBLY
4A00 D5         00220 MULADD    PUSH    DE              ;SAVE DE
4A01 58         00230           LD      E,B             ;#BYTES TO E
4A02 1600       00240           LD      D,0             ;DE NOW HAS #
4A04 1B         00250           DEC     DE              ;DE NOW HAS #-1
4A05 DD19       00260           ADD     IX,DE           ;POINT TO LS BYTE
4A07 FD19       00270           ADD     IY,DE           ;POINT TO LS BYTE
4A09 D1         00280           POP     DE              ;RESTORE ORIGINAL
4A0A AF         00290           XOR     A               ;RESET CARRY
4A0B DD7E00     00300 LOOP      LD      A,(IX)          ;GET DESTINATION
4A0E FD8E00     00310           ADC     A,(IY)          ;ADD SOURCE
4A11 DD7700     00320           LD      (IX),A          ;STORE RESULT
4A14 1001       00330           DJNZ    LOOP1           ;GO IF NOT DONE
4A16 C9         00340           RET                     ;RETURN
4A17 DD2B       00350 LOOP1     DEC     IX              ;PNT TO NEXT HIGHER
4A19 FD2B       00360           DEC     IY              ;PNT TO NEXT HIGHER
4A1B C30B4A     00370           JP      LOOP            ;CONTINUE
0000            00380           END
00000 TOTAL ERRORS
LOOP1   4A17
LOOP    4A0B
MULSUB  4A00
```

MULSUB Subroutine

The MULSUB subroutine performs multiple-precision subtracts. Two multiple-precision operands from one byte to 256 bytes in length are subtracted from each other, and the result is put into the destination operand memory locations. The source operand remains unchanged. The operands are located anywhere in memory desired, with the data arranged most significant byte through least significant byte from low memory through high memory. On entry the IX register points to the first byte (most significant) of the destination operand and IY points to the first byte (most significant) of the source operand. Both operands are treated as the same length. The B register contains the number of bytes in each operand from 1 through 255. An entry of B=0 is treated as a length of 256 bytes. On exit, the destination operand contains the result of the subtract. IX and IY are unchanged. The B register is zeroed, and the A register is destroyed.

```
                00100 ;SUBROUTINE TO DO MULTIPLE-PRECISION SUBTRACTS
                00110 ;   ENTRY:(IX)=POINTS TO MS BYTE OF DESTINATION
                00120 ;         (IY)=POINTS TO MS BYTE OF SOURCE
                00130 ;         (B)=# OF BYTES IN OPERANDS
                00140 ;         CALL  MULSUB
                00150 ;         (RETURN)
                00160 ;   EXIT:(IX)=UNCHANGED
                00170 ;        (IY)=UNCHANGED
                00180 ;        (A)=DESTROYED
                00190 ;        (B)=0
                00200 ;
4A00            00210         ORG     4A00H          ;CHANGE ON REASSEMBLY
4A00 D5         00220 MULSUB  PUSH    DE             ;SAVE DE
4A01 58         00230         LD      E,B            ;#BYTES TO E
4A02 1600       00240         LD      D,0            ;DE NOW HAS #
4A04 1B         00250         DEC     DE             ;DE NOW HAS #-1
4A05 DD19       00260         ADD     IX,DE          ;POINT TO LS BYTE
4A07 FD19       00270         ADD     IY,DE          ;POINT TO LS BYTE
4A09 D1         00280         POP     DE             ;RESTORE ORIGINAL
4A0A AF         00290         XOR     A              ;RESET CARRY
4A0B DD7E00     00300 LOOP    LD      A,(IX)         ;GET DESTINATION
```

```
400E FD9E00    00310          SBC    A,(IY)        ;SUBTRACT SOURCE
4011 DD7700    00320          LD     (IX),A        ;STORE RESULT
4014 1001      00330          DJNZ   LOOP1         ;GO IF NOT DONE
4016 C9        00340          RET                  ;RETURN
4017 DD2B      00350 LOOP1    DEC    IX            ;PNT TO NEXT HIGHER
4019 FD2B      00360          DEC    IY            ;PNT TO NEXT HIGHER
401B C30B4A    00370          JP     LOOP          ;CONTINUE
0000           00380          END
00000 TOTAL ERRORS
LOOP1   4017
LOOP    400B
MULSUB  4000
```

CMPARE Subroutine

The CMPARE subroutine compares two 8-bit operands in true algebraic fashion, that is, a −5 is less than a −1, and so forth. Three return points must be provided by the user after the CALL to CMPARE. Each return point must have a jump instruction of three bytes. The first return point is the return made when the A operand is less than the B operand. The second return point is the return made when the two operands are equal. The third return point is the return made when operand A is greater than operand B. By putting in jumps to the same areas, any combination of equalities may be constructed. For example, if the three return points have

```
JP  ONE   ;JUMP TO ONE ON LESS THAN
JP  ONE   ;JUMP TO ONE ON EQUAL
JP  TWO   ;JUMP TO TWO ON GREATER THAN
```

a jump will be made to location "ONE" if A is less than or equal to B, and a jump to location "TWO" will be made if A is greater than B.

On entry, the A register contains the first operand, and the B register contains the second. On exit, the return point is based on the comparison of A to B. A remains unchanged along with B, and the HL register is destroyed.

```
00000 TOTAL ERRORS
GREATR  4011
LESST   4013
DIFFER  4015
```

```
                00100 ;SUBROUTINE TO COMPARE TWO 8-BIT SIGNED OPERANDS
                00110 ;   ENTRY:(A)=OPERAND 1
                00120 ;         (B)=OPERAND 2
                00130 ;         CALL   CMPARE        ;CALL SR
                00140 ;         (RTN FOR A LT B)     ;PUT JP  LESST HERE
                00150 ;         (RRTN FOR A=B)       ;PUT JP  EQUAL HERE
                00160 ;         (RTN FOR A GT B)     ;PUT JP  GREATR HERE
                00170 ;   EXIT: (A)=UNCHANGED
                00180 ;         (B)=UNCHANGED
                00190 ;         (HL)=DESTROYED
                00200 ;
4A00            00210           ORG    4A00H         ;CHANGE ON REASSEMBLY
4A00 E1         00220 CMPARE    POP    HL            ;GET RTN ADDRESS
4A01 D5         00230           PUSH   DE            ;SAVE DE
4A02 110300     00240           LD     DE,3          ;ADDRESS INCREMENT
4A05 B8         00250           CP     B             ;COMPARE A:B
4A06 2808       00260           JR     Z,EQUAL       ;GO IF EQUAL
4A08 F5         00270           PUSH   AF            ;SAVE FLAGS
4A09 A8         00280           XOR    B             ;TEST SIGN BITS
4A0A 17         00290           RLA                  ;XOR TO C
4A0B DA154A     00300           JP     C,DIFFER      ;GO IF DIFFERENT SIGNS
4A0E F1         00310           POP    AF            ;RESTORE FLAGS
4A0F 3802       00320           JR     C,LESST       ;GO IF A LT B
4A11 19         00330 GREATR    ADD    HL,DE         ;BUMP RTN BY 3
4A12 19         00340 EQUAL     ADD    HL,DE         ;BUMP RTN BY 3
4A13 D1         00350 LESST     POP    DE            ;RESTORE DE
4A14 E9         00360           JP     (HL)          ;RTN TO 0,3,6
4A15 F1         00370 DIFFER    POP    AF            ;RESTORE FLAGS
4A16 DA114A     00380           JP     C,GREATR      ;GO IF A GT B
4A19 C3134A     00390           JP     LESST         ;A LT B
0000            00400           END
EQUAL   4A12
CMPARE  4A00
```

MUL16 Subroutine

The MUL16 multiplies an unsigned 16-bit number in the DE register by an unsigned 8-bit number in the B register,

putting the result in the HL register. As the numbers are unsigned, DE may hold from 0 through 65535 and B may hold from 0 through 255. Overflow may result if the product is too large to be held in 16 bits. There is no check on overflow. On entry, DE contains the 16-bit multiplicand and B contains the 8-bit multiplier. On exit, HL contains the 16-bit product, DE is destroyed, and B contains 0.

```
                00100 ;SUBROUTINE TO MULTIPLY 16 BY 8
                00110 ;   ENTRY:(DE)=MULTIPLICAND,UNSIGNED
                00120 ;         (B)=MULTIPLIER,UNSIGNED
                00130 ;         CALL  MUL16
                00140 ;   EXIT: (HL)=PRODUCT
                00150 ;         (DE)=DESTROYED
                00160 ;         (B)=0
                00170 ;
4000            00180           ORG     4000H           ;CHANGE ON REASSEMBLY
4000 210000     00190 MUL16     LD      HL,0            ;CLEAR PARTIAL PRODUCT
4003 CB38       00200 LOOP      SRL     B               ;SHIFT OUT M'IER BIT
4005 3001       00210           JR      NC,CONT         ;GO IF NO CARRY (1 BIT)
4007 19         00220           ADD     HL,DE           ;ADD MULTIPLICAND
4008 C8         00230 CONT      RET     Z               ;GO IF M'IER
4009 EB         00240           EX      DE,HL           ;MULTIPLICAND TO HL
400A 29         00250           ADD     HL,HL           ;SHIFT MULTIPLICAND
400B EB         00260           EX      DE,HL           ;SWAP BACK
400C C30340     00270           JP      LOOP            ;CONTINUE
0000            00280           END
00000 TOTAL ERRORS
CONT    4008
LOOP    4003
MUL16   4000
```

DIV16 Subroutine

The DIV16 subroutine divides an unsigned 16-bit number in the HL register pair by an unsigned 8-bit number in the D register. The quotient is placed in the IX register and the remainder is left in H. As the numbers are unsigned, the dividend in HL may be 0 through 65535 and the divisor in D may

be 0 through 255. Overflow will result on division by zero. The remainder is less than the divisor. On entry, HL contains the 16-bit dividend and D contains the 8-bit divisor. On exit, IX contains the 16-bit quotient and H contains the 8-bit remainder. The L and A registers are destroyed, E is zeroed, and the D register is unchanged.

```
00100 ;SUBROUTINE TO DIVIDE 16 BY 8
00110 ;   ENTRY:(HL)=DIVIDEND 16 BITS
00120 ;         (D)=DIVISOR 8 BITS
00125 ;         CALL  DIV16
00130 ;   EXIT: (IX)=QUOTIENT 16 BITS
00140 ;         (H)=REMAINDER 8 BITS
00150 ;         (L)=DESTROYED
00160 ;         (D)=UNCHANGED
00170 ;         (E)=0
00180 ;         (A)=DESTROYED
00190 ;
4A00            00200         ORG     4A00H          ;CHANGE ON REASSEMBLY
4A00 7D         00210 DIV16   LD      A,L            ;LS BYTE DIVDND
4A01 6C         00220         LD      L,H            ;MS BYTE DIVDND
4A02 2600       00230         LD      H,0            ;CLEAR FOR SUBT
4A04 1E00       00240         LD      E,0            ;SETUP FOR SUBTRACT
4A06 0610       00250         LD      B,16           ;16 ITERATIONS
4A08 DD210000   00260         LD      IX,0           ;INITIALIZE QUOTIENT
4A0C 29         00270 LOOP    ADD     HL,HL          ;SHIFT DIVD LEFT
4A0D 17         00280         RLA                    ;SHIFT 8 LS BITS
4A0E D2124A     00290         JP      NC,LOOP1       ;GO IF 0 BIT
4A11 2C         00300         INC     L              ;SHIFT TO HL
4A12 DD29       00310 LOOP1   ADD     IX,IX          ;SHIFT QUOTIENT LEFT
4A14 DD23       00320         INC     IX             ;0 BIT=1
4A16 B7         00330         OR      A              ;CLEAR CARRY FOR SUB
4A17 ED52       00340         SBC     HL,DE          ;TRY SUBTRACT
4A19 D21F4A     00350         JP      NC,CONT        ;GO IF IT WENT
4A1C 19         00360         ADD     HL,DE          ;RESTORE
4A1D DD2B       00370         DEC     IX             ;SET Q BIT=0
4A1F 10EB       00380 CONT    DJNZ    LOOP           ;GO IF NOT 16
```

```
4A21 C9        00390        RET                    RETURN
0000           00400        END
00000 TOTAL ERRORS
CONT    4A1F
LOOP1   4A12
LOOP    4A0C
DIV16   4A00
```

HEXCV Subroutine

HEXCV is a subroutine to convert an 8-bit value (two hexadecimal digits) into two ASCII characters representing the hexadecimal characters 0 through 9 and A through F. On entry, the A register contains the 8-bit value to be converted. On exit, the H register contains an ASCII character representing the hex character for the four high-order bits and the L register contains an ASCII character representing the hex character for the four low-order bits. The A and C registers are destroyed.

```
                00100 ;SUBROUTINE TO CONVERT FROM HEX TO ASCII
                00110 ;
                00120 ;   ENTRY:(A)=8-BIT VALUE TO BE CONVERTED
                00130 ;         CALL   HEXCV
                00140 ;         (RETURN)
                00150 ;   EXIT: (HL)=TWO ASCII VALUES, HIGH AND LOW
                00160 ;         (A)=DESTROYED
                00170 ;         (C)=DESTROYED
                00180 ;
4A00            00190         ORG    4A00H          ;CHANGE ON REASSEMBLY
4A00 4F         00200 HEXCV   LD     C,A            ;SAVE TWO HEX DIGITS
4A01 CB3F       00210         SRL    A              ;ALIGN HIGH DIGIT
4A03 CB3F       00220         SRL    A
4A05 CB3F       00230         SRL    A
4A07 CB3F       00240         SRL    A
4A09 CD154A     00250         CALL   TEST           ;CONVERT TO ASCII
4A0C 67         00260         LD     H,A            ;SAVE FOR RTN
4A0D 79         00270         LD     A,C            ;RESTORE ORIGINAL
4A0E E60F       00280         AND    0FH            ;GET LOW DIGIT
```

```
4A10 CD154A   00290        CALL   TEST        ;CONVERT TO ASCII
4A13 6F       00300        LD     L,A         ;SAVE FOR RTN
4A14 C9       00310        RET
4A15 C630     00320 TEST   ADD    A,30H       ;CONVERSION FACTOR
4A17 FE3A     00330        CP     3AH         ;TEST FOR 0-9
4A19 FA1E4A   00340        JP     M,TEST1     ;GO IF 0-9
4A1C C607     00350        ADD    A,7         ;CORRECT FOR A-F
4A1E C9       00360 TEST1  RET                ;RETURN
0000          00370        END
00000 TOTAL ERRORS
TEST1  4A1E
TEST   4A15
HEXCV  4A00
```

SEARCH Subroutine

The SEARCH subroutine searches a table for a given *key* value of 8 bits. The table may be any number of entries from 1 through 256, with each entry a *fixed-length* of one to n bytes. The table may be located anywhere in memory. The key value is assumed to be the first byte in each entry.

On entry, the A register holds the 8-bit key of 0 through 255. HL points to the start of the table in memory. DE contains the length of each entry in bytes. The C register holds the number of entries in the table, from 1 through 255. On exit, the Z flag is set if the key has been found in the table, and the HL register points to the entry containing the matching value in this case. If the key is not found, the Z flag is not set upon return. If the key is found, BC contains the current number of entries *left* in the table. In this case the subroutine may be called again to search for another occurrence of the key, without changing the contents of HL, DE, BC, or A.

SET, RESET, and TEST Subroutines

These subroutines are used to set, reset, and test a point on the screen in similar fashion to SET, RESET, and POINT in BASIC. The screen coordinate values given are converted into corresponding memory locations in video memory, which are

```
                00100 ;SUBROUTINE FOR TABLE SEARCH
                00110 ;   ENTRY:(A)=KEY
                00120 ;         (HL)=TABLE START
                00130 ;         (DE)=LENGTH OF EACH ENTRY IN BYTES
                00140 ;         (C)=# OF ENTRIES IN TABLE
                00150 ;         CALL  SEARCH
                00160 ;   EXIT: Z FLAG SET IF FOUND,NOT SET IF NOT FOUND
                00170 ;         (HL)=LOCATION OF MATCH IF FOUND
                00180 ;         (BC)=CURRENT # LEFT
                00190 ;         (DE)=UNCHANGED
                00200 ;
4A00            00210         ORG     4A00H         ;CHANGE ON REASSEMBLY
4A00 0600       00220 SEARCH  LD      B,0           ;BC NOW HAS #
4A02 EDA1       00230 LOOP    CPI                   ;COMPARE A WITH (HL)
4A04 CA0E4A     00240         JP      Z,FOUND       ;GO IF FOUND
4A07 E20F4A     00250         JP      PO,NFND       ;AT END AND NOT FND
4A0A 19         00260         ADD     HL,DE         ;CURRENT+LENGTH+1
4A0B 2B         00270         DEC     HL            ;CURRENT +LENGTH
4A0C 18F4       00280         JR      LOOP          ;TRY AGAIN
4A0E 2B         00290 FOUND   DEC     HL            ;ADJUST TO FOUND LOC
4A0F C9         00300 NFND    RET                   ;RETURN
0000            00310         END
00000 TOTAL ERRORS
NFND    4A0F
FOUND   4A0E
LOOP    4A02
SEARCH  4A00
```

then processed. The high-order bit of each memory location is set when any of the three subroutines is called, on the assumption that the coordinates addressed represent graphics points.

On entry, DE contains the y,x coordinates. The D register contains the Y value of 0 through 47, while the E register holds the X value of 0 through 127. A CALL is made to SET, RESET, or TEST to set, reset, or test the coordinate specified.

On exit, the A, B, C, D, E, H, and L registers are destroyed. If a test was involved, the Z flag is set if the point was a zero and reset if the point was a one.

Care must be taken in using this subroutine to make certain that the x and y values are in the range given, as the subroutine may wreak havoc if invalid values are input.

```
                00100 ;SUBROUTINE TO CONVERT SCREEN COORDINATES
                00110 ;   ENTRY:(DE)=Y,X COORDINATES OF POINT
                00120 ;              X=0 TO 127;Y=0 TO 47
                00130 ;         CALL   SET      ;SETS POINT
                00140 ;         CALL   RESET    ;RESETS POINT
                00150 ;         CALL   TEST     ;TESTS POINT RETURNS Z FLAG
                00160 ;   EXIT: (A THROUGH L)=DESTROYED
                00170 ;         Z FLAG SET IF TEST
                00180 ;
4A00            00190          ORG    4A00H
4A00 3EC6       00200 SET      LD     A,0C6H          ;SET B,(HL) INSTRUCTION
4A02 1806       00210          JR     TEST10          ;GO TO STORE
4A04 3E86       00220 RESET    LD     A,86H           ;RES B,(HL) INSTRUCTION
4A06 1802       00230          JR     TEST10          ;GO TO STORE
4A08 3E46       00240 TEST     LD     A,46H           ;BIT B,(HL) INSTRUCTION
4A0A 32D04A     00250 TEST10   LD     (INST+1),A      ;STORE 2ND BYTE
4A0D 7A         00260 ADDRES   LD     A,D             ;GET Y
4A0E 06FF       00270          LD     B,0FFH          ;-1
4A10 04         00280 LOOP     INC    B               ;SUCCESIVE SUBS FOR DIV
4A11 D603       00290          SUB    3               ;  BY THREE
4A13 F2104A     00300          JP     P,LOOP          ;GO IF NOT MINUS
4A16 C603       00310          ADD    A,3             ;YQ IN B, YR IN A
4A18 CB27       00320          SLA    A               ;YR*2
4A1A 4F         00330          LD     C,A             ;SAVE YR*2
4A1B 68         00340          LD     L,B             ;YQ TO L
4A1C 2600       00350          LD     H,0             ;YQ IN HL
4A1E 0606       00360          LD     B,6             ;CNT FOR MULTIPLY BY 64
4A20 29         00370 LOOP1    ADD    HL,HL           ;YQ*2
```

```
4021 10FD     00380        DJNZ   LOOP1          ;GO IF NOT Y0=64
4023 1600     00390        LD     D,0            ;DE NOW HAS X
4025 CB3B     00400        SRL    E              ;X0
4027 3001     00410        JR     NC,CONT        ;GO IF XR NE 1
4029 0C       00420        INC    C              ;C NOW HAS YR+2+XR
402A 19       00430 CONT   ADD    HL,DE          ;HL NOW HAS Y0+2+X0
402B 11003C   00440        LD     DE,3C00H       ;START OF DISPLAY
402E 19       00450        ADD    HL,DE          ;HL NOW HAS DISPLACE
402F CB21     00460        SLA    C              ;ALIGN TO FIELD
4031 CB21     00470        SLA    C
4033 CB21     00480        SLA    C
4035 3A3D4A   00490        LD     A,(INST+1)     ;GET INSTRUCTION
4038 81       00500        ADD    A,C            ;SET FIELD
4039 323D4A   00510        LD     (INST+1),A     ;STORE
403C CB       00520 INST   DEFB   0CBH           ;PERFORM BIT,SET,RES
403D 00       00530        DEFB   0              ;WILL BE FILLED IN
403E CBFE     00540        SET    7,(HL)         ;FOR GRAPHICS
4040 C9       00550        RET
0000          00560        END
00000 TOTAL ERRORS
CONT    402A
LOOP1   4020
LOOP    4010
MEMRES  400D
INST    403C
TEST    4006
RESET   4004
TEST10  400A
SET     4000
```

SECTION III

Appendices

APPENDIX I

Z-80 Instruction Set

A Register Operations

Complement CPL
Decimal DAA
Negate NEG

Adding/Subtracting Two 8-Bit Numbers

A and Another Register

ADC A,r	SBC A,r
ADD A,r	SUB A,r

A and Immediate Operand

ADC A,n	SBC A,n
ADD A,n	SUB A,n

A and Memory Operand

ADC A,(HL)	ADD A,(HL)	SBC (HL)	SUB (HL)
ADC A,(IX+d)	ADD A,(IX+d)	SBC (IX+d)	SUB (IX+d)
ADC A,(IY+d)	ADD A,(IY+d)	SBC (IY+d)	SUB (IY+d)

Adding/Subtracting Two 16-Bit Numbers

HL and Another Register Pair
ADC HL,ss ADD HL,ss SBC HL,ss
IX and Another Register Pair
ADD IX,pp ADD IY,rr

Bit Instructions

Test Bit			
Register	BIT b,r		
Memory	BIT b,(HL)	BIT b,(IX+d)	BIT b,(IY+)
Reset Bit			
Register	RES b,r		
Memory	RES b,(HL)	RES b,(IX+d)	RES b,(IY+d)
Set Bit			
Register	SET b,r		
Memory	SET b,(HL)	SET b,(IX+d)	SET b,(IY+d)

Carry Flag

Complement CCF
Set SCF

Compare Two 8-Bit Operands

A and Another Register CP r
A and Immediate Operand CP n
A and Memory Operand
CP (HL) CP (IX+d) CP (IY+d)
Block Compare
CPD,CPDR,CPI,CPIR

Decrements and Increments

Single Register
DEC r INC r DEC IX DEC IY INC
Register Pair
DEC ss INC ss DEC IX DEC IY INC IX DEC IY
Memory
DEC HL DEC (IX+d) DEC (IY+d)

Exchanges

DE and HL EX DE,HL
Top of Stack
EX (SP),HL EX (SP),IX EX (SP),IY

Input/Output

I/O To/From A and Port
IN A,(n) OUT (n),A
I/O To/From Register and Port
IN r,(C) OUT (C),r
Block
IND,INDR,INR,INIR,OTDR,OTIR,OUTD,OUTI

Interrupts

Disable DI
Enable EI
Interrupt Mode
IM 0 IM 1 IM 2
Return From Interrupt
RETI RETN

Jumps

Unconditional
JP (HL) JP (IX) JP (IY) JP (nn) JR e
Conditional
JP cc,nn JR C,e JR NZ,e JR Z,e
Special Conditional
DJNZ e

Loads

A Load Memory Operand
LD A,(BC) LD A,(DE) LD A,(nn)

A and Other Registers
  LD A,I LD A,R LD I,A LD R,A
Between Registers, 8-Bit
  LD r,r'
Immediate 8-Bit
  LD r,n
Immediate 16-Bit
  LD dd,nn LD IX,nn LD IY,nn
Register Pairs From Other Register Pairs
  LD SP,HL LD SP,IX LD SP,IY
From Memory, 8-Bits
  LD r,(HL) LD r,(IX+d) LD r,(IY+d)
From Memory, 16-Bits
  LD HL,(nn) LD IX,(nn) LD IY,(nn) LD dd,(nn)
Block
  LDD,LDDR,LDI,LDIR

Logical Operations 8 Bits With A

A and Another Register
  AND r OR r XOR r
A and Immediate Operand
  AND n OR n XOR n
A and Memory Operand
  AND (HL) OR (HL) XOR (HL)
  AND (IX+d) OR (IX+d) XOR (IX+d)
  AND (IY+d) OR (IY+d) XOR (IY+d)

Miscellaneous

Halt HALT
No Operation NOP

Prime/Non-Prime

Switch AF
  EX AF,AF'
Switch Others
  EXX

Shifts

Circular (Rotate)
  A Only RLA, RLCA, RRA, RRCA
  All Registers RL r RLC r RR r RRC r
  Memory
    RL (HL) RLC (HL) RR (HL) RRC (HL)
    RL (IX+d) RLC (IX+d) RR (IX+d) RRC (IX+d)
    RL (IY+d) RLC (IY+d) RR (IY+d) RRC (IY+d)
Logical
  Registers SRL r
  Memory SRL (HL) SRL (IX+d) SRL (IY+d)
Arithmetic
  Registers SLA r SRA r
  Memory
    SLA (HL) SRA (HL)
    SLA (IX+d) SRA (IX+d)
    SLA (IY+d) SRA (IY+d)

Stack Operations

PUSH IX PUSH IY PUSH qq POP IX POP IY POP qq

Stores

Of A Only
LD (BC),A LD (DE),A LD (HL),A LD (nn),A
All Registers
LD (HL),r LD (IX+d),r LD (IY+d),r
Immediate Data
LD (HL),n LD (IX+d),n LD (IY+d),n
16-Bit Registers
LD ((nn),dd LD (nn),IX LD (nn),IY

Subroutine Action

Conditional CALLs CALL cc,nn
Unconditional CALLs CALL nn
Conditional Return RET cc
Unconditional Return RET cc
Special CALL RST p

APPENDIX II

Z-80 Operation Code Listings

Mnemonic	Format			Description	S	Z	P/V	C
ADC HL,ss	11101101	01ss1010		HL + ss + CY to HL	●	●	●	●
ADC A,r	10001 r			A + r + CY to A	●	●	●	●
ADC A,n	11001110	n		A + n + CY to A	●	●	●	●
ADC A,(HL)	10001110			A + (HL) + CY to A	●	●	●	●
ADC A,(IX + d)	11011101	10001110	d	A + (IX + d) + CY to A	●	●	●	●
ADC A,(IY + d)	11111101	10001110	d	A + (IY + d) + CY to A	●	●	●	●
ADD A,n	11000110	n		A + n to A	●	●	●	●
ADD A,r	10000 r			A + r to A	●	●	●	●
ADD A,(HL)	10000110			A + (HL) to A	●	●	●	●
ADD A,(IX + d)	11011101	10000110	d	A + (IX + d) to A	●	●	●	●
ADD A,(IY + d)	11111101	10000110	d	A + (IY + d) to A	●	●	●	●
ADD HL,ss	00ss1001			HL + ss to HL				●
ADD IX,pp	11011101	00pp1001		IX + pp to IX				●
ADD IY,rr	11111101	00rr1001		IY + rr to IY				●
AND r	10100 r			A AND r to A	●	●	●	0
AND n	11100110	n		A AND n to A	●	●	●	0
AND (HL)	10100110			A AND (HL) to A	●	●	●	0
AND (IX + d)	11011101	10100110	d	A AND (IX + d) to A	●	●	●	0
AND (IY + d)	11111101	10100110	d	A AND (IY + d) to A	●	●	●	0

BIT b,r	11001011	01 b r			Test bit b of r	●	●	●		
BIT b,(HL)	11001011	01 b 110			Test bit b of (HL)	●	●	●		
BIT b,(IX+d)	11011101	11001011	d	01 b 110	Test bit b of (IX+d)	●	●	●		
BIT b,(IY+d)	11111101	11001011	d	01 b 110	Test bit b of (IY+d)	●	●	●		
CALL cc,nn	11 c 100	n	n		CALL subroutine at nn if cc					
CALL nn	11001101	n	n		Unconditionally CALL nn					
CCF	00111111				Complement carry flag					●
CP r	10111 r				Compare A:r	●	●	●	●	
CP n	11111110	n			Compare A:n	●	●	●	●	
CP (HL)	10111110				Compare A:(HL)	●	●	●	●	
CP (IX+d)	11011101	10111110	d		Compare A:(IX+d)	●	●	●	●	
CP (IY+d)	11111101	10111110	d		Compare A:(IY+d)	●	●	●	●	
CPD	11101101	10101001			Block Compare, no repeat	●	●	●		
CPDR	11101101	10111001			Block Compare, repeat	●	●	●		
CPI	11101101	10100001			Block Compare, no repeat	●	●	●		
CPIR	11101101	10110001			Block Compare, repeat	●	●	●		
CPL	00101111				Complement A (1's comple)					
DAA	00100111				Decimal Adjust A	●	●	●		
DEC r	00 r 101				Decrement r by one	●	●	●		
DEC (HL)	00110101				Decrement (HL) by one	●	●	●		

Mnemonic	Format			Description	S	Z	P/V	C
DEC (IX+d)	11011101	00110101	d	Decrement (IX+d) by one	●	●	●	
DEC (IY+d)	11111101	00110101	d	Decrement (IY+d) by one	●	●	●	
DEC IX	11011101	00101011		Decrement IX by one				
DEC IY	11111101	00101011		Decrement IY by one				
DEC ss	00ss1011			Decrement register pair				
DI	11110011			Disable interrupts				
DJNZ e	00010000	e-2		Decrement B and JR if B≠0				
EI	11111011			Enable interrupts				
EX (SP),HL	11100011			Exchange (SP) and HL				
EX (SP),IX	11011101	11100011		Exchange (SP) and IX				
EX (SP),IY	11111101	11100011		Exchange (SP) and IY				
EX AF,AF'	00001000			Set prime AF active				
EX DE,HL	11101011			Exchange DE and HL				
EXX	11011001			Set prime B-L active				
HALT	01110110			Halt				
IM 0	11101101	01000110		Set interrupt mode 0				
IM 1	11101101	01010110		Set interrupt mode 1				
IM 2	11101101	01011110		Set interrupt mode 2				
IN A,(n)	11011011	n		Load A with input from n				

IN r,(C)	11101101	01 r 000		Load r with input from (C)	●	●	●
INC r	00 r 100			Increment r by one	●	●	●
INC (HL)	00110100			Increment (HL) by one	●	●	●
INC (IX+d)	11011101	00110100	d	Increment (IX+d) by one	●	●	●
INC (IY+d)	11111101	00110100	d	Increment (IY+d) by one	●	●	●
INC IX	11011101	00100011		Increment IX by one			
INC IY	11111101	00100011		Increment IY by one			
INC ss	00ss0011			Increment register pair			
IND	11101101	10101010		Block I/O input from (C)	●	●	●
INDR	11101101	10111010		Block I/O input, repeat	●	●	●
INI	11101101	10100010		Block I/O input from (C)	●	●	●
INIR	11101101	10110010		Block I/O input, repeat	●	●	●
JP (HL)	11101001			Unconditional jump to (HL)			
JP (IX)	11011101	11101001		Unconditional jump to (IX)			
JP (IY)	11111101	11101001		Unconditional jump to (IY)			
JP cc,nn	11 c 010	n	n	Jump to nn if cc			
JP nn	11000011	n	n	Unconditional jump to nn			
JR C,e	00111000	e-2		Jump relative if carry			
JR e	00011000	e-2		Unconditional jump relative			
JR NC,e	00110000	e-2		Jump relative if no carry			

Mnemonic	Format				Description	S	Z	P/V	C
JR NZ,e	00100000	e-2			Jump relative if non-zero				
JR Z,e	00101000	e-2			Jump relative if zero				
LD A,(BC)	00001010				Load A with (BC)				
LD A,(DE)	00011010				Load A with (DE)				
LD A,I	11101101	01010111			Load A with I	●	●	●	
LD A,(nn)	00111010	n	n		Load A with location nn				
LD A,R	11101101	01011111			Load A with R	●	●	●	
LD (BC),A	00000010				Store A to (BC)				
LD (DE),A	00010010				Store A to (DE)				
LD (HL),n	00110110	n			Store n to (HL)				
LD dd,nn	00dd0001	n	n		Load register pair with nn				
LD dd,(nn)	11101101	01dd1011	n	n	Load register pair with location nn				
LD HL,(nn)	00101010	n	n		Load HL with location nn				
LD (HL),r	01110 r				Store r to (HL)				
LD I,A	11101011	01000111			Load I with A				
LD IX,(nn)	11011101	00101010	n	n	Load IX with nn				
LD IX,nn	11011101	00100001	n	n	Load IX with location nn				
LD (IX+d),n	11011101	00110110	d	n	Store n to (IX+d)				
LD (IX+d),r	11011101	01110 r	d		Store r to (IX+d)				

LD IY,nn	11111101	00100001	n	n	Load IY with nn
LD IY,(nn)	11111101	00101010	n	n	Load IY with location nn
LD (IY+d),n	11111101	00110110	d	n	Store n to (IY+d)
LD (IY+d),r	11111101	01110 r	d		Store r to (IY+d)
LD (nn),A	00110010	n	n		Store A to location nn
LD (nn),dd	11101101	01dd0011	n	n	Store register pair to loc'n nn
LD (nn),HL	00100010	n	n		Store HL to location nn
LD (nn),IX	11011101	00100010	n	n	Store IX to location nn
LD (nn),IY	11111101	00100010	n	n	Store IY to location nn
LD R,A	11101101	01001111			Load R with A
LD r,r'	01 r r'				Load r with r'
LD r,n	00 r 110	n			Load r with n
LD r,(HL)	01 r 110				Load r with (HL)
LD r,(IX+d)	11011101	01 r 110	d		Load r with (IX+d)
LD r,(IY+d)	11111101	01 r 110	d		Load rf with (IY+d)
LD SP,HL	11111001				Load SP with HL
LD SP,IX	11011101	11111001			Load SP with IX
LD SP,IY	11111101	11111001			Load SP with IY
LDD	11101101	10101000			Block load, f'ward, no repeat ●
LDDR	11101101	10111000			Block load, f'ward, repeat 0

Mnemonic	Format			Description	S	Z	P/V	C
LDI	11101101	1010000		Block load, b'ward, no repeat			●	
LDIR	11101101	10110000		Block load b'ward, repeat			0	
NEG	11101101	01000100		Negate A (two's complement)	●	●	●	●
NOP	00000000			No operation				
OR r	10110 r			A OR r to A	●	●	●	0
OR n	11110110	n		A OR n to A	●	●	●	0
OR (HL)	10110110			A OR (HL) to A	●	●	●	0
OR (IX+d)	11011101	10110110	d	A OR (IX+d) to A	●	●	●	0
OR (IY+d)	11111101	10110110	d	A OR (IY+d) to A	●	●	●	0
OTDR	11101101	10111011		Block output, b'ward, repeat	●	●	●	
OTIR	11101101	10110011		Block output, f'ward, repeat	●	●	●	
OUT (C),r	11101101	01 r 001		Output r to (C)				
OUT (n),A	11010011	n		Output A to port n				
OUTD	11101101	10101011		Block output, b'ward, no rpt	●	●	●	
OUTI	11101101	10100011		Block output, f'ward, no rpt	●	●	●	
POP IX	11011101	11100001		Pop IX from stack				
POP IY	11111101	11100001		Pop IY from stack				
POP qq	11qq0001			Pop qq from stack				
PUSH IX	11011101	11100101		Push IX onto stack				

PUSH IY	11111101	11100101			Push IY onto stack				
PUSH qq	11qq0101				Push qq onto stack				
RES b,r	11001011	10 b r			Reset bit b of r				
RES b,(HL)	11001011	10 b 110			Reset bit b of (HL)				
RES b,(IX+d)	11011101	11001011	d	10 b 110	Reset bit b of (IX+d)				
RES b,(IY+d)	11111101	11001011	d	10 b 110	Reset bit b of (IY+d)				
RET	11001001				Return from subroutine				
RET cc	11 c 000				Return from subroutine if cc				
RETI	11101101	01001101			Return from interrupt				
RETN	11101101	01000101			Return from non-maskable int				
RL r	11001011	00010 r			Rotate left thru carry r	●	●	●	●
RL (HL)	11001011	00010110			Rotate left thru carry (HL)	●	●	●	●
RL (IX+d)	11011101	11001011	d	00010110	Rotate left thru carry (IY+d)	●	●	●	●
RL (IY+d)	11010101	11001011	d	00000110	Rotate left thru carry (IY+d)	●	●	●	●
RLA	00010111				Rotate A left thru carry				●
RLC r	11001011	00000 r			Rotate left circular r	●	●	●	●
RLC (HL)	11001011	00000110			Rotate left circular (HL)	●	●	●	●
RLC (IX+d)	11011101	11001011	d	00000110	Rotate left circular (IX+d)	●	●	●	●
RLC (IY+d)	11111101	11001011	d	00000110	Rotate left circular (IY+d)	●	●	●	●
RLCA	00000111				Rotate left circular A				●

SCF	00110111				Set carry flag				1
SET b,(HL)	11001011	11 b 110			Set bit b of (HL)				
SET b,(IX+d)	11011101	11001011	d	11 b 110	Set bit b of (IX+d)				
SET b,(IY+d)	11111101	11001011	d	11 b 110	Set bit b of (IY+d)				
SET b,r	11001011	11 b r			Set bit b of r				
SLA r	11001011	00100 r			Shift r left arithmetic	●	●	●	●
SLA (HL)	11001011	00100110			Shift (HL) left arithmetic	●	●	●	●
SLA (IX+d)	11011101	11001011	d	00100110	Shift (IX+d) left arithmetic	●	●	●	●
SLA (IY+d)	11111101	11001011	d	00100110	Shift (IY+d) left arithmetic	●	●	●	●
SRA r	11001011	00101 r			Shift r right arithmetic	●	●	●	●
SRA (HL)	11001011	00101110			Shift (HL) right arithmetic	●	●	●	●
SRA IX+d)	11011101	11001011	d	00101110	Shift (IX+d) right arithmetic	●	●	●	●
SRA (IY+d)	11111101	11001011	d	00101110	Shift (IY+d) right arithmetic	●	●	●	●
SRL r	11001011	00111 r			Shift r right logical	●	●	●	●
SRL (HL)	11001011	00111110			Shift (HL) right arithmetic	●	●	●	●
SRL (IX+d)	11011101	11001011	d	00111110	Shift (IX+d) right arithmetic	●	●	●	●
SRL (IY+d)	11111101	11001011	d	00111110	Shift (IY+d) right arithmetic	●	●	●	●
SUB r	10010 r				A-r to A	●	●	●	●
SUB n	11010110	n			A-n to A	●	●	●	●
SUB (HL)	10010110				A-(HL) to A	●	●	●	●

Index